

---

# Debian

## The Universal OS



**TEAM 4**

Abhishek Jha - 2019UCO1514

Harshit Gupta - 2019UCO1580

# Table of Contents

---

Section	Page Number
• About	2
• Introduction to Debian	3
• System Structure	4-5
• Boot Process	6
• Process and Thread Management	7-13
• Process Scheduling	14-16
• Process Synchronization	17-20
• Memory Management	21-28
• Disk Management	29-33
• File System	34-35
• References	36

# About

---

This project is a CASE STUDY which involves the study of the functionalities and inner workings of an Operating System. The system chosen was a Linux distribution named as **Debian OS**.

**COURSE :** CECSC09 Operating Systems

**INSTRUCTOR:** Mrs. Savita Yadav

The aim of this project is to learn about how the Linux distribution, Debian, implements various Operating System functionalities on top of the Linux Kernel.

The approach to build this project was to first briefly introduce the concept that we are trying to study. After that, elaborate upon how Linux implements the concept with its kernel and then explore how Debian incorporates that functionality in the system, through its packages.

# Introduction to Debian

---

Debian is a very popular open source LINUX distribution that is built on top of the GNU/LINUX kernel, (however a new version of Debian built on top of a microkernel like Mach is also being developed). It was started by Ian Murdock in 1993 and is currently maintained by a team of people over the internet. It is developed and distributed under the GNU free software license and is also the basis for many other distributions like Ubuntu.

Since it is built on top of the LINUX kernel, it is in the UNIX-like family of operating systems and its latest version (Buster) uses the Linux kernel version 4.19. (Note that LINUX is just a kernel and Debian is an entire operating system built on top of the kernel. In fact it is more than an OS as it comes with thousands of application programs as well.)

Debian is developed with an aim to be an OS that is able to run on different types of systems, making it a universal operating system. In line with this aim, Debian has been ported to 9 different hardware architectures and is also being built on top of other kernels like Mach, for servers, and FreeBSD.

# Structure of Debian

All operating systems follow a structure where the different components are organized in relation to each other for the smooth running of the system, hiding the complexities and isolating failures.

In Debian, although the kernel itself is a monolithic binary file that is dynamically linkable with other modules, the entire operating system is made up of different components that provide different services. These components are:

- **The Kernel**

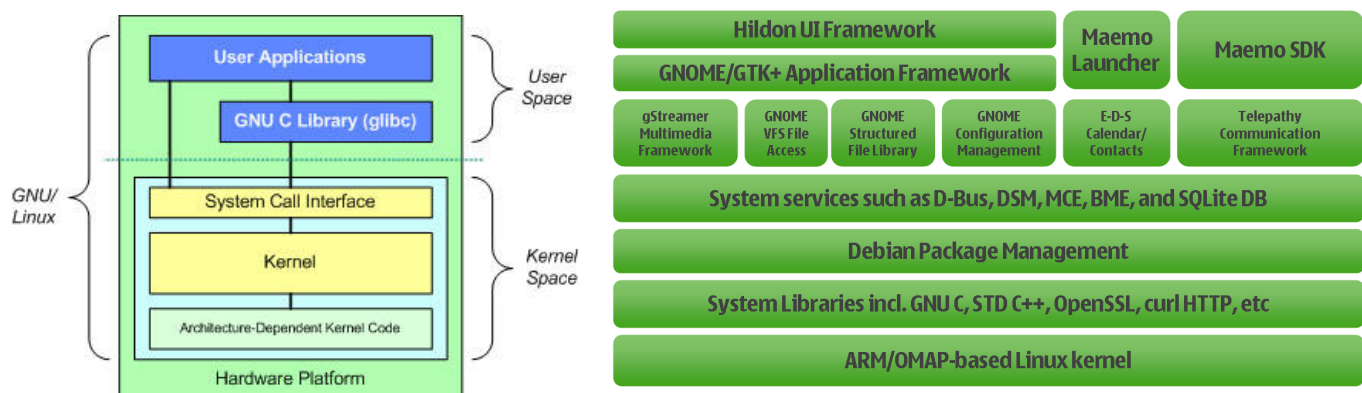
This is the LINUX kernel that is used by Debian.

- **System call interfaces**

This component allows the processes in the user space to ask the kernel to perform privileged operations on behalf of them like servicing an interrupt, reserving memory space, creating a new process, scheduling a hardware resource etc.

- **System Libraries**

These are the essential files and tools that are required by and come with the GNU/LINUX kernel. It includes the GNU compilers, debuggers, editors etc.



- **Package Managers**

Debian uses package managers like APT and dpkg to install new application programs in the system. Since version 3.1, a Debian-Installer is also available that makes installation easier for novice users.

- **System services like daemons**

These are the processes that keep running the entire time the system is running, listening for certain events that are signalled by interrupts like Web Servers and Database Servers.

- **Shell**

Shell is a piece of software that provides an interface for users. Traditionally, it refers to the command line interface of the operating system, but the GUI is also called as the graphical shell. Debian uses Bash as the default interactive shell and Dash as the default non-interactive shell used to run system scripts.

