Assignment 1:

IPC Tasks:
- Create and use named pipes with mkfifo.
- Demonstrate message passing between processes using shared memory.

Using mkfifo

```
t0315000@thales:~/SHREYA/25-02-2025$ ls -l
total 48
prw-r--r-- 1 t0315000 t0315000     0 Feb 25 10:18 my_named_pipe
-rwxr-xr-x 1 t0315000 t0315000 17304 Feb 25 10:21 reader
-rw-r--r-- 1 t0315000 t0315000   648 Feb 25 10:21 reader.cpp
-rwxr-xr-x 1 t0315000 t0315000 17304 Feb 25 10:21 writer
-rw-r--r-- 1 t0315000 t0315000   648 Feb 25 10:19 writer.cpp
t0315000@thales:~/SHREYA/25-02-2025$ g++ reader.cpp -o reader
t0315000@thales:~/SHREYA/25-02-2025$ ./reader
^C
t0315000@thales:~/SHREYA/25-02-2025$ nano reader.cpp
t0315000@thales:~/SHREYA/25-02-2025$ g++ reader.cpp -o reader
t0315000@thales:~/SHREYA/25-02-2025$ ./reader
Opening pipe for reading: ./my_named_pipe
Received message: Hello from the writer program!
```

```
t0315000@thales:~/SHREYA/25-02-2025$ nano writer.cpp
t0315000@thales:~/SHREYA/25-02-2025$ g++ writer.cpp -o writer
t0315000@thales:~/SHREYA/25-02-2025$ ./writer
Opening pipe for writing: ./my_named_pipe
Message written to pipe.
```

- **mkfifo my_named_pipe** creates the named pipe in your directory (~/SHREYA/25-02-2025).
- The **reader** program opens the pipe for reading.
- The **writer** program opens the same pipe for writing.
- Data flows from the writer to the reader through the named pipe

Explanation of Key Functions for shared memory:

- **shm_open()**: Creates or opens a shared memory object. It takes the name of the shared memory object, the flags (O_CREAT to create, O_RDWR for read/write access), and the permissions.
- **ftruncate()**: Sets the size of the shared memory object. We use it to allocate the desired memory size.
- **mmap()**: Maps the shared memory object into the process's address space so that the process can read/write to it.
- **memcpy()**: Copies the message from the producer to the shared memory.
- **munmap()**: Unmaps the shared memory region when the process is done.
- **shm_unlink()**: Removes the shared memory object from the system (cleanup).

2.Message passing using shared memory
We can Clean up by using shm_unlink

```
t0315000@thales:~/SHREYA/25-02-2025$ nano producer.cpp
t0315000@thales:~/SHREYA/25-02-2025$ g++ producer.cpp -o producer -lrt
t0315000@thales:~/SHREYA/25-02-2025$ ./producer
Producer: Message written to shared memory.
t0315000@thales:~/SHREYA/25-02-2025$
```

```
t0315000@thales:~/SHREYA/25-02-2025$ nano consumer.cpp
t0315000@thales:~/SHREYA/25-02-2025$ g++ consumer.cpp -o consumer
t0315000@thales:~/SHREYA/25-02-2025$ ./consumer
Consumer: Message from shared memory: Hello from the producer!
t0315000@thales:~/SHREYA/25-02-2025$
```

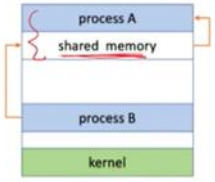
IN-next free position
OUT-first free position

SHARED MEMORY
- Area of the shared memory in address space of process creating shared memory segment
- Other processes must attach it to their address space,both process can use it like regular memory
- Can exchange info by reading,writing the data

Shared Memory (Linux)

int shmget(key, size, flags)

- Creates shared memory segment
- Returns id of segment – **shmid**
- **key**: unique identifier of shared memory segment
- **size**: size of shared memory



int shmat(shmid, addr, flags)

- Attach shared memory (with id **shmid**) to address space of calling process
- **addr**: Pointer to memory address space of calling process

int shmdt(shmid)

