**Linux Fundamentals**

Linux is a free Unix-type operating system originally created by Linus Torvalds with the assistance of developers around the world.

**A picture containing text, electronics

Description automatically generated**

**Diagram, shape

Description automatically generated**

The Unix file system looks like an inverted tree structure. You start with the root directory, denoted by /, at the top and work down through sub-directories underneath it.

**LINUX Bash COMMANDS**

**Table

Description automatically generated**

**Network Management:**

**Table

Description automatically generated**

**Process Management:**

**Table

Description automatically generated**

**Utility**

**Table

Description automatically generated**

**Graphical user interface

Description automatically generated with medium confidence**

**GNUBinary Utilities**

1) Objdump :

a) display segments of your source code

b) Above command will display segment bss info

2) readelf

$ objdump -D &lt;executable file&gt;

Bin -&gt; hex file

Executable -&gt; assembly

$ objdump -j .bss -D main1

Above command will display segment bss info

.text

.data

Can use this for getting info of different segments

readelf -s &lt;executable file&gt;

PLT: procedure link up table

**Basic Input/output file systems**

File Descriptor: non negative integer number

FD &gt; 0

OS assigns a FD when we open a file using open system call

FD can be max 4

i.e. we can open 4 files at a time without closing previously opened file

#include&lt;fcntl.h&gt;

This header file contains flags

* **Open**

open(const char \*pathname, int flags, mode[permission])

On success: Return file descriptor for the given file

On fail: Return -1

* **Flags:**

O\_RDONLY

O\_WRONLY

O\_RDWR

O\_CREAT : this is used with or operation with other three flag operations

* **Read**

read(int fd, char \*buffer, ssize\_t length);

On success: Return number of bytes successfully read from a file

On fail: Return -1

printf(“%s”,buffer);(To print data read in buffer)

Current file offset

Is a measure or count that measures number of bytes from the beginning of the file

* **Write**

write(int fd, char \*buffer, ssize\_t nbytes);

On success: Return number of bytes successfully written to a file

On fail: Return -1

* Lseek

To reposition current file offset:

Lseek system call performs the reposition of current file offset position.

lseek(int fd, offset\_t value, int whence\_arg)

Whence\_arg:

SEEK\_SET : Begin+value

SEEK\_CUR

SEEK\_END

**Type of blocks**

1) Boot Block: booting sequence data

2) Super Block : Statistics of all file systems info and directory info

3) Inode Block: core of the file, contains all information except data

4) Data block: contents of the file

Inode Contains information of :

* Name
* Type
* Size
* Permission
* File operation structure
* Pointer to data blocks
* Struct file operation : for storage device

Linux allows file sharing among multiple processes.So, when we access the same file.txt from different processes, a new fd table is created and a new file structure is created but uses the same inode structure for both the processes.

Struct stat

{

St\_ permission

St\_ size;

St\_ time;

St\_inode;

St\_gid;

st\_userid;

}

**Kernel maintains 3 different tables**

This is for performing operations such as read, write

1) File Descriptor file

2) File structure

3) Inode table

Each process is allocated with a unique identifier. When a process is killed, it’s pid is reused after a delay. It’s not assigned immediately to the new process to ignore the confusion.

Pid == 0 : core kernel process

Swapper process

pid=1 : Init process, this is a user process and will act as a parent process for all the processes running on the system.

The init process is in /sbin folder

getpid()

Returns the pid of the current program running if called from a program

getppid()

Returns parent pid

Ctrl+C :

generates hardware interrupt and it’s notified by the kernel. Kernel immediately sends a termination signal to the process. On reception of the signal the calling process has to terminate.

**Process Execution states**

1) New

2) Ready

3) Execution/Running (CPU)

4) Wait 4) Terminate

5) Halt

**Process Queue**

1) Ready Queue: Linked list of processes that are ready to load and execute

2) Wait Queue: Linked list of processes that are been suspended(waiting for some event)by system to wait

* Scheduler job is to scan the ready queue and choose the best process to execute.
* And the process is chosen in terms of priority, which is the ranking system provided to your processes.
* System is dual core (2 CPU) : will have 2 ready queues
* System is octa core: will have 8 ready queues

⇒ number of cpu = number of ready queues

Kernel launch to space:

User space PAS: process address space

**Contains different segments:**

1. Data
2. Text
3. Heap
4. BSS
5. Stack

**Kernel space PCB: Process Control Block**

Contains:

* pid
* Pointer to process queues
* Maintain CPU registers
* Program Counter Register
* Stack pointer registers
* Instruction registers
* General registers
* Scheduling and thread library

For any process any user space in PAS, there is a process is PCB Process Creation

#include&lt;unistd.h&gt;

pid\_t fork(void)

pid\_t : int data type a kernel data type

* Linux system provides a API called fork to launch a new process from the current task
* Fork is used widely in concurrent(multitasking operations) applications.
* Shell is also using a fork to launch a new application.

