OS PROJECT REPORT

MACH IPC IN LINUX

Teacher: Dr. Ghufran Ahmed(Theory)

Sir Amin Sadiq(Lab)

Group Members: Muhammad Moaaz bin Sajjad (20k-0154)

Saad bin Khalid (20k-0161)

Muhammad Ahmed Jamil (20k-0388)

Table of Contents

Abstract

This project aims to implement the Mach inter-process communication IPC mechanism within a traditional Linux environment.

Overview

The Mach IPC is a message-passing communication model. All inter-task communication is carried out by messages. Messages are sent to, and received from, mailboxes, which are called ports in Mach. A message is sent to one port, and a response is sent to a separate reply port. Message passing may occur between any two ports on the same host or on separate hosts on a distributed system. Each task also has access to a *central* or *bootstrap* server, which is responsible for storing and distributing information of running tasks. As messages are sent to the port, the messages are copied into a queue, which can be read by the receiver thread.

Technology Used

The UNIX networking and sockets APIs are used to implement the communication channels between processes.

Implementation

The central server is the first to execute. The central server first binds its listen port at port number 3333. It then awaits any connection requests. When it receives a request, it connects to said peer and adds it to the list of connected clients. It also constantly checks all the connected clients for any requests. Any received requests are serviced accordingly. There are four different types of requests:

1. *innit*: Request by newly created processes to get themselves registered with the central server. Upon receiving this request, the central server connects to the process and adds it to the list of connected peers.
2. *exit*: Request by terminating processes to get themselves removed from the server's list of connected peers. Upon receiving this request, the server

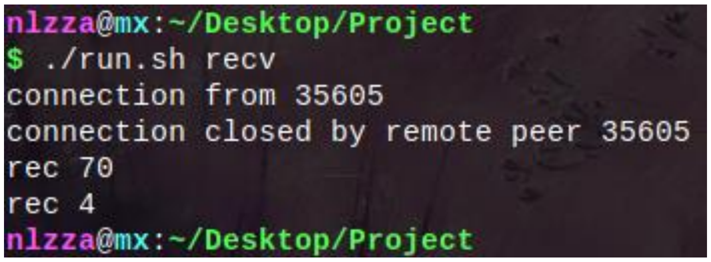
searches its list of connected clients until it finds one with matching key; this client is then removed from the list of connected peers.

3. *peers*: Request by client to know all the available peers on the network. The server first sends the client the number of available peers and then sends information of each peer one by one.
4. *closeserver*: Request by client to terminate server, thus ending the program. Upon receiving this request, the server terminates.

Now that we have established the working of the central server, let's dive into the two communicating processes: the sender and the receiver.

The sending process is executed before the receiving one. In our test file (send.cpp), the sender has ID 1. The sender waits to connect to the receiver. Once the connection is established, it sends a message. In the aforementioned test file, the message is an integer with the value of 70. However, in reality, the message can be anything. The test files are for demonstration only.

Lastly, we execute the receiver. In our test file (recv.cpp), the receiver has ID 2. The receiver waits until it receives the message. The message is then displayed on the screen. This helps verify whether the message has been successfully received or not. In the case of our test files, a successful execution results in the following message being displayed:

