Inter process communication

IPC = How Process talk to each other in an OS.

when your program is split into multiple process, you need a way to share data & sync actions. That's what IPC solves

Q1&2) [Programs in loc: Python-4-week work / Programmatic/Inter Process communication]
unnamed Pipe (pipe() + fork())

What it is:

*) A one-way communication channel

*) Used Between Parent and child process

Real-world Analogy

*) A tube that can send water in one direction only

use case

Parent sends a message to child or vice versa.

ex:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main()
{
    int pipefd[2];
    char buf[100];

    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
```

```
if (fork() == 0) {
    // child reads
    close(pipefd[1]);   // close write end
    read(pipefd[0], buf, sizeof(buf));
    Printf("Child received %s \n", buf);
} else {
    // parent write
    close(pipefd[0]);   // close read end
    write(pipefd[1], "Hello from parent", 18);
}
return 0;
}
```

---

2) Named Pipe

*) A pipe that survives beyond the life of a process

*) A File-like object on disk.

Use case

* communication Between unrelated processes.

ex: mkfifo mypipe.

Writer program

```
int fd = open("mypipe", O_WRONLY);
write(fd, "Hi", 3);
```

Reader program

```
int fd = open("MYPIPE", O_RDONLY);
read(fd, buf, 100);
```

35-36 Shared memory

[Shmget(), Shmat(), Shmdt(), Shmctl()]

▷ what it is :

*) Fastest IPC : multiple processes access same memory block

*) link multiple processes using same global variable.

Use case : Huge data, fast exchange, example : Camera Buffer

1) Shmget() => get / create a shared memory segment

Create an access to shared memory

Syntax :

int Shmget (key_t key, size_t size, int shmflg);

Params :

key : unique identifier

size : size of memory

shmflg : permissions.

2) Shmat() - Attach shared memory to your process

map the shared memory

Syntax :

void *Shmat (int shmid, const void *shmaddr, int shmflg);

Params

shmid : ID returned by shmget()

shmaddr : usually NULL, OS picks address

shmflg : 0 for read/write.

3) Shmdt() - Detach shared memory

Disconnect the shared memory

int Shmdt (const void *shmaddr);

Returns [0 on success, -1 on error]

4] shmctl() Control operations (Remove, info, Permissions)

Perform Control Permissions

Syntax

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

common cmds:

IPC_RMID : Remove the segment

IPC_STAT : Get info about segment

IPC_SET : Change permissions.

EXAMPLE

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main() {
    int shmid;
    char *data;
shmid = shmget(12345, 1024, IPC_CREAT | 0666);
data = (char*) shmat(shmid, NULL, 0);
strcpy(data, "Shared memory works");
Printf("Wrote %s\n", data);
shmdt(data); // Detach
shmctl(shmid, IPC_RMID, NULL); // Delete
return 0;
}
```

37 - 38 Semaphores (Semget(), Semop())

What it is

*) lock / unlock system to protect shared memory access

*) Binary semaphore = like a mutex

use case

Two processes don't write to same Buffer at the same time.

EXAMPLE

```
Semop (semid, &lock, 1)   // lock.
    // critical section.
Semop (semid, &unlock, 1); // unlock
```

9) Pipes vs message Queue vs shared memory

Feature	Pipe	message Queue	Shared memory
Direction	one-way	Two-way	Read/write
Speed	medium	medium	Fastest.
Persistence	unnamed = No	Yes.	Yes.
Complex Data	Hard	easy (structs/msg)	You manage structure.
Sync			
Speed	No	No	Yes (semaphore)

b) Best use cases

use	IPC
Parent - child quick data share	unnamed PIPE
Different apps need to send messages	FIFO (or) MQ
Large dataset (images, logs, etc)	Shared memory
concurrent access to data	shared memory + sem
logging alerts	message queue

System VS Message Queues

1) msgget() -

Syntax: int msgget (key_t key, int msgflg);

*) Key: A unique identifier. use ftok() to generate

*) msgflg: Flags like IPC_CREAT | 0666 (Permissions)

Sample code

```
#include <sys/ipc.h>
#iclued <sys.msg.h>
#include <stdio.h>

int main() {
    key_t key = ftok ("progfile", 65) // generate unique k
    int msgid = msgget (key, IPC_CREAT | 0666); //create que
    if (msgid == -1) {
        Perror ("msgget failed");
        return 1;
    }
    Printf("message queue ID %d \n", msgid);
    return 0;
}
```