Platform: H/W Architecture + OS

**Fork**

After fork() system call, the scheduler decides which process to get executed first either the parent or the child.

**Thread**

* Learning about threads(Light Weight Process)
* Processes are called heavyweight processes bcoz lots of resources are allocated.
* Creating , Maintaining and destroying threads are far cheaper when compared to process.
* Threads are called lightweight processes that need only a few resources.

**Pthread termination**

1) main() return

2) pthread\_exit()

3) pthread\_cancel()

a) Return int data type

b) Used to terminate other threads using thread id : pthread\_cancel(thread\_id)

c) Cancel its own thread : pthread\_cancel(pthread\_self)

**IPC Communication techniques**

1. Pipe
2. Message signal
3. Queue
4. Shared Memory

* Shared memory allows to or more process to access a given region of shared memory.
* Shared Memory is fastest IPC communication tech because need not to jump from one memory location to other memory location for data read and write operation between reader and writer process or a client and server.
* Shared memory can be used between client and server machines only trick is synchronization access in given shared region with semaphores technique.
* (IPCS -m) to list all the shared memory in the system.
* Each shared memory has a structure called as (Shmid) structure.
* Argument of Shared Memory:

Int Shmget (key\_t key, size bytes, int flag)

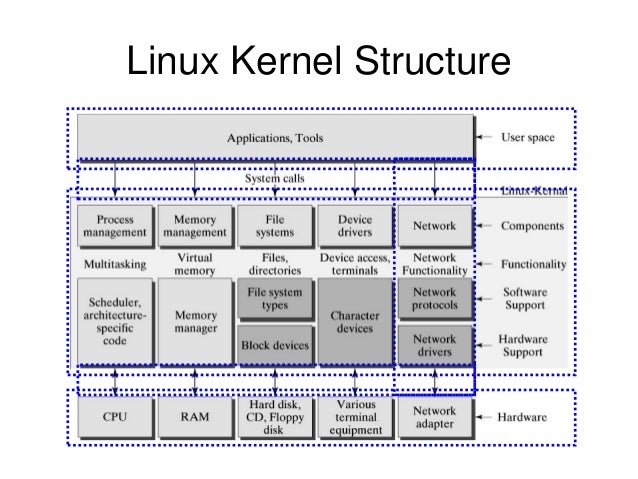
Void \* Shmat (int shmid, void \*add, int flag) on successful execution returns process address space to shared memory.

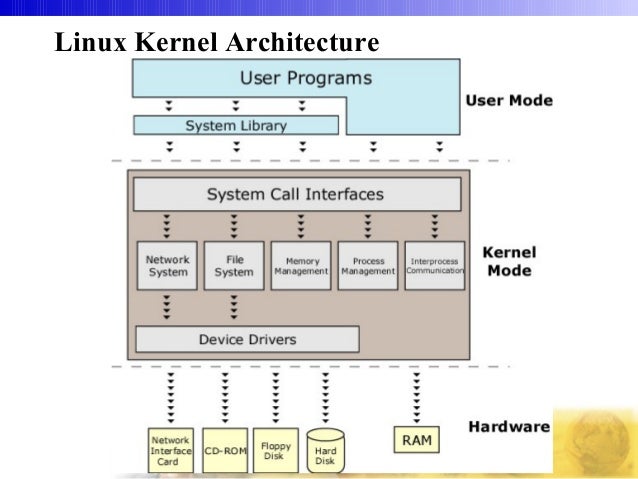
* When a process executes shmid function we take the given shared memory region and mapped at a free process address space (address argument provided a 0).

Shmdt (void \* address) used for Detachment

* To remove a shared memory ” ipcrm -m id” or “ipcrm -M key”
* Shmctl (shmid, int cmd, struct Shmid \*buf)

Linux Kernel architecture





System calls: system calls are kernel space services not user space services, they are meant to invoke kernel services. Each system calls has to pass to system call interface and then invoke appropriate kernel services.