- **Desktop Environment/ Frameworks**

This is the GUI component in Debian that specifies how to display the windows and allows graphical user interactions. This component is optional in Debian and one can choose to install no Desktop Environment and just use the Shell. Neither is there is only one option of a Desktop Environment as the user is given a list of Desktop Environments to choose from and can install multiple of them. Even after the installation the user can use the apt-package manager to install new Desktop Environments.

```
[*] Debian desktop environment
[ ] GNOME
[ ] Xfce
[ ] KDE Plasma
[ ] Cinnamon
[ ] MATE
[ ] LXDE
[ ] LXQt
```

The options provided to a user when installing Debian version 10 or 11

# Boot Process

---

The boot up process of Debian is similar to other operating systems, involving the BIOS and a bootloader but it also allows the user to customize the boot process using parameters that can be passed to the various programs that are executed in the boot process. This is done by modifying certain files that specify the boot process.

The steps in the boot process for Debian are:

- 1) The boot process starts when the CPU executes the BIOS program stored in the ROM on the power-on event. The BIOS performs the POST(Power on Self Test), initializes the hardware, runs the bootloader from the MBR in the specified participation of the primary disk and hands the system control to the next step. The default bootloader of Debian is GRUB.
- 2) The bootloader then loads the system kernel image and the temporary file system image "initrd".
- 3) Then the /init shell script is run which initializes the kernel and transfers control to the next step.
- 4) The root filesystem is switched from the one in the memory to the one in the actual hard disk and the init program is run with a PID of 1. The init program is "systemd" since Debian 8 and provides event driven and parallel initialization of many programs.

# Process and Thread Management

---

**Processes** and **Threads** are at the heart of an Operating System's functionality.

Defining the structures and workings of processes is one of the most important characteristics of an Operating System.

In this section the process structures, process queue structures and their schedulers are discussed. Subsequently, the details of process creation, termination and suspension are discussed. Type of IPC system used in processes and how two processes communicate is discussed further in the following section.

Thread structure and multithreading models used in the Debian OS are discussed further and thread management is present in later sections.

All the above mentioned topics are discussed by first providing a brief overview of the concept, then moving on to the Linux implementation and, if present, Debian functionalities are elaborated.

## A. Process Structure

- A process is a program in execution.
- It is defined by more than program code, that is the *text section*, and consists of a *state* in which it can exist in. Each process is represented by a PCB - *Process Control Block*, in an OS. Some of the common data values which make up a process and thus its PCB are :
  - **Process state**
  - **Process ID**
  - **Program Counter**
  - **Stack**
  - **Data Section**
  - **Heap/Memory Information**
  - **State of Process**
  - **CPU Scheduling Information**



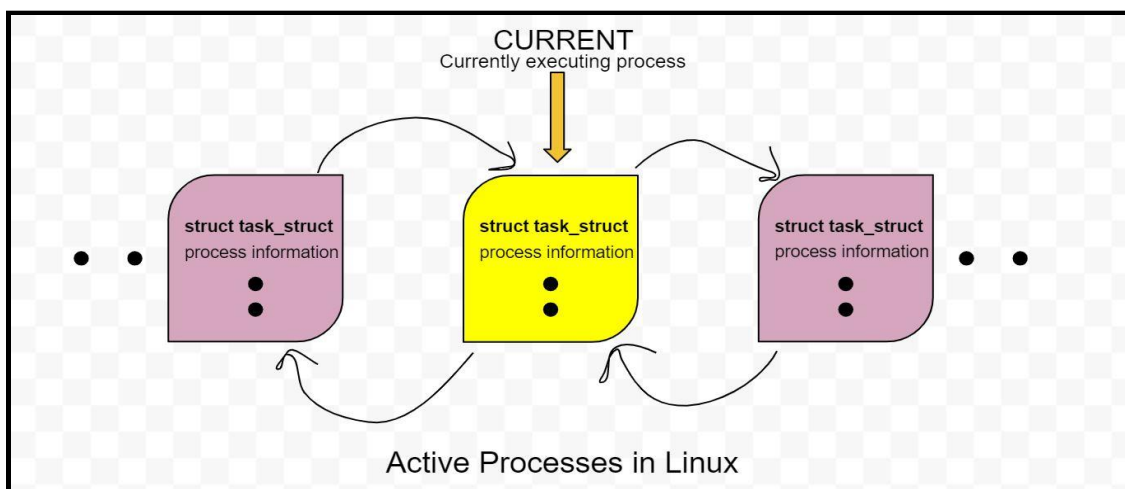
## - Representation in Kernel

The process control block in Linux is represented by the C structure `task_struct`. It contains all the necessary information for representing the process completely in the system. Some of the information that this structure includes is *identifier information, state of the process, scheduling and memory management information, files being referred, pointers to parent and children processes* to name a few. Some of the most important fields are :

1. `pid_t pid;` // process identifier
2. `long state;` // state of the process
3. `unsigned int time_slice;` // scheduling information
4. `struct task_struct *parent;` // pointer to this process's parent
5. `struct list_head children;` // the process's children
6. `struct files_struct;` // the list of open files
7. `struct mm_struct *mm;` // address space of this process

And more...