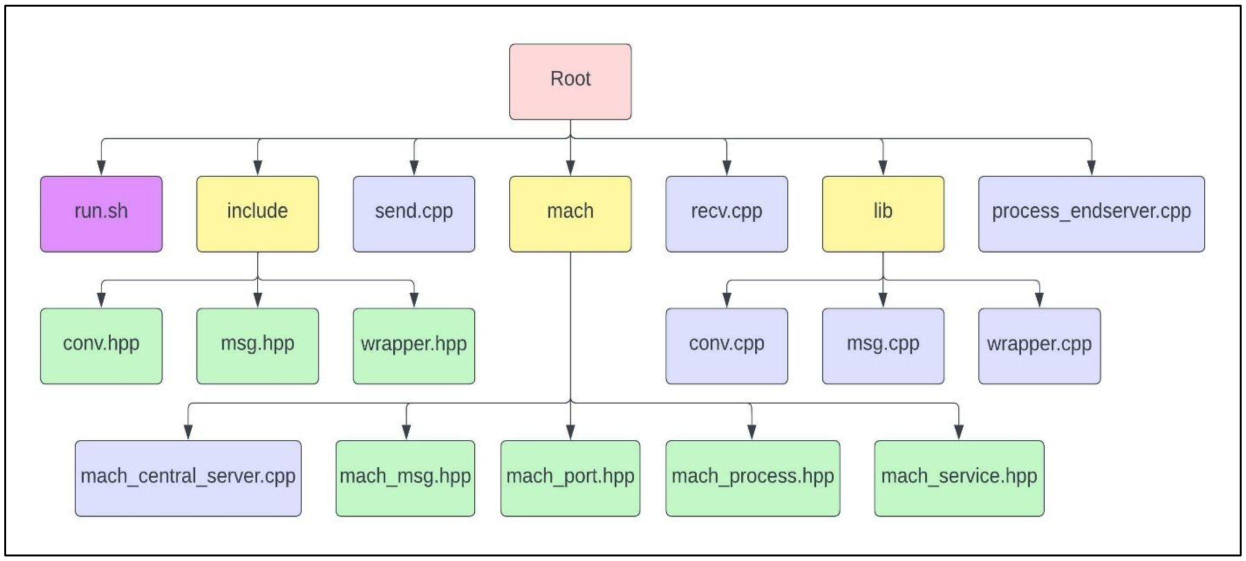


```
nlzza@mx:~/Desktop/Project
$ ./run.sh recv
connection from 35605
connection closed by remote peer 35605
rec 70
rec 4
nlzza@mx:~/Desktop/Project
```

Now that the basic working has been explained, we take a deeper look at each and every component of our project.

Directory Structure

The project has been divided into directories and subdirectories as illustrated on the next page.



Now let’s go over the files and directories one by one:

send.cpp

The sender process.

recv.cpp

The receiver process.

process_endserver.cpp

Sends a signal to terminate the central server.

run.sh

The shell script used to run our project. This has been described in greater detail here.

include

Contains all necessary declarations of function prototypes which are used throughout the project. It contains *conv.hpp*, *msg.hpp*, and *wrapper.hpp*.

conv.hpp

Contains prototypes of functions used for converting an IPv4 address from string to network byte value and vice-versa.

msg.hpp

Contains prototypes of functions used for reading from and writing to a socket.

wrapper.hpp

Contains prototypes of various wrapper functions, hence the name.

lib

Contains all the required definitions of functions which are used throughout the project. It contains *conv.cpp*, *msg.cpp*, and *wrapper.cpp*.

conv.cpp

Defines the functions declared in *conv.hpp*.

msg.cpp

Defines the functions declared in *msg.hpp*.

wrapper.cpp

Defines the functions declared in *wrapper.hpp*.

mach

The main project has been built in this directory. It contains the following files: *mach_central_server.cpp*, *mach_msg.hpp*, *mach_port.hpp*, *mach_process.hpp*, and *mach_service.hpp*.

mach_central_server.cpp

Executes the central server. The server is first created and then binded to an address. It then listens to any connections. When a connection is detected, it is added to a list of connected clients. Additionally, the server also services any received requests such as a client informing the server of its creation or termination, request to terminate the server, as well as request for list of all connected peers.

mach_msg.hpp

Defines *mach_msg* structure used to represent messages. Each *mach_msg* has a unique ID, a size, and a pointer to data.

mach_port.hpp

Defines *mach_port* structure used to represent ports. Each *mach_port* comprises the port itself, the socket to which it is connected, and its IP address. Additionally, this file defines the *mach_port_bind()* function used to bind a port to a socket.

mach_process.hpp

Defines the *mach_process* class, used to represent processes, with all of its attributes and methods. Each process has a key used to uniquely identify it. Each process also has a send port and a receive port for sending and receiving messages respectively. Associated with each process is a list of clients connected to it and a message queue. Lastly, there is a mutex lock (for synchronization purposes) and a receiving thread responsible for connecting to any processes that wish to, and for receiving messages and placing them in the message queue.

