./

Learning Report – Linux and OS Programming



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver.Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **To be approved By** | **Remarks/Revision Details** |
| 1 | 05/03/21 | 99003662 |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

Table of Contents

1. Introduction………………………………….…………………………………4
2. C Program Build Process…………………………………………………….5
3. Build using GCC……………………………………………………....……….5
4. Static Library Linking……………………………………………..………….5

5. Dynamic Library Linking…………………………………………..………...8

9. System Call………………………………………………………………..…....7

10. Linux OS Architecture…………………………………………………..…..7

11. Interrupts…………………………………………………………………...….9

12. System call………………………………………………………………….....9

13. Process Table and control block…………………………………..……...7

14. Signal overview………………………………………………………….…..12

15. Threads………………………………………………………………………...14

17. Mutex…………………………………………………………………………...15

18. Semaphore…………………………………………………………...……....16

19. Pipes………………………………………………………………….………...16

20. Shared Memory…………………………….……………………….……….17

21.Message queues……………………………………………………………...17

22. Activites and Assignment………………………………………..……….18

**Introduction :-**

On this report we learn Linux devlopment Tools like how to code that interfaces directly with the kernel and core system libraries, including the shell, text editor, compiler, debugger, core utilities, and system daemons. The majority of both Unix and Linux code is still written at the system level, and Linux System Programming focuses on everything above the kernel, where applications such as Apache, bash, cp, vim, Emacs, gcc, gdb, glibc, ls, mv, and X exist.

An overview of Linux, the kernel, the C library, and the C compiler

• Reading from and writing to files, along with other basic file I/O operations, including how the Linux kernel implements and manages file I/O

• Buffer size management, including the Standard I/O library

• Advanced I/O interfaces, memory mappings, and optimization techniques

• The family of system calls for basic process management

• Advanced process management, including real-time processes

• File and directories-creating, moving, copying, deleting, and managing them

• Memory management -- interfaces for allocating memory, managing the memory you have, and optimizing your memory access

• Signals and their role on a Unix system, plus basic and advanced signal interfaces

• Time, sleeping, and clock management, starting with the basics and continuing through POSIX clocks and high resolution timers

**C Program Build Process:**

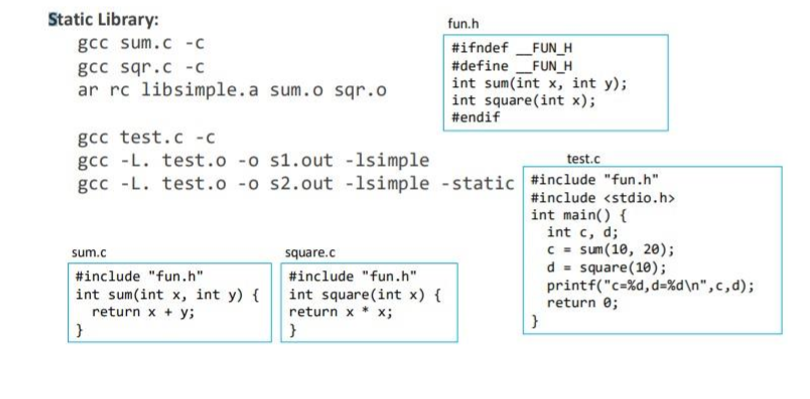
1) Pre-processor : gcc -E filename.c cpp hello.c -o hello.i

2) Compilation: gcc -S filename.c gcc -S hello.i

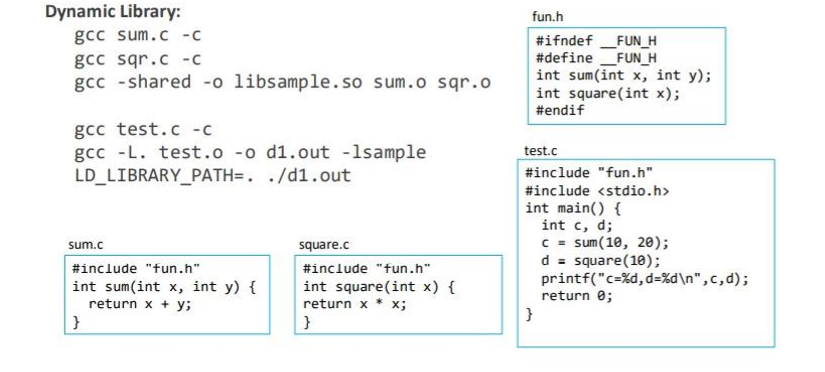
3) Assembler: gcc -c filename.c as -o hello.o hello.s

4) Linker: gcc filename.c ld -o hello.out hello.o ...libraries...