The data structure used to represent the active processes in Linux is a *doubly linked list* of `task_struct` type and the system kernel maintains a pointer - *current* - to the current executing process on the system.



## B. Process Schedulers

Schedulers are present in an Operating System to manage in what order processes are executed in a computer system. Schedulers are broadly classified as short-term and long-term schedulers. Since schedulers are a part of the kernel, we would be looking at how *Debian Kernel* implements these schedulers at the kernel level and which algorithm is employed by them

- **Short-Term schedulers** are used in determining the order or sequence of processes to be fed into the computer system's main memory and are used to schedule processes that are ready to be executed but waiting currently. Frequency of execution for a short term scheduler is high as processes may encounter interrupts, going into the waiting state and thus would need to be scheduled again.
- **Long term scheduling** is used in determining which jobs are to be brought in the main memory from a job pool on the disk. Frequency of execution is low for a long term scheduler as it may only be invoked when a process leaves the system.

### NOTE

Since Debian is a *desktop* operating system, there is no need for a Long term Scheduler to be present in the OS as there are no jobs with very long run times and thus requiring to be scheduled in the long run.

## C. Process Life-Cycle

Every process executing in a system has a certain *state* in which it can currently be. Each process may be in one of the following states - *new, running, waiting, ready or terminated*. It is important to realize that only one process must be running on any processor at any instant but many processes may be ready and waiting. Again, since the kernel is responsible for the management of the processes in the system, we shall look at how Linux actually manages the life cycle of a process in the system.

## Process Life in Kernel

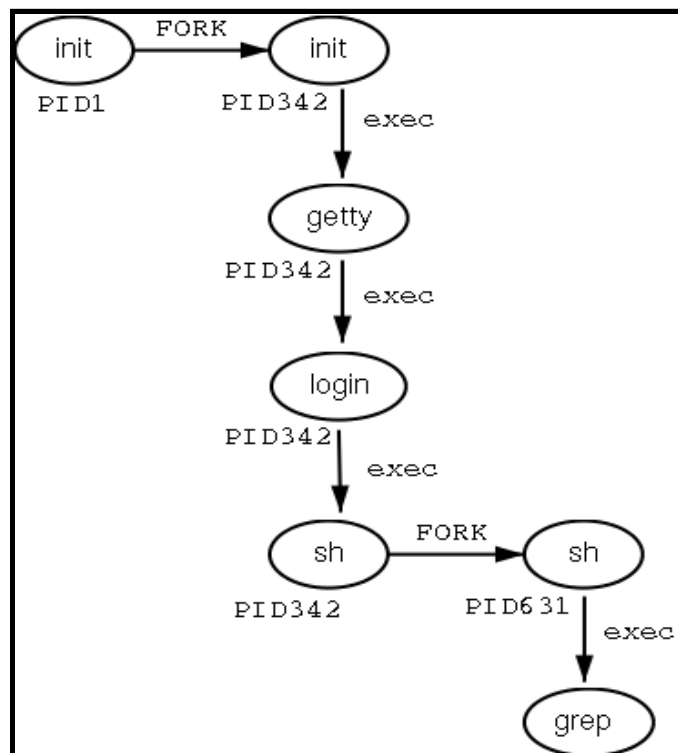
During execution, a process changes from one state to another depending on its environment/circumstances. In Linux, a process has the following possible states:

- **Running** – here it's either running (it is the current process in the system) or it's ready to run (it's waiting to be assigned to one of the CPUs).

- **Waiting** – in this state, a process is waiting for an event to occur or for a system resource. Additionally, the kernel also differentiates between two types of waiting processes; interruptible waiting processes – can be interrupted by signals and uninterruptible waiting processes – are waiting directly on hardware conditions and cannot be interrupted by any event/signal.
- **Stopped** – in this state, a process has been stopped, usually by receiving a signal. For instance, a process that is being debugged.
- **Zombie** – here, a process is dead, it has been halted but it still has an entry in the process table.

## 1. Process Creation

- A new process is created because an existing process makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called *forking* and **fork()** system call is used to initiate the process.
- After the forking process, the address space of the child process is overwritten with the new process data. This is done through an **exec()** call to the system.



Process spawning tree in **Linux**.

The process ID changes after the fork procedure

- There is also a possibility of *daemonizing* a process which just means making processes keep on running in the background. This is particularly useful for processes if they want to spawn a child and then finish their own execution. Processes may *daemonize* their child processes so that they can keep on running even when the parent process stops.

## 2. Process Termination

- Processes end because they receive a **signal**. There are multiple signals that you can send to a process.
- A process can exit using the `_exit` system call, this will free up the resources that process was using for reallocation. So when a process is ready to terminate, it lets the kernel know why it's terminating with something called a termination status.
- The return codes can then be interpreted by the parent, or in scripts. The values of the return codes are program-specific. For example, the `grep` command returns -1 if no matches are found.

## D. Inter Process Communication Systems

Processes can be generated in a system by various methods but using *the terminal* to start a process or using the `fork()` system call within a process are the two most dominant methods. The main mechanisms used by Debian Kernel for IPC are **shared files, shared memory, pipes, message queues, sockets and signals**.

Here we discuss three of the most used methods by the Linux kernel for communication amongst processes.