2) Send & receive using msgsnd() and msgrcv()

*) message structure:

```
struct msg_buffer {
    long msg_type; // must be > 0
    char msg_text[100];
};
```

*) Sending:

```
msgsnd(msgid, &message, sizeof(message.msg_text), 0);
```

*) Receiving (msgrcv()):

```
msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0);
```

Code

```
struct msg_buffer {
    long msg_type;
    char msg_text[100];
};
int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);
    struct msg_buffer message;
    message.msg_type = 1;
    strcpy(message.msg_text, "Hello from sender");
    msgsnd(msgid, &message, sizeof(message.msg_text), 0);
    printf("Sent %s\n", message.msg_text);
    return 0;
}
```

3) Exchange message between 2 process

+) Run Sender in one terminal

+) Run receiver in another terminal,

Receiver code :-

```
struct msg_buffer {
    long msg_type;
    char msg_text[100];
};
int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);
    struct msg_buffer message;
msgrcv(msgid, &message, sizeof(message.msg_text), 1,
Printf("Received %s\n", message.msg_text);
return 0;
}
```

---

24 Remove message queue using msgctl()

msgctl(msgid, IPC_RMID, NULL)

It will destroys queue put this in either sender or receiver after done.

5) message queue limits

```
ipcs -q          // list all queues.
ipcmk -Q .       // manually create
ipcrm -q <msgid>  // remove
cat /proc/sys/kernel/msgmni   // max queues
```

6) structured messages

You can send anything in msg_text, even structs, as long as size is known

```
struct my_data {
    int id;
    float value;
};
struct msg {
    long msg_type;
    struct my_data data;
};
```

7) Handle IPC_NOWAIT flag

if no message available.

msgrcv (...., IPC_NOWAIT);

Returns -1 with errno ENOMSG if no message is present.

---

8) System V vs POSIX Queues

Feature	System V	POSIX
Naming	key_t + ftok()	Named with strings.
API simplicity	older, less clean	cleaner.
Notification	poll/select not supported	Has Notification support.
use case	legacy systems	Modern linux apps.
Performance	comparable.	comparable

---

9) Blocking behavior in System V

*) msgrcv() blocks if no message is present unless IPC_NOWAIT is used.

*) msgsnd() blocks if queue is full unless IPC_NOWAIT is set.

---

10) Handle permission errors

*) you need 0666 (rw-rw-rw) (or) IPC_CREAT | IPC_EXCL for creation.

*) If you get EACCES, the permissions are not enough.

check with:

```
ipcs -q
```

Then

```
ipcrm -q <msgid>
```

To remove if needed:

Sample program

We will write two programs

1) Sender process -> sends structured messages
2) Receiver process -> receives them, optionally with IPC_NOWAIT, and prints them

Comp loc: [Physics-4-week-work / Program_practice / System Vmessage queue]

If permission give on 0666 it sending and receiving

If we use 0000 it will sending msgrcv failed:

Permission denied:

message queue deleted [in receiver terminal]

In sender terminal if type a message and it shows msgsnd failed: Invalid argument.

POSIX Message Queues

[All new created files in /dev/mqueue/]

What are POSIX message queues?

They allow process to exchange messages in FIFO manner, with support for priorities, non-Blocking and synchronization. Unlike system V, POSIX MQs use file descriptors. Just like Sockets and files.

Concepts to cover:

mq_open() → How to create/open an MQ.

mq_send() → Send messages from one process

mq_receive() → Receive message from another proc

mq_attr → Set max msg count, size, etc.

Blocking receive → How recv blocks until a msg a

Non-Blocking mode → set O_NONBLOCK in mq_op

mq_unlink ⇒ delete queue properly.

Full condition → what happens if queue is full.

Select()/Poll() → monitors queue activity

Producer & consumer → Real-world pattern betwe

two Process.

POSIX message queue functions are implemented in the librt library

---

1) create and open a message queue using mq_open()

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>      // for O_* constants
#include <sys/stat.h>   // for mode constants.

int main() {
    const char *queue_name = "/myqueue"; // must start with /.