The individual methods have been described in detail here.

mach_service.hpp

Defines *mach_service_type*, an enumerated type used to identify the type of service a client has requested from the server. Its members are *init*, *peers*, *exit*, and *closeserver*. *init* and *exit* are used by clients to inform the server of their creation and termination respectively. *peers* is used when a client wants a list of all connected peers. Lastly, *closeserver* is used to terminate the central server, thereby ending the program.

Moreover, it also defines *mach_service*, a structure to represent a service requested by a client from the server. Each *mach_service* has a type (discussed above) as well as a key (indicates the process demanding the service) and a port (indicating which port of the process has made the request for the service).

Functions and Methods

In lib

conv.cpp

- **in_addr_t pton(const char *ipv4_addr):** Converts a string representation of IPv4 address into a network byte value. Takes as argument *addr* – the string representing IPv4 address. Returns the converted numeric value on success and -1 and error message on error.
- **const char *ntop(in_addr_t numeric_addr, char *buffer):** Converts a numeric network address into a string IPv4 address. Takes as arguments *numeric_addr* – the numeric address value – and *buffer* – pointer to the

destination string. Returns pointer to the same buffer on success and NULL and error message on error.

msg.cpp

- **void writen(sock_t sock, const void *buffer, size_t size):** Writes *size* number of bytes from *buffer* onto the given socket, *sock*. Returns nothing.
- **int readn(sock_t sock, void *buffer, size_t size):** Reads *size* number of bytes from the given socket, *sock,* onto *buffer*. Returns 0 on success and -1 if the connection is closed by the peer during reading.

wrapper.cpp

- **sock_t Socket():** Creates a new TCP socket descriptor of IPv4 protocol. Returns the socket descriptor on success and -1 on failure.
- **void Connect(sock_t sock, const sockaddr_in *remote_addr, socklen_t addr_size):** Connects the given socket to the peer at the given address. Takes as arguments *sock*, the socket we wish to connect; *remote_addr*, the address of the peer; and *addr_size*, the size of said address. Returns nothing.
- **void Bind(sock_t sock, const sockaddr_in *local_addr, socklen_t addr_size):** Binds the given socket at the given address. Takes as arguments *sock,* the socket to bind; *local_addr,* the address to bind at; and addr_size, the size of said address. Returns nothing.
- **void Listen(sock_t sock, int max_conn):** Prepares the given socket to accept connections. Takes as arguments *sock,* the socket descriptor, and *max_conn,* the maximum number of connections. Returns nothing.
- **sock_t Accept(sock_t sock, sockaddr_in *clientaddr, socklen_t addr_size):** Awaits a connection on a given socket. When a connection arrives, open a new socket to communicate with it. Takes as arguments *sock*, the socket which will accept the connection; *clientaddr,* pointer to address structure that should receive the connected peer's remote address (NULL if not required); and *addr_size,* the size of address structure (0 if not required). Returns connected socket descriptor on success.
- **int select_readable(int maxfdp1, fd_set *readset, time_t timeout_sec):** Instructs the kernel to wait on multiple descriptors. Takes as arguments *maxfdp1,* maximum number of descriptors to be tested; *readset,* pointer to descriptor set to test for reading; and *timeout_sec,* number of seconds to

wait before returning. Returns number of ready descriptors on success and 0 on timeout.