### 1. Shared Files

- Shared files are the most basic IPC mechanism. Considering the example of a producer-consumer problem, a *producer* may create and write to a file and another *consumer* may read from this same file.
- This approach of data sharing requires **locking** of the files such that one write operation is not conflicting with any other operations. These locks can be *shared* or *exclusive*, depending on the nature of file access.
- The standard I/O library of C includes a utility function called `fcntl` that can be used to inspect and manipulate both exclusive and shared locks on a file
- For file locking, Linux provides the library function `flock`, which is a thin wrapper around `fcntl`.

## 2.Shared Memory

- Any data written, by a process, to a shared memory region can be read immediately by any other process that has mapped to the same region.
- Shared memory does not guarantee synchronization of the processes but when equipped with synchronization, for example through **semaphores**, it becomes a powerful technique for IPC
- Linux systems provide two separate APIs for shared memory: *the legacy System V API* and the *POSIX API*.
- A shared memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as an independent address space.
- This shared-memory object acts as a backing service for *shared memory regions* just as a file acts as a backing store for *memory mapped regions*.

## 3.Pipes

- The standard UNIX **pipe** mechanism allows a child process to inherit a communication channel from its parent.
- Data written from one end of the pipe can be read from at the other end.
- Each pipe has a pair of **wait queues** to synchronize the reader and writer.

## E. Thread Structure

Threads are a way to abstract the *subtasks of a process*. There may be a browser loading which has a thread to make requests to the server, another thread to load the incoming data, another to handle keystrokes of a user typing in the search box and so on.

Threads allow for *concurrent programming* and, on multiple processor systems, *true parallelism*.

### Threads in Linux

To the Linux kernel, there is **no concept of a thread**. Linux implements *all threads as standard processes*.

- For Linux kernel, a thread is merely *a process that shares certain resources with other processes*. Each thread has a unique **task\_struct** and appears to the kernel as a normal process. It is identified differently in the sense that this **task\_struct** happens to share resources such as an address space with other processes

- On systems with explicit thread support, there might exist one process descriptor that in turn *points* to different threads. The process descriptor describes the shared resources, such as an address space or open files. The threads then describe the resources they alone possess.
- Conversely, in Linux, there are simply  $n$  processes for  $n$  threads and thus  $n$  normal **task\_struct** structures. All processes(representing threads) are set up to share certain resources.

## Thread Creation

Similar to the **fork()** system call for creating new processes, threads can be created using the **clone()** system call.

When **clone()** is invoked, it is passed a set of flags, which determine how much sharing is done between the parent and child tasks.

Some of these flags are :

- **CLONE\_FS** - File-system information stored
- **CLONE\_VM** - Same memory space is shared
- **CLONE\_FILES** - Set of open files is shared
- **CLONE\_SIGHAND** - Signal handlers are shared

**clone(CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND , 0 ) ;**

The above code results in a behaviour identical to **fork()**, but just the *address space, open files, filesystem resources, file descriptors and signal handlers* are shared.

These clone flags mentioned above are defined in the **<linux / sched.h>** header of C.

The major difference between **fork()** and **clone()** system calls is that when **fork()** is invoked, a new task is created, along with a *copy* of all the associated data structures of the parent process whereas when **clone()** is executed, rather than copying all the data structures, the new task *points* to the structures of the parent task, depending on the flags mentioned while calling the **clone()**

# Process Scheduling

## Process Scheduling in Debian Kernel

### Completely Fair Process Scheduling

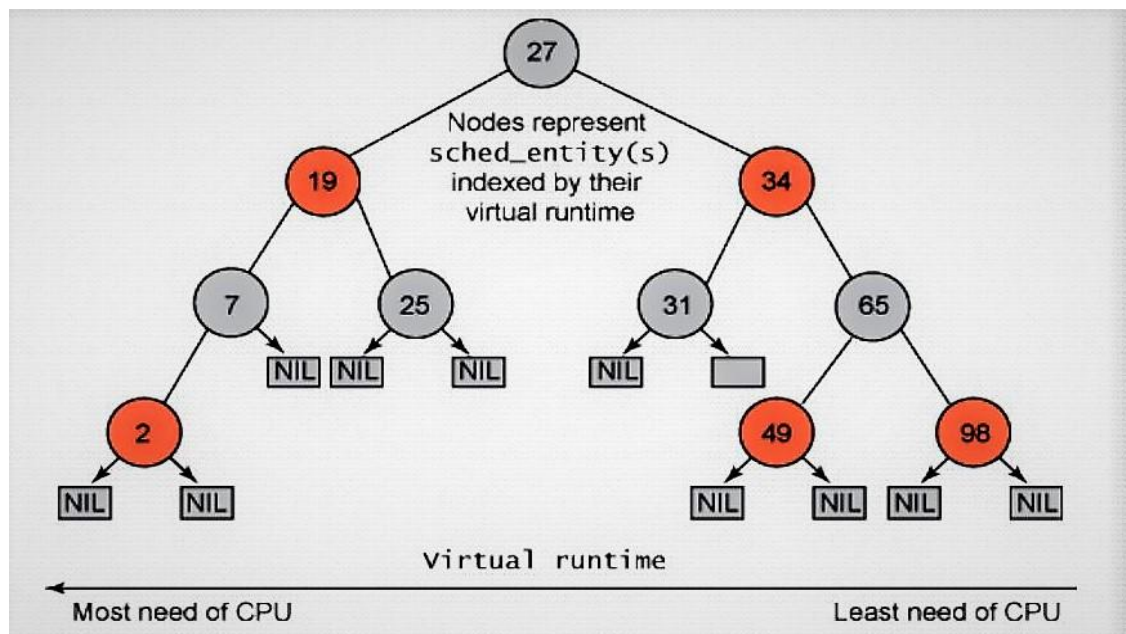
Since 2007, Linux kernel uses CFS algorithm for the scheduling of processes and it has turned process scheduling into thread scheduling by treating a scheduled process as if it were single-threaded. If a process is multi-threaded with  $N$  threads, then  $N$  scheduling actions would be required to cover the threads.