// Define message queue attributes.
    struct mq_attr attr;
    attr.mq_flags = 0;      // Blocking mode.
    attr.mq_maxmsg = 10;    // max messages
    attr.mq_msgsize = 256;  // max message size
    attr.mq_curmsgs = 0;    // number of messages currently in queue.

// Open or create message queue.
    mqd_t mq = mq_open(queue_name, O_CREAT | O_RDWR,
                       0666, &attr);

    if (mq == -1) {
        perror("mq_open");
```

```
printf("message queue %s created and opened
        successfully \n", queue_name);

// cleanup : close queue
    mq_close (mq);
    return 0;
}
```

TO Compile

```
gcc -o mq_create_open mq_create_open.c -lrt
```

lrt links the real time library, required for POSIX M

Run

```
./mq_creat_open
```

OP

message queue '/mqueue' creat and opened successfully.

The queue is created we can check it with

```
ls /dev/mqueue
```

msgget()

102) mq_send [Check in lap bo bell pro phyton 4 week / programpractice / posix]

1) mq_send() is used to send a message to a message queue.

2) It requires the queue descriptor, the message buffer, size and priority

3) messages are stored in FIFO order by priority

Code.

```
#include <stdio.h>
:
:
int main(){
    mqd_t mq;
    const char *queue_name = "/demo-queue";
    const char *message = "Hello from mq-send";
    mq = mq_open(queue_name, O_WRONLY);
    if (mq == (mqd_t)-1){
        perror("mq-open");
        exit(1);
    }
if (mq_send(mq, message, strlen(message)+1, 0)== -1)
    perror("mq_send");
    exit(1); }
printf("message sent: %s \n", message);
mq_close(mq);
return 0;
```

3) Receive message using mq_receive()

+) mq_receive() is used to receive a message from an opened message queue.

+) Use mq_getattr() give you the internal queue structure.

*) mq_maxmsg : max messages queue can hold.

*) mq_msgsize : size of each message.

+) you must use mq_msgsize to a create a safe buffer for receiving

---

4) Setting Attributes in POSIX message queue:

struct mq_attr lets you set custom attributes for a queue

```
struct mq_attr {
    long mq_flags; // 0 for Block, O_NONBLOCK
                   for non Blocking
    long mq_maxmsg; // max number of messages
                    in the queue.
    long mq_msgsize; // max size of each
                     message.
    long mq_curmsgs; // number of messages
                     currently in queue.
};
```

Program

```
int main() {
    mqd_t mq;
    struct mq_attr attr;

// Define custom attributes
    attr.mq_flags = 0; //Blocking mode.
    attr.mq_maxmsg = 5; // max messages allowed in queue.
    attr.mq_msgsize = 256; // max size of each message.
    attr.mq_curmsgs = 0; // Not set by the user (used only)

mq = mq_open(QUEUE_NAME, O_CREAT | O_RDWR, 0644, &attr);

if (mq == -1) {
    perror("mq_open");
    exit(EXIT_FAILURE);
}
printf("%d %d %d", attr.mq_maxmsg, attr.mq_msgsize, attr.mq_flags);