In mach

mach_port.hpp

- **void mach_port_bind(mach_port *port, unsigned port_no = 0):** Binds a port to a specific kernel port. Takes as arguments *port,* the port to bind, and *port_no,* the port number of the kernel. Returns nothing.

mach_process.hpp

The functions described here are methods of the *mach_process* class that has been described above, with the exception of *mach_receiving_thread()* which is a friend function.

- **mach_process(key_t _key):** Constructor of the *mach_process* class. Initializes *key,* the data member, to *_key*, the parameter. Binds the receive and send ports to the newly created process and then notifies the server of its creation via a call to *notify_cserver().* Lastly, it initializes the mutex lock (data member of the *mach_process* class) and the receiving thread.
- **~mach_process():** Destructor of the *mach_process* class. Sends an *exit* request to notify the central server of its termination and closes all of its ports. Lastly, it terminates the receiving thread and destroys the mutex lock.
- **bool connect(key_t remote_key):** Connects the process to the peer whose *remote_key* is passed as parameter. Acquires the list of all available peers and then searches for a peer whose key matches the provided key. If the search fails, an error message is printed and false returned to indicate failure; otherwise, we connect to the process, add it to the list of connected peers, and return true to denote success.
- **bool send(key_t remote_key, mach_msg msg):** Sends message, *msg,* to connected peer with key, *remote_key.* We search the list of connected peers for the peer with matching key. If no such peer is found, an error message is printed and false returned to indicate failure. Otherwise, we send the message to the client and return true to denote success.
- **bool receive(mach_msg *msg_buffer):** Function to receive messages. Reads the messages into the *mach_msg* structure pointed to by *buffer.* If the message queue is empty, there are no messages to read; as such, false is

returned to denote failure. Otherwise, the message at the front of the queue is read into the structure pointed to by *buffer* and promptly removed from the queue. During this procedure, the mutex lock is locked to prevent any new messages from arriving while the queue is being altered. At the end, true is returned to indicate success.

- **void closeCServer():** Sends the central server request to terminate, thus effectively ending the program. Returns nothing.

The methods described until now were all public. Now, we move onto private methods:

- **void notify_cserver():** Notifies the central server of process's creation, thereby allowing the central server to add it to the list of connected clients. Returns nothing.
- **std::vector<mach_service> available_peers():** Returns list of available clients. The process first requests the central server for a list of all peers. The server first sends the number of peers, and then sends information related to each peer in a *mach_service* structure one by one. This information is read by the process in a loop and appended to a vector which is returned by the function once the loop ends.
- **void check_server(int err):** Checks whether the server has closed. If it has, an error message is printed, and the process terminates via a call to *exit().*
- **void *mach_receiving_thread(void *arg):** Friend function of the *mach_process* class. It is executed by the receiving thread. The process itself is passed as argument. The thread connects the process to any clients that wish to connect with it and adds these clients to the list of connected peers. Additionally, it checks all the connected clients constantly for any message. If any client sends a message, it is received and pushed into the message queue. Once again, the mutex lock is locked while appending the message to the queue to avoid any messages from being popped off the queue while it is being altered. When the thread is terminated by the destructor, it closes all the sockets before exiting.

How to run the project locally (for gcc version 9)

Download the project on your machine.

```
git clone git@github.com:AjaxAueleke/machipc.git
cd gitmachipc/
```

Make a source folder for all object files.

```
mkdir src
```

Start the central server.

```
./run.sh mach/mach_central_server
```

Start the sending process.

```
./run.sh send
```

Start the receiving process.

```
./run.sh recv
```

To close the central server:

```
./run.sh process_endserver
```

Group Members

1. Muhammad Moaaz bin Sajjad
    - Email: moaaz88sajjad@gmail.com or k200154@nu.edu.pk
    - Github: @nlzza
2. Saad bin Khalid
    - Email: ayyansaad46@gmail.com or k200161@nu.edu.pk
    - Github: @saad0510
3. Muhammad Ahmed Jamil
    - Email: ahmed.jamil7410@gmail.com or k200388@nu.edu.pk
    - Github: @ajaxaueleke