- The CFS model is inspired by the classic preemptive scheduling algorithm. CFS builds upon preemptive scheduling by still using the concept of *time slices and priorities* but they are no longer fixed — *The amount of time for a given task on a processor is computed dynamically as the scheduling context changes over the system's lifetime.*
- CFS scheduler has a **target latency**, which is the minimum amount of time — idealized to an infinitely small duration — required for every runnable task to get at least one turn on the processor. Suppose there are  $N$  tasks contending for a processor then each runnable task then gets a  $1/N$  slice of the target latency.
- Some processes terminate, new ones are spawned and process' states change. Thus, the value of  $N$  is dynamic and so, therefore, is the  $1/N$  time slice computed for each runnable task contending for a processor. Each task is further allocated its time slice according to the priority of the task and accordingly lower-priority tasks would be allocated a smaller share of the  $1/N$  time slice as compared to the higher priority ones.
- CFS also tries to minimize context switches, as time spent on a context switch is time unavailable for other tasks. So, once a task gets the processor, it runs for its entire weighted  $1/N$  slice before being preempted in favor of some other task.
- To determine preemption, CFS keeps track of **virtual runtime (vruntime)** for all tasks and the lower a task's *vruntime*, the more deserving the task is for the processor. So, suppose task T1 has run for its weighted  $1/N$  slice, and runnable task T2 currently has the lowest *virtual runtime* (vruntime) among the tasks contending for the processor. In this case, T1 would be preempted in favor of T2.



# Implementation

- CFS, the scheduling class described so far, is called `SCHED_NORMAL` in the kernel and is a code module written in C.
- The *runqueue*, in Linux kernel, is implemented by a *vruntime* ordered red-black tree data structure. This helps to efficiently insert and remove arbitrary processes from the runqueue in  $O(\log_2 n)$  time.
- The tree's internal nodes represent *tasks or task groups*, and these nodes are indexed by their *vruntime* values so that (in the tree as a whole or in any subtree) the internal nodes to the left have lower *vruntime* values than the ones to the right
  - Tasks with the lowest *vruntime*—and, therefore, the greatest need for a processor—reside somewhere in the left subtree
  - Tasks with relatively high *vruntimes* congregate in the right subtree
  - A task with the smallest *vruntime* winds up in the tree's leftmost node



The *runqueue* of CFS scheduler represented as a **red black tree**

- Since CFS also supports symmetric multiprocessing (SMP), thus every processor has a specific *runqueue* of tasks, and no task occurs at the same time in *more than one* runqueue
- As mentioned earlier, each task is represented as an instance of the **task\_struct** data structure which contains a **sched\_entity** structure. This internal structure contains the



scheduling - specific information, in particular, the *vruntime* of a task , through which the task is ordered in *runqueue*.

- In the red black tree, a **cfs\_rq** structure instance embeds a `rb_root` field named **tasks\_timeline**, which points to the root of a red-black tree. Each of the tree's internal nodes has pointers to the parent and the two child nodes; the leaf nodes have nil as their value.

# Process Synchronization

---

Debian uses the GNU/LINUX kernel at its core and thus the kernel level synchronization task is handled by the LINUX kernel. For synchronization in the user space, POSIX libraries like pthreads provide an API to application programmers which has tools like mutex locks, semaphores, condition variables etc. These libraries are widely used in application programs for thread creation and synchronization.

## Synchronization in Kernel mode

The LINUX kernel of Debian provides multiple ways to achieve process synchronization in the kernel space. Each of them are employed for different purposes under different scenarios and are described below:

### 1) Atomic Integers

This is the simplest form of synchronization tool used by the kernel. It relies on the computer hardware's atomic instructions on certain atomic data types. It is represented as an `atomic_t` in the code and any operation performed on it like `atomic_add(10, &counter)` where `counter` is an `atomic_t` variable is performed as a single machine instruction. This approach, though limited in its use, is suitable when certain integers have to be modified atomically as there is no overhead of a locking mechanism.

### 2) Mutex locks (Binary Semaphores)

The kernel utilizes mutex locks mostly as a solution to the critical section problem. Whenever a portion of code has to be executed in an exclusive manner, the process first acquires a mutex lock by calling `mutex_lock()` and then releasing the lock in the exit section through `mutex_unlock()`. If a lock is unavailable, then the thread or process is put to sleep and it is awakened when `mutex_unlock()` is executed by the process currently holding the lock.