mq_close(mq);
return 0;
}
```

Op

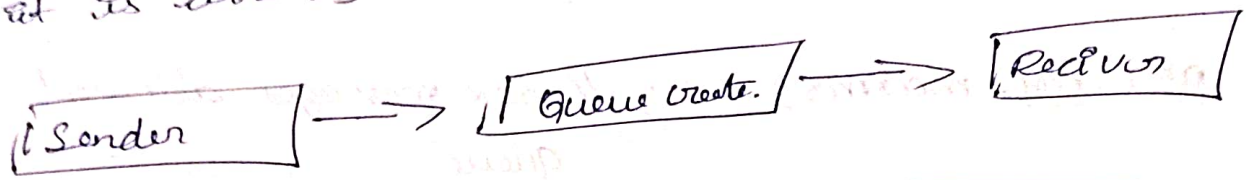
```
mq_maxmsg = 5
msgsize = 256
```

5) Blocking Receive in POSIX MQ

How POSIX Queue works

Shared memory

The queue will be created by the sender if not it is already present



Blocking means the function waits (Block execution) until the condition is satisfied.

For mq_receive:

if no messages in the queue : it waits until a message arrives.

For mq send

if queue is full it waits until space becomes available.

How to check Blocking is working

first run receive - it will block (or) hang

In another terminal run mq_send → now mq_receive will immediately print the message and exit

6) Non-Blocking

if you use the O_NONBLOCK flag while opening, the

*) mq_receive(): returns -1 immediately if no message is available.
(instead of blocking)

*) mq_send(): returns -1 if the queue is full.

Feature	Behaviour in O_NONBLOCK mode
mq_send	Returns EAGAIN if queue full
mq_receive	Returns EAGAIN if queue empty
Benefit	No Blocking or hanging

7) mq_unlink

mq_unlink() is used to permanently remove a message queue from the system.

*) It is like unlink() but for files - it removes the name of the queue

*) It does NOT immediately destroy the queue if it is still open by any process

*) once all process close the queue (mq_close()), the its fully removed from memory.

We can check all ques in ls/dev/mqueue

8) How to handle message queue full conditions ?

when a POSIX message queue is full and a sender tries to send a new message, what happens depends on the flags used with mq_send() :

Blocking mode (O_NONBLOCK not used)

mq_send(mqdes, msg_ptr, msg_len, msg_prio);

if the queue is full, the sender will block (wait) until

a) A receiver consumes at least one message

b) or the process is killed / the queue is deleted

use this if Blocking is acceptable.

Non-blocking Mode (O_NONBLOCK used).

mq_open("/mq", O_WRONLY | O_NONBLOCK);

mq_send(...);

if the queue is full mq_send() immediately fail and returns -1.

1) errno is set to EAGAIN

9) monitor a message Queue using Select() (or) Poll()

To asynchronously monitor a message queue.

+) check when messages arrive without blocking indefinitely

*). combine multiple I/O sources (like sockets, files, mq) in one event loop.

POSIX provides this feature

```
int fd = mq_getfd(mqd_t mqdes);
```

But actually, there's no mq_getfd().

**) use mq_getattr() + mq_notify with SIGEV_SIGNAL (or) SIGEV_Thread

(or)

+) use mqdes in select() / poll() by calling mq_open() and treating it like a file descriptor.

---

10) Producer and consumer

Producer sends message to the queue.

Consumer receives and prints them

File handling

1) C program to open a file and read its content

```
#include <stdio.h>
int main() {
    FILE *fp;
    char ch;
    fp = fopen("sample.txt", "r");
    if (fp == NULL) {
        printf("file not opened");
        return 1;
    }
printf("file content");
while ((ch = fgetc(fp)) != EOF) {
    putchar(ch);   // print char by char
    }
fclose(fp);
return 0;
}
```

2) write to a text file using fwrite function:

[size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream

Program

```
#include <stdio.h>
#include <string.h>
int main() {
    FILE *fp;
    char data[] = "Hello fwrite";
```

```
fp = fopen ("output.txt", "w");
if (fp == NULL) {
    printf ("error opening");
    return 1;
}
size_t written = fwrite (data, sizeof (char), strlen (data), fp);
if (written == strlen (data)) {
    Printf ("Data written successfully");
} else {
    Printf ("Failed to write");
}
fclose (fp);
return 0;
}
```

) use fseek() to jump to the middle of a file and print data.

[int fseek (FILE *stream, long offset, int origin);

origin can be

*) SEEK_SET -> beginning of file

*) SEEK_CUR -> current position.

*) SEEK_END -> end of file

```
#include <stdio.h>
int main() {
    FILE *fp;
    long file_size;
    int ch;
fp = fopen("sample.txt", "r");   // open file.
if (fp == NULL) {
    printf("file not found\n");
    return 1;
}
fseek(fp, 0, SEEK_END); // Move pointer to find file size
file_size = ftell(fp); // get total size in bytes.