- Detach shared memory

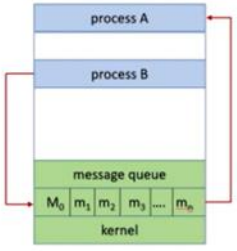
For producer process
While(true)
{
While(((in+1)%buffersize)==out);
Buffer[in]=nextproducer;
in =(in+1)%buffersize;
}

For consumer process
While(true)
{
while(in==out)
Nextconsumer=buffer[out];
Out=(out+1)%buffersize
}

2.MESSAGE PASSING

IPC – Message Passing

- Shared memory created in kernel
- System calls used for communicating:
  - **send** (*A, message*) – send a message to process A
  - **receive**(*B, message*) – receive a message from process B
- Advantage: Explicit sharing, less error prone
- Limitation: slow



A.Naming
-direct communication
-indirect communication
B.Synchronisation
- blocked sender/blocked receiver
-non blocked sender/non blocked receiver
C.Buffereing
- 0
- finite
- Infinite

Communication in Client server systems
Portnumbers:
1.FTP-21
2.Telnet-23
3.HTTP -80

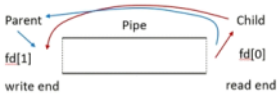
- Sockets
- Remote Procedure Calls(RPC's)
  - client calls client stub or client proxy

- client stub packs parameter into message and marshalling
- client OS sends message to the server using the system call
- server calls server stub or server proxy
-server stub will do demarshalling
- server OS sends message to

- **Java**: Use HttpURLConnection or HttpClient (Java 11+).
- **Go**: The net/http package is the go-to way to make HTTP requests.
- **C++**: Use external libraries like libcurl since C++ doesn't have built-in HTTP support.
- Python: uses request library

- Pipes
- Fd[0] is read end
- Fd[1] is write end

```
#define BUFFER_SIZE 25
int main(void) {
char read_msg[BUFFER_SIZE];
char write_msg[BUFFER_SIZE] = "Greetings";
int fd[2];
pid_t pid;
if (pipe(fd) == -1)                          /* create pipe */
{ fprintf(stderr,"Pipe failed");  return 1; }
pid = fork();                                /* fork child process */
if (pid < 0) {
fprintf(stderr, "Fork Failed"); return 1; }  /* error */
if (pid > 0) {                               /* parent process */
close(fd[0]);                                /* close unused end */
fprintf(fd[1], write_msg, strlen(write_msg)+1);  /* write to pipe */
close(fd[1]); }                              /* close write end */
    else {                                   /* child process */
    close(fd[1]);                            /* close unused end */
    fscanf(fd[0], read_msg, BUFFER_SIZE);    /* read from pipe */
    printf("read %s", read_msg);
    close(fd[0]); }                          /* close read end */
    return 0; }
```



protoc:

The Protocol Buffers compiler, used to compile .proto files into source code in various languages.
To remove : sudo apt remove --purge protobuf-compiler libprotobuf-dev
installations

```
t0315000@thales:~/SHREYA/25-02-2025$ nano main1.cpp
t0315000@thales:~/SHREYA/25-02-2025$ ./person_program
Deserialized Person:
Name: John Doe
ID: 1234
Email: johndoe@example.com
t0315000@thales:~/SHREYA/25-02-2025$ xxd person_data.bin
00000000: 0a08 4a6f 686e 2044 6f65 10d2 091a 136a  ..John Doe.....j
00000010: 6f68 6e64 6f65 4065 7861 6d70 6c65 2e63  ohndoe@example.c
00000020: 6f6d                                     om
t0315000@thales:~/SHREYA/25-02-2025$
```

nano person.proto has all these in it

```
syntax = "proto3";

message Person {
 string name = 1;
 int32 id = 2;
 string email = 3;
}
```

-protoc --cpp_out=. person.proto

The protoc compiler generates two C++ files:
person.pb.h (header file)
person.pb.cc (source file)

sudo apt install pkg-config
sudo apt install pkgconf
g++ main.cpp person.pb.cc -o person_app -lprotobuf

- main.cpp: Your main program.
- person.pb.cc: The generated implementation file.
- pkg-config --cflags --libs protobuf: This ensures that the protobuf compiler and libraries are linked.