### 3) Spin locks

The kernel uses spin locks to protect the kernel data structures from concurrent access in a multiprocess environment. The code is written in a way to ensure that locks are only held for a short period of time, thus the cost of busy waiting on a CPU core is much less than the cost of context switching. Also, while one core is busy waiting, other cores are still executing so the time spent waiting doesn't lead to a total stop in productive work being done.

#### 4) Disabling kernel preemption and interrupts

However, in a single processor system, busy waiting is unacceptable to any degree as no useful work is done at all during the wait period. Therefore, the kernel process disables pre-emption and interrupts when it is accessing a shared resource and then re-enables them after it is done. This ensures that no other process or thread can modify the kernel data structures when it has access to it.

### Synchronization in User mode

Synchronization for the user threads are performed by application programmers when they write code using the POSIX threads or Pthreads library. This library provides an OS-agnostic and language-independent API that works on most Unix-like operating systems (Some versions also exist for Windows based operating systems like pthreads-w32). This uniform API across different languages greatly simplifies the application development process while still some languages like JAVA provide their own APIs for process synchronization.

Pthreads uses the atomic instructions provided by most architectures to implement mutex locks, semaphores and spin locks. For example it uses the commonly available **xchg** instruction that swaps values of two variables and returns the old value of the first variable. The following loop structure can be used to acquire a lock exclusively.

```
while(xchg(&lock, 1)!=0) {
    //If we do nothing, then it's a spin lock
    //If we sleep, then its a sleep lock or a mutex lock
}
```

The pthreads synchronization tools are described below in more detail:

#### 1) Mutex locks

Pthreads uses the `pthread_mutex_t` data type to represent a mutex lock in code. We first need to create a variable of this type and then pass it to the **pthread\_mutex\_init()** function by reference to initialize the lock (make it available as a global lock visible to all threads). Then we can use the **pthread\_mutex\_lock** method to acquire the lock and **pthread\_mutex\_unlock** to release the lock.

The following code example illustrates the use of a mutex lock;

```
#include <pthread.h>
...
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
pthread_mutex_lock(&lock);
```

```
...    //the critical section
pthread_mutex_unlock(&lock);
...
```

## 2) Named Semaphores

These semaphores are given a name and can thus be referred by the name from the process that created it or by any unrelated process as well. We use the `sem_t` data type to represent the semaphore in code and use the `sem_open()` function to create a named semaphore. We need to pass the following parameters:

- Name of the semaphore
- Creation Flag
- Whether semaphore has read write access for other processes
- Initial value of the semaphore

and it returns a reference to the semaphore.

We then signal and wait on a semaphore using the `sem_wait()` and `sem_post()` functions.

```
#include<semaphore.h>
...
sem_t *sem;

//semaphore is named S. and is only created if a semaphore
//It is only created if a semaphore with that name doesn't exist
//Semaphore has read write access for other processes.
//It's initial value is 1
sem=sem_open("S", 0_CREAT, 0666, 1);
...
sem_wait(sem)
//critical section
sem_post(sem)
...
```

## 3) Unnamed Semaphores

These semaphores do not have a name by which they can be referred, thus usually are only available to the threads of the process that creates the semaphore. We can create an unnamed semaphore using the `sem_init()` function. It takes a reference to a `sem_t` type, the number of processes other than the current that may have access to this semaphore and the initial value of the semaphore.

The following code illustrates how to use an unnamed semaphore:

```
#include <semaphore.h>
...
```

```

sem_t sem;
sem_init(&sem, 0, 1);
...
sem_wait(&sem);
//critical section
sem_post(&sem);
...

```

#### 4) Condition Variables

They are provided in Pthreads as the `pthread_cond_t` type. They can be created by the function `pthread_cond_init()` and waited upon by using the `pthread_cond_wait()` and be signalled by the `pthread_cond_signal()` functions.

Since, C doesn't have monitors, we associate the condition variables with a mutex lock so that only one process can check the condition to determine whether to wait for the condition variable or not. Upon executing `pthread_cond_wait()` the process releases the associated mutex lock so that some other process can use it. We should remember to release the lock after executing `pthread_cond_signal()` so that the signalled process can access the conditional.

# Memory Management

---

Memory management in a computer system is a **kernel level** task associated with the Operating System kernel. The Debian Linux kernel contains a *memory management subsystem* which is responsible for the management of memory in the system.

This includes implementation of virtual memory and demand paging, memory allocation both for kernel internal structures and user space programs, mapping of files into processes address space and many other tasks involving memory usage.

Memory management under the Debian kernel has two components.

- First deals with the allocation and freeing of physical memory — *pages, groups of pages and small blocks of memory*.
- The second one handles the virtual memory, which is the memory mapped into the address space of the running processes.