**Static Library Linking :-**



**Dynamic Library Linking :-**

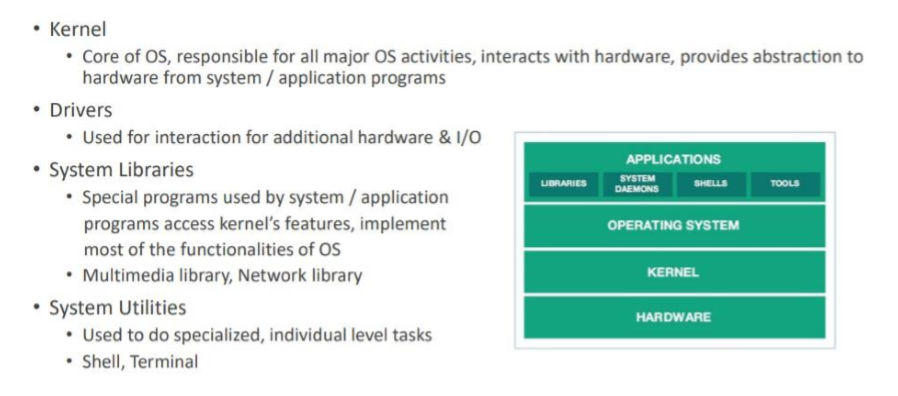


**System Calls:**

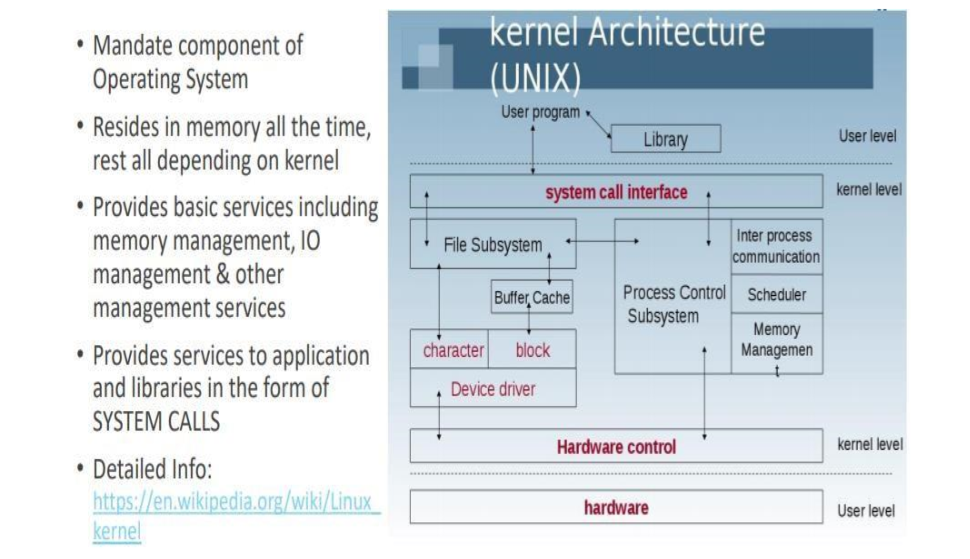
System programming starts and ends with system calls. System calls (often shortened to syscalls) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system. System calls range from the familiar, such as read() and write(), to the exotic, such as get\_thread\_area() and set\_tid\_address().

Linux implements far fewer system calls than most other operating system kernels. For example, a count of the x86-64 architecture’s system calls comes in at around 300, compared with the suspected thousands of system calls on Microsoft Windows. In the Linux kernel, each machine architecture (such as Alpha, x86-64, or PowerPC) can augment the standard system calls with its own. Consequently, the system calls available on one architecture may differ from those available on another. Nonetheless, a very large subset of system calls—more than 90 percent—is implemented by all architectures. It is this shared subset, these common interfaces, that we cover in this book.