DRIVERS through kernel point of view:

* Device driver is a piece of software that communicates with the hardware manages the hardware and brings the functionality of the end device to the user.
* In Linux operating system device driver resides in kernel space because only the kernel space driver has a privilege to talk to the hardware.
* Driver has a two interfaces one is driver and application interface i.e. OS specific and the other is driver and hardware

**Semaphore in IPC**

* Semaphore function deals with the array of semaphore. Array of semaphore is a bit complex issue but in large application software process need to work on lots of resources need more protection than having a array of semaphore is a big advantage.
* Semget(), Semop(), Semctl(), all are design to work on array of semaphore.
* Int Semget(key\_t key, int no of sem, int semflag)
* On successful execution of semget() function generates semaphore and return SemID.
* Semop system call is used to change the value of Semaphore.
* Int Semop (int SemId, struct sembuff \*ptr, size\_t no of struct are used)

Struct struct

{

Sem\_no

Sem\_op (either -1 (waiting for semaphore) or +1(Semaphore available))

Sem\_flag (Sem\_UNDO)

}

* SemControl is used to control the semaphore operation.
* Int command as a SETVAL, this command is used to initialize the semaphore to well know value or number, the value is required to pass value member of union SEM\_UN only than the semaphore will setup.
* This process is perform when we are using the Semaphore for the first time.
* Semctl (int SemId, int Sem\_no, int Cmd, Union SemUni (optional))

Semflg

Kernel will read sem\_UNDO and will know what changes have been done by the process. The process terminates without releasing the semaphore. Sem\_UNDO will allow automatically release semaphore flag.

Linux operating system signals:

* Signals are the software interrupts they notify process about an event occurred.
* Signals are asynchronous in nature they can happen at any time.
* A Cpu kernel or any software that is running on CPU can trigger signals to the process.
* A process with enough permissions can send a signal to other process.
* A process can send a signal to itself.

**Terminating Generated signals:**

* The signal generated by certain terminal keys (CTRL+C) by name called as SIGINT is delivered by kernel to process.

Linux signals have a naming convention they start with three-character SIG, each signal is defined by a number provided in a header file called signal.h.

Linux Signal is classified into two standards:

1. Std/traditional: this signal is numbered from 1 to 31.
2. Real Time Signal

**Hardware exception Signals:**

This are detected by the hardware and notify the kernel and then kernel will send appropriate message to the process.

1. Invalid Memory Reference
2. Core dumped

**Software Signals:**

|  |  |
| --- | --- |
| **SIGUSR1** | User-defined signal 1. This is one of two signals designated for custom user signal handling. |

A Signal between signal generation and signal delivery is know as pending signal. In response to signal process must do some action,

* so, process can run some default action,
* process can ignore a signal,
* and a process can run a user define function

Instead of running a default action on delivery of a signal a user can program a user define function and register can register with the kernel. On delivery of a signal kernel should invoke a user define function.

**Signal handler**

Must have a <signal.h> file

Void (\* Signal (int signal, void (\*fun) (int)) (int)

(Called as installing or establishing a signal handler with the kernel)

Signal function itself returning a function pointer which takes a single argument

**Signal Subsystem**

* In the kernel there is signal subsystem which is responsible for delivery of signal for each process using user space.
* Scenario: When application 1 wants to send signal to application 2 it must register with signal subsystem and convey the application ID and signal number to which the application 1 wants to send a signal. Signal subsystem on behalf of signal 1 delivery the signal to application 2.
* Case -2: Application X wants to perform periodic task function than register with timer subsystem for delivery of signal at a particular time out.
* (Timer subsystem collects the timer slices) Timer subsystem invokes the signal subsystem after some timeout and signal subsystem will than delivery a signal called SIGALRM.

**(1) Job control Signal**

1. SIGINT 2
2. SIGKILL 9 (CTRL+C)
3. SIGSEGV 11 (Core dump)
4. SIGTERM 15(Process will terminate) (used by the kernel while using termination)
5. SIGSTOP 19 (used by seclude to perform process preemption)

Preemptive Process: In this process kernel forcefully shifting CPU from process without requesting.

Non-Preemptive: until the time slash is done the process will not leave the CPU or else the process should go to the block state.