## A. Physical Memory Management

Linux separates the physical memory into three different **zones**, due to specific hardware characteristics. These zones are architecture specific and are classified as -

- **ZONE\_DMA** - This is the zone where *direct memory access* can take place via the CPU. It is the smallest in size amongst all the other zones and some architectures such as Intel 80x86, devices can only access the first 16MB of physical memory using **ZONE\_DMA**.
- **ZONE\_NORMAL** - It is used for the most routine memory accesses and is used for identifying the physical memory mapped to the CPU address space. If DMA is not limited, the system may altogether skip **ZONE\_DMA** and just use **ZONE\_NORMAL**.
- **ZONE\_HIGHMEM** - This refers to the "high memory zone" available in the system. It is the physical memory which is *not mapped to the kernel space in the system*. For example, on 32-bit Intel architecture ( where  $2^{32}$  provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space. The remaining memory is allocated to the **ZONE\_HIGHMEM** .

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 - 896 MB
ZONE_HIGHMEM	> 896 MB

Relationship of physical memory zones in the Intel 80x86

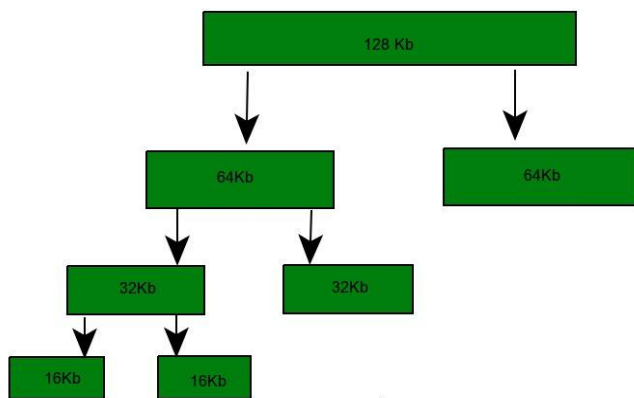
## - Primary memory manager

In Linux, **page allocator** is the primary memory manager.

Each memory zone consists of its own page allocator and it is responsible for allocation and freeing of all physical pages for that zone.

There are 2 main types of systems used for the allocation of physical memory in the Linux kernel. They are the **buddy allocator** and the **slab allocator**.

### • BUDDY ALLOCATOR



Buddy Memory allocation being done for a 16 KB request

1. In this type of allocation, adjacent units of memory are paired together.
2. As seen in the figure, each memory region has a *buddy* region.
3. Whenever two adjacent regions free up, they combine to form a bigger free region.
4. Also, if a small memory request can't be fulfilled by existing regions, larger free regions are split up, until one of them can.
5. Note that memory is always *split up* into 2 until the region which can serve the request with *minimum wasted space* is found.

## • SLAB ALLOCATOR

1. A **slab** is used for allocating memory for kernel data structures and is made of physical contiguous memory pages.

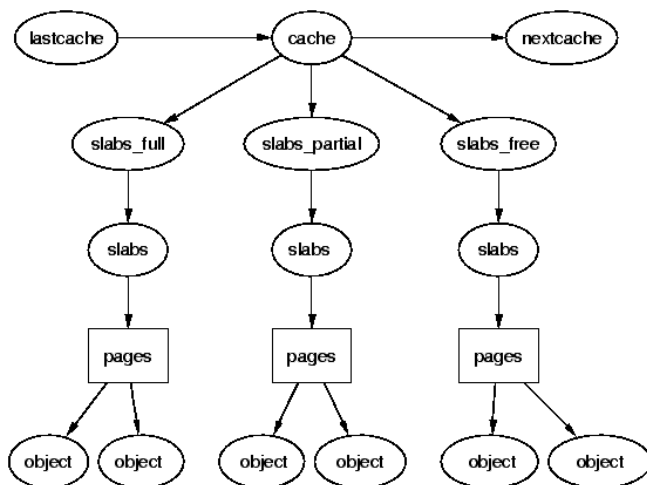
2. A **cache** consists of one or more slabs and each one is associated with a particular *kernel data structure*.

3. Each cache is thus filled up with **objects** of one particular data structure which means that, for example, the cache representing the process descriptors stores instances of process descriptors.

4. The slab allocator has three principle aims:

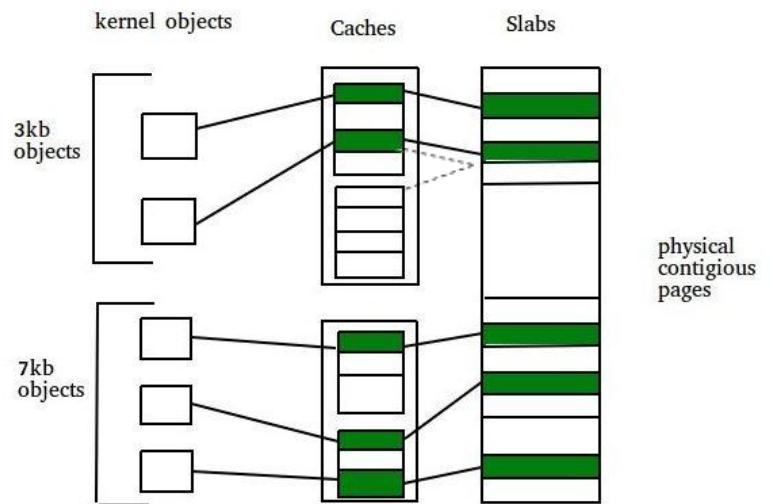
- The allocation of small blocks of memory to help eliminate internal fragmentation that would be otherwise caused by the buddy system.
- The caching of commonly used objects so that the system does not waste time allocating, initialising and destroying objects.
- The better utilisation of hardware cache by aligning objects to the L1 or L2 caches.