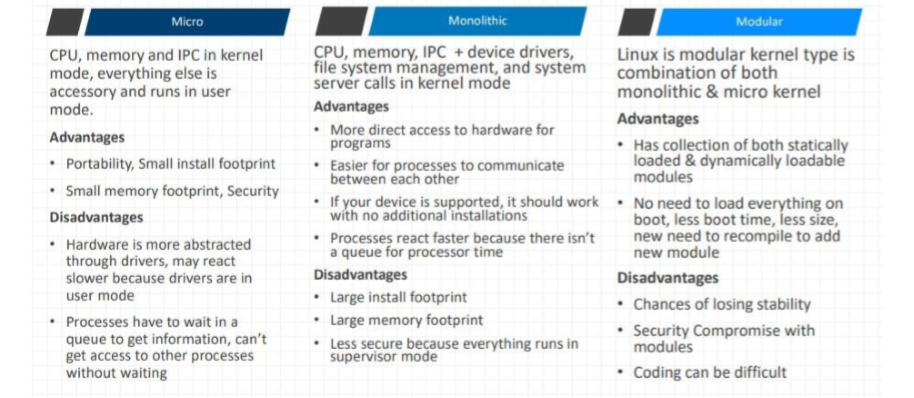
**Linux OS Architecture:**

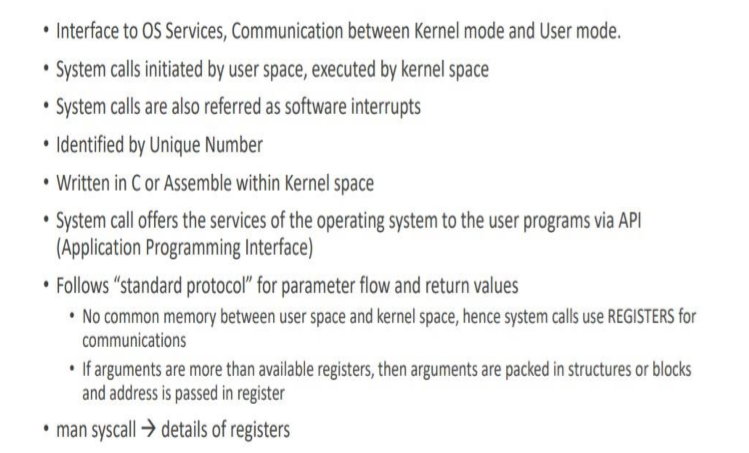


**Kernel :**

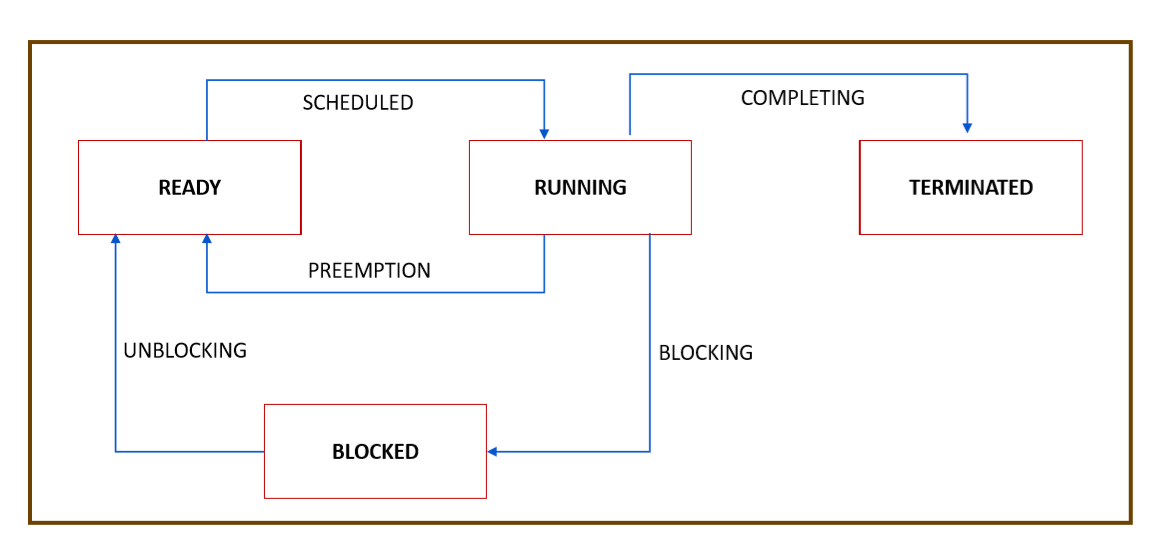


**Types of Kernel :**

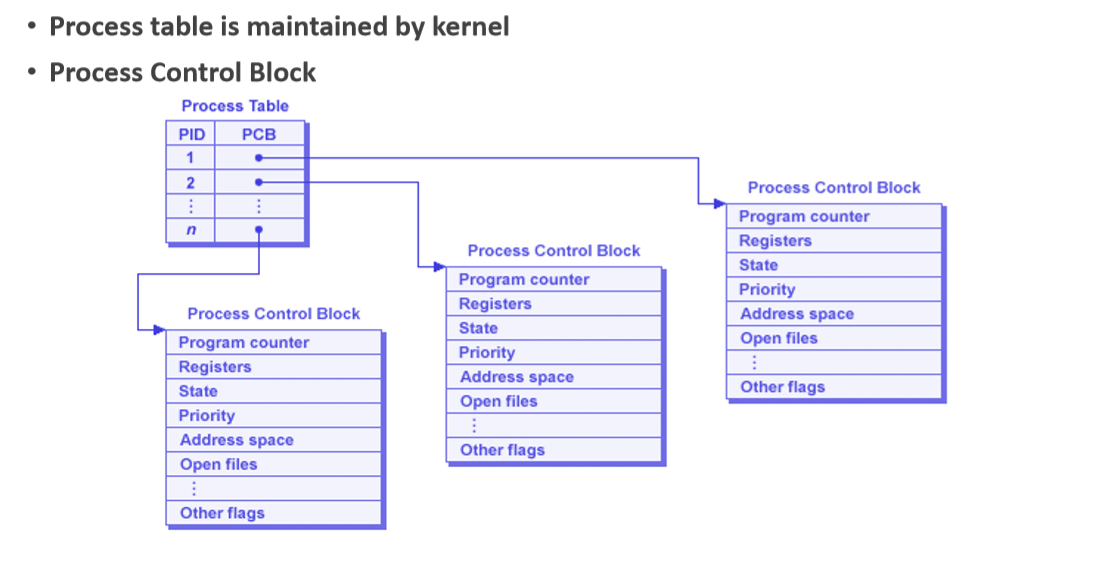


**Interrupts :-**

**Process Life Cycle:**



**Process Table and Process Control Block :**



**New Process Creation:**

->fork ()\_;

• Creates a new process known as child process

• New pid, process control block (PCB) / process descriptor (PD) will be allocated to child (new entry in process table)

• Duplicates resources from parent to child

• fork returns zero to child, non zero to parent

• Child resumes from next statement after fork

• Parent & child run concurrently based on architecture.

**Process Termination:**

• exit() function causes normal process termination and the value of status is returned to the parent

• Process normal termination can be

• success – exit (0)

• Failure – exit with positive value

• abnormal termination

• With exceptions.

**Waitpid:**

• Blocks parent process till completion of child process

• Collect exit status of child

• Cleans some pending resources of child (else child will become Zombie)

• waitpid paramaters

• 1st param : pid of child process waiting for, -1 means any one child

• 2nd param : status of terminated child (pass by address)

• 3rd param : flags

• man waitpid

**Execl :**

• Overwrites child address space with resources of specified program

• Process remains same, but program/resources will change

• Any code after execl is redundant, if execl succeeds

• Syntax

• execl(const char \*path, const char \*arg, ..., NULL); For ex

• excel(“/usr/bin/cal”, “cal”, “2018”, NULL)

• Excel uses absolute path, excelp uses cmd name

**Signals overview:**

• Signals always operate at process level

• Signals communicate between applications at user level

• Used for communication of abnormal termination, illegal memory access & events that go wrong

• Signals are considered as software interrupts, but there is no interrupt vector table