// calculate middle
    long middle = file_size / 2;

// seek to the middle
    fseek(fp, middle, SEEK_SET);
printf("content from the middle of the file:\n");
while ((ch = fgetc(fp)) != EOF) {
    putchar(ch);
}
fclose(fp);
return 0;
}
```

4) check if file exist before opening.

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp= fopen("checkme.txt", "r");
  if (fp==NULL){
     printf("File not exist");
  } else {
     printf("File exist");
     fclose(fp);
  }
  return 0;
}
```

(01) use Access() [unix only]

```
#include <stdio.h>
#include <unistd.h>
int main(){
    if (access("checkme.txt", F_OK)==0){
        printf("File exist");
    }
    else{
        printf("File does not exist");
    }
    return 0;
}
```

5) Count number of lines in a given file

```
#include <stdio.h>
int main() {
    FILE *fp;
    char ch;
    int line_count = 0;
    fp = fopen("sample.txt", "r");
    if (fp == NULL) {
        printf("unable to open");
        return 1;
    }
    while ((ch = fgetc(fp)) != EOF) {
        if (ch == '\n') {
            line_count++;
        }
    }
```

// if file isn't empty, add 1 to count if last line (if it doesn't end with \n)

```
    fseek(fp, -1, SEEK_END);
    if (fgetc(fp) != '\n') {
        line_count++;
    }
    fclose(fp);
    printf("Total number of lines: %d\n", line_count);
    return 0;
}
```

b) copy contents of one file to another.

```
#include <stdio.h>
int main () {
    FILE *src , *dest;
    char ch;
// open source file
src = fopen("source.txt", "r");
if (src == NULL){
    Printf ("source file not found \n");
    return 1;
}
while ((ch = fgetc (src) != EOF){
    fputc (ch, dest);
}
Printf (" File copied ");
    fclose (src);
    fclose (dest);
return 0;
}
```

Bonus — Binary files use fread() and fwrite() with Buffer

```
char buffer [1024];
size_t n;
while ((n = fread (buffer, 1, sizeof (buffer), src))>0)
{
    fwrite (buffer, 1, n, dest);
}
```

7) Difference between Text and Binary file Handling in C

8)

Text files

*) modes "r", "w", "a", "r+"

*) Functions fgetc(), fgets(), fprintf(), fscanf().

*) Newline char may be translated (\n to \r\n on windows)

*) slower but easier to debug and read

ex: fprintf(fp, "Name %s \n", name);

Binary

*) modes "rb", "wb", "ab", "rb+"

*) function fread(), fwrite(), fseek(), fclose()

*) No translation data

*) Faster, not-human readable.

Feature	Text file	Binary file
Readability	Human-readable	Not-human readable.
File mode	"r", "w", "a"	"rb", "wb", "ab"
Functions	fprintf, fscanf, fgets	fread, fwrite
Size	can be slightly larger	more compact.
use case	configs, logs, plain data	Images, audio, structs, compiled file
Portability	less portable.	more portable.

8) How to append data to an existing file

```
#include <stdio.h>
int main () {
    FILE *fp;
    char data[] = "This line was appended";

    fp = fopen ("sample.txt", "a");
    if (fp == NULL){
        Printf ("error opening file")"
        return 1;
    }
    fputs (data, fp);
    Printf ("Data appened successfully");
    fclose (fp);
    return 0;
    }
```

) use fscanf() to read formatted input from file

```
#include <stdio.h>
int main () {
    FILE *fp;
    char name[50];
    int age;
    fp = fopen ("data.txt", "r");
    if (fp == NULL) {
        Printf ("could not open\n");
```

```
Printf ("Reading names and ages \n");
while (fscanf (fp, "%s %d", name, &age) == 2) {
Printf ("Name %s, Age %d \n", name, age);
}
fclose (fp);
return 0;
}
```

[why does while (fscanf (fp, "%s %d", name, &age) == 2]

fscanf returns the number of input items successfully read and assigned.

---

10) Write a program to reverse the content of a file

```
#include <stdio.h>
int main() {
    FILE *in_fp, *out_fp;
    long file_size;
    char ch;
in_fp = fopen ("original.txt", "r");
    if (in_fp == NULL) {
        Printf ("unable to open");
        return 1;
    }
out_fp = fopen ("reversed.txt", "w");
    if (out_fp == NULL) {
        Printf ("unable to open");
```

```
    fclose(in_fp);
    return 1;
    }

fseek(in_fp, 0, SEEK_END); //move to end
file_size = ftell(in_fp); // Total bytes

    Read chars from reverse
for(long i = file_size - 1; i >= 0; i--){
                          minus
    fseek(in_fp, i, SEEK_SET); // move to position i
    ch = fgetc(in_fp); // Read one character
    fputc(ch, out_fp); // write to reversed file.
}
printf("file reversed successfully");
    fclose(in_fp);
    fclose(out_fp);
    return 0;
}
```