**(2) SIGUSR1, SIGUSR2, SIGALRM,SIGCHLD**

* SIGKILL and SIGSTOP signal cannot be handler or cannot be caught.

1. **I/O Signals**

I/O signals are delivery by I/O subsystem to the process when file descriptor or sockets are ready to perform input output operations.

**Execution of default handler:**

In signal subsystem there is a function called as send\_signal and this function is executed before delivery of signal.

Steps followed by send\_signal

1. Send signal function will get reference of PCB to which signal is to be delivered.
2. Will point to signal structure referred by PCB.
3. There is a signal pointer to a signal structure that is pointing to a vector of 64 elements.
4. Send signal will manipulate the signal structure. (example for signal INIT manipulating sending element of the vector.(0 is used for Disabled and ! is used for enabled)
5. After setting send signal will raise CPU exception, then CPU stops the current task and CPU executes exceptional handler functions.
6. Now exceptional handler function will access the 2 structures in the PCB that is Signal handler structure.
7. A pointer of signal handler structure will be pointing to a vector of integers.
8. Now exceptional handler will look for signal structure and look for the enabled element and map to respective function in signal handler function.
9. Now the exceptional handler will execute the default function in the interrupt handler.
10. Until this handler is executed the process will not resume.
11. When default function is executed in interrupt context the context switching is not possible.

**Process Blocking Signals:**

* Scenarios: At time process need to block a signals example process dealing with a critical section and the critical section is adapting a database, now during the adaption the process does not want preamp than process cam block a particular signal.
* Application has blocked a signal X, and application Y wants to fire the same signal: Kernel will take the signal X move to the PCB of application X and then places a signal in the seek pending signal structure. And the signal will be released only when the application X unblock the signal.

Sigemptyset (sigset\_t \*set); (initializing with no elements)

Sigfillset (Sigset\_t \*set); (initializing with all elements)

* Sigprocmask system call will add a signal and remove a signal from block state based on sigblock and sigunblock flags.

**Error detect bug**

* A mistake in programming code is error.
* If the error detected by a tester is called as a defect.
* If the defect is accepted by respective developer than it Is called as bug.
* When a system software fails to perform function(execution) leads to failure.
* Fault: a fault is a condition because of the system software fails to perform an action.

**Static code analysis:**

* It is a process of identifying programming errors and bugs in the source code before the program has been run
* It is done is done on asset of instructions by using same coding standards.
* This kind of analysis will help to identify the loopholes and the weakness in the source code that might be harmful.
* The analysis is done on a stationary piece of software that’s why called as Static Code Analysis.
* **Splint** is the static code analyzer tool is used to identify the programming errors and suspicious code and stylistic error.
* **Programming errors:** they are shown and identify by the compiler at the compilation time. i.e., syntax error, usage of undeclared variables, undefine function.
* **Stylistic errors:** if the code is deviating from prescribed coding standards then it is called stylistic code error.

**Responsibility of linker:**

* Linker job is to provide a runtime code to the executable file.
* Th lob of a linker is to add run time code to build executable file
* Run time is not a library it is asset of routines added by linker during programming-built time
* Run time talks about three different macros init,
* The moment the application starts to execute it starts with \_init macros and is also called as initializer.
* Initializer will allocate key resources for the object file and configure the object file. (i.e., providing the address to the object file to load and executes.)
* Once addresses are configured control goes to the \_start macros, it pre return to execute the main function and control jumps to main function execution.
* Here the application functionality executes, when main function terminates the functionality is also terminates than control goes to the start macro, and it calls \_fini macro. (Fini: deallocate all the resources allocate by de init macro.

**Clang:**

* Clang is compiler which used for c and c++ compiling and it is built using C++ and release under Apache 2.2 license.
* Clang is faster and takes less memory on compared to GCC (ex: google chrome browser for Linux and windows in now built using Clang compiler).
* Clang supports few development enrollments.

**Dynamic tools:**

1. GCOV:

* Full form GCC Coverage tool.
* It is an open-source tools.
* It works only with GCC
* GCOV is used to analysis of source code, and it will check for untested part of the source code and identify the unexecuted instructions.
* Can also be used as a profiling tool for browsing and navigating the source code.
* It also us to modify enhance the source code.

-ftest-coverage: analysis the performance of Our file, and records how many times the code is executed

-fprofile-argc: using this file with the source code it generates the profiling output file with .gcda and .gcno.