• Signals between processes

• SENDER send / triggers signals from one process to other process

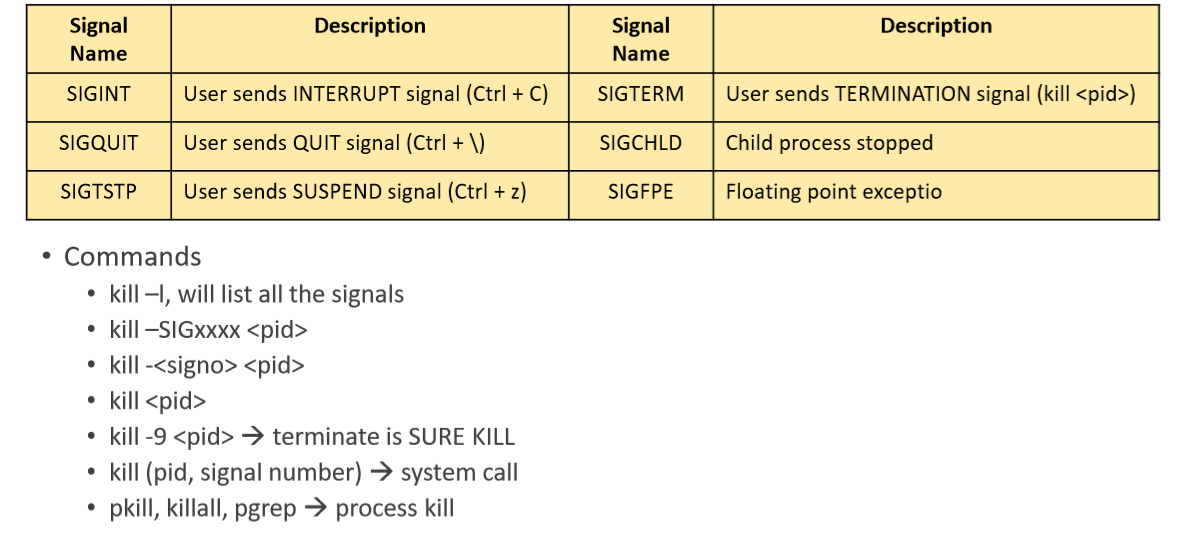
• TARGET will set the corresponding bit based on sender’s signal bit

• Target will lookup in the signal handler table for handler addresses for each of signal handler

• Process descriptor (PD) / process control block (PCB) has signal related fields

• Most of the default signal handlers will cause abnormal termination.

**Signals in common actions:**



**Basics of Threads:**

**Advantage of Thread over Process**

• Concurrent execution and faster response, less time for context switch • Effective use of multiprocessor system

• Resource sharing: code, global data, files can be shared among threads

• PC, Stack and Registers is separate for each thread

• Private / local data is not shared

• Easier communication between threads

• Enhanced throughput of the system

• Number of jobs completed per unit time Note: If one thread makes a blocking call, whole process gets blocked.

**Commands:**

• ps –e –L –o pid,ppid,lwp,nlwp,stat,cmd

• ps –eLf

• To create threads, POSIX thread library is used

• pthread\_create

• pthread\_join

• pthread\_self

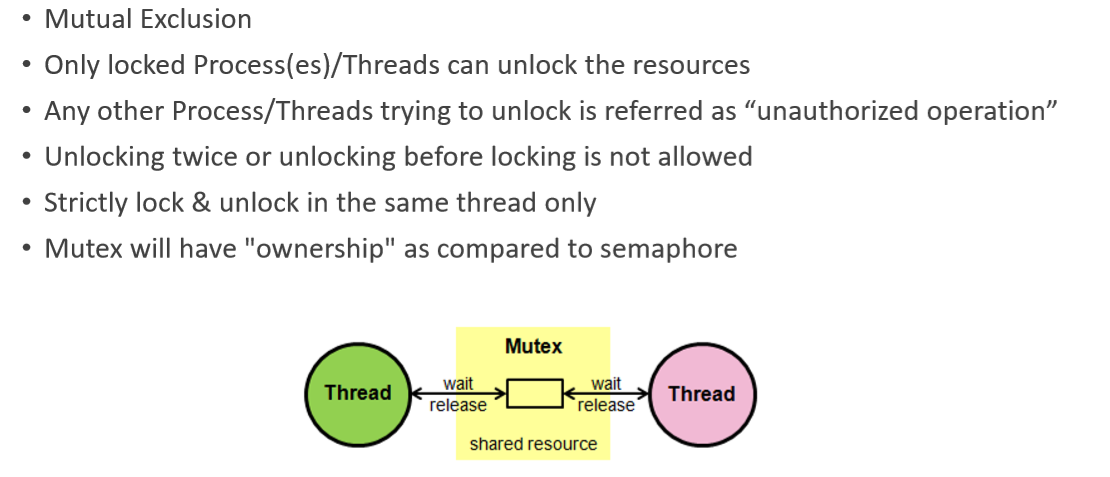
• pthread\_equal

• pthread\_yield

• pthread\_cancel

• gcc psample.c -lpthread

**MUTEX:**



**Mutex API’s**

• #include <pthread.h>

• pthread\_mutex\_t m1=PTHREAD\_MUTEX\_INITIALIZER (declare & initialize)

• pthread\_mutex\_init(&m1)

• pthread\_mutex\_lock(&m1) (lock)

• pthread\_mutex\_unlock(&m1) (unlock)

• pthread\_mutex\_destroy (&m1) (destroy)

Always check return value for Success or Failure.

**SEMAPHORES:**

• Sequencing, Signaling mechanism, used for process/thread synchronization

• Manage and protect access to shared resources

• Kernel level data structure Types of usage

• Binary Semaphore • Value of semaphore ranges between 0 & 1

• Mutual Exclusion / Access to a single resource

• Counting Semaphore

• Value of semaphore can be 0 (zero) & any positive value

• Accessing/sharing multiple similar resources Two (2) varieties of semaphores

• Traditional System V semaphores

• POSIX semaphores.

Two (2) types of POSIX semaphores

• Named

• Unnamed

**Pipes:**

System Calls related to pipe #include <unistd.h>

• int pipe(int pipedes[2]) (Create unnamed pipe)

• ssize\_t write(int fd, void \*buf, size\_t count) (Write to pipe)

• ssize\_t read(int fd, void \*buf, size\_t count) (Read from pipe)

• int close(int fd) (Close pipe)

**Shared Memory:**

• int shm\_open (const char \*name, int oflag, mode\_t mode); Create, or gain access to, a shared memory object.

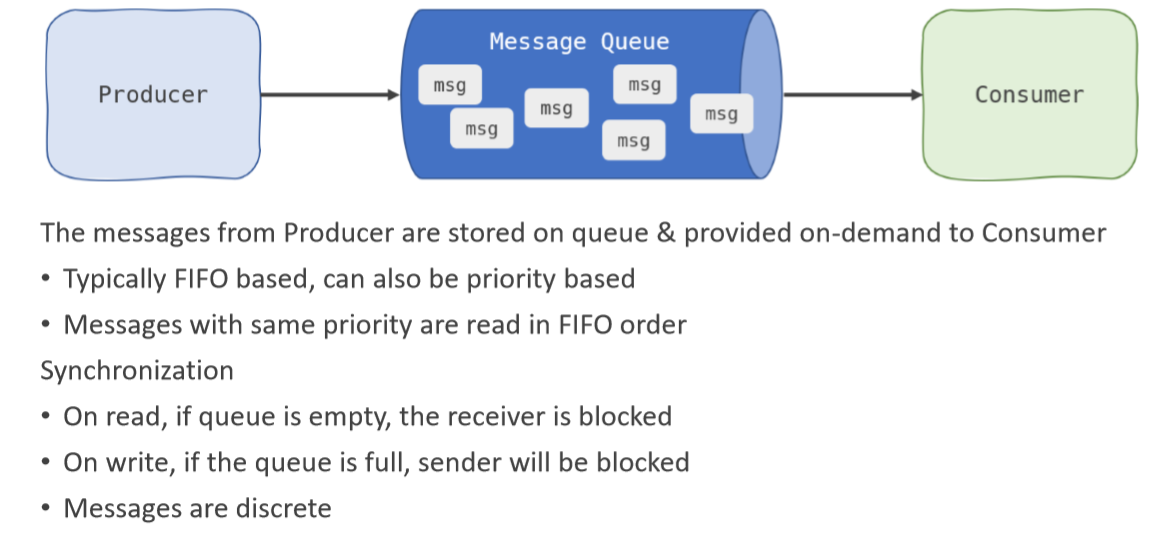
• void \*mmap (void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset); Map a shared memory object into its address space.

Do operations on shared memory (read, write, update).

• int munmap (void \*addr, size\_t length); Delete mappings of the shared memory object.

• int shm\_unlink (const char \*name); Destroy a shared memory object when no references to it remain open.

**Message Queues:**



#include <fcntl.h> /\* For O\_\* constants \*/

#include <sys/stat.h> /\* For mode constants \*/

#include <mqueue.h>

• mqd\_t mq\_open(const char \*name, int oflag)

• mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode, struct mq\_attr \*attr)

• int mq\_send(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned int msg\_prio)

• ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned int \*msg\_prio)

• int mq\_close(mqd\_t mqdes)

• int mq\_unlink(const char \*name)

**Linux Assignments:**

**Static and Dynamic library creation:**

* Repository link:

https://github.com/99003662/Activity\_Linux

* Learning outcome:

Created 12 .c files and 3 .h files make file and a test file.

Finally we created .a, .so. and .out files

**OS Programming Assignments:**

* Repository link:

https://github.com/99003662/Activity\_Linux/tree/main/Activity\_3

* Learning outcomes:

Usage of system calls, process, threads, semaphores, mutex, shared memory, and message queue.