5. The slab allocator consists of a variable number of caches that are linked together on a doubly linked circular list called a *cache chain*. A cache, in the context of the slab allocator, is a manager for a number of objects of a particular type like the **mm\_struct** or **fs\_cache** cache and is managed by a struct **kmem\_cache\_s**. The caches are linked via the next field in the cache struct.



Layout of the Slab Allocator

The slab allocator in Linux showing 3 and 7 KB object allocation



6. **slab\_s** and **kmem\_cache\_s** structures are used to represent *slabs* and *caches* in the linux system.

7. During the allocation, a **slab** may be in any one of the 3 states : **Full**, **Empty** or **Partial**. The allocator first attempts to satisfy the request with a free object in a partial slab. If none exist, a free object is assigned from an empty slab. If not present, a new slab is formed.



## B. Virtual Memory Management

The Linux virtual memory system is responsible for maintaining the address space visible to each process. It maintains two separate views of the process's address space :

- *As a set of separate regions* : This is also known as the *logical view* of the address space. In this, address space is divided into *non-overlapping regions* where each represents a subset of the address space.

The `vm_area_struct` structure is used for the identification of a region and in one address space, they are linked together as a balanced BST.

- *As a set of pages* : This is known as the *physical view* of the address space. This view is stored in the hardware page tables for the process.

The physical view is managed by a set of routines, invoked whenever a page fault is encountered. Each `vm_area_struct` contains a field pointing to a table of functions which actually handle any page faults that occur for a given virtual memory region.

### Regions

The basis of definition of regions is the *type of backing store for the region*. It is just a store which defines where the *pages come from*. Memory regions can be backed in three ways -

- *Backed by nothing* : this region represents the **demand zero memory** which means that any page read in this region is simply a page of memory filled with zeroes.
- *Backed by a file* : this type of memory region acts as a window to the section of that particular file. This means whenever a process tries to access a page in that region, it gets back a page address from the kernel's page cache, with the appropriate offset to go to *that* region of the file. Note that any change done to the region is immediately visible to any process mapped to that space.

Virtual memory region can also be defined by its *reaction to writes*. The mapping based on reaction to writes can be **private or shared**. If a process writes to a **privately** mapped region, then the change is done by a copy-on write procedure whereas in a **shared** region, the changes are visible immediately to all processes.

### Lifetime

A new virtual address space is created in two cases :

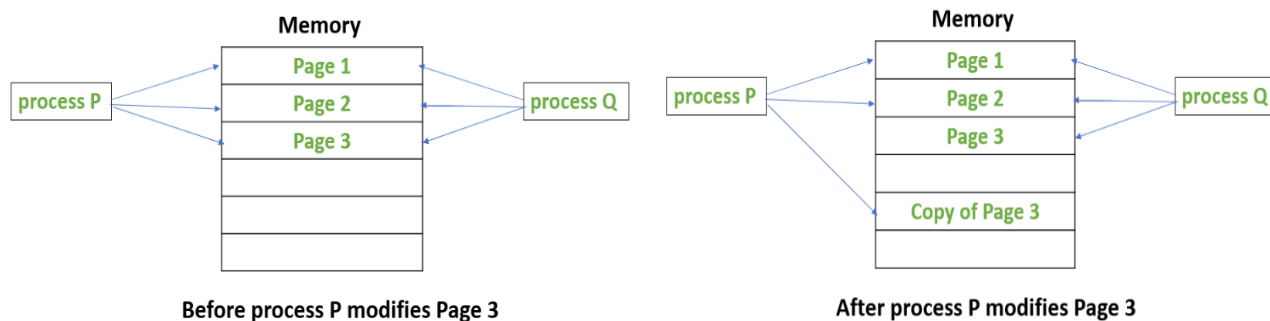
- *A new program is created with `exec()`* : On the execution of a new program, the process is given a *new, completely empty virtual address space*. The functions which

load them are responsible for associating this address space with virtual memory regions

- A new process is created with `fork()` : During forking, a copy of the existing process's virtual address space is made. The `vm_area_struct` descriptors are copied and new page tables, populated with the parent's page table, are created for the child process.

A special case occurs when a copying operation is being performed on a privately mapped virtual memory region. If a page has been marked as *private to a process* then any change by the parent must not affect the page for its children. So, whenever a write operation takes place on that page, it must not be changed for the children and thus a **new copy** of that page is made for the parent to refer to subsequently.

If the page is private and **not shared**, then the page is simply modified and not copied.



Copy-on-write procedure in child process Q and parent process P

## Swapping and Paging

A virtual memory system has an important task of relocation of pages from physical memory to disk, when that memory is needed by some other processes.

The Linux kernel relies on *paging* to carry out this task. Individual pages of virtual memory are moved between physical memory and the disk to free up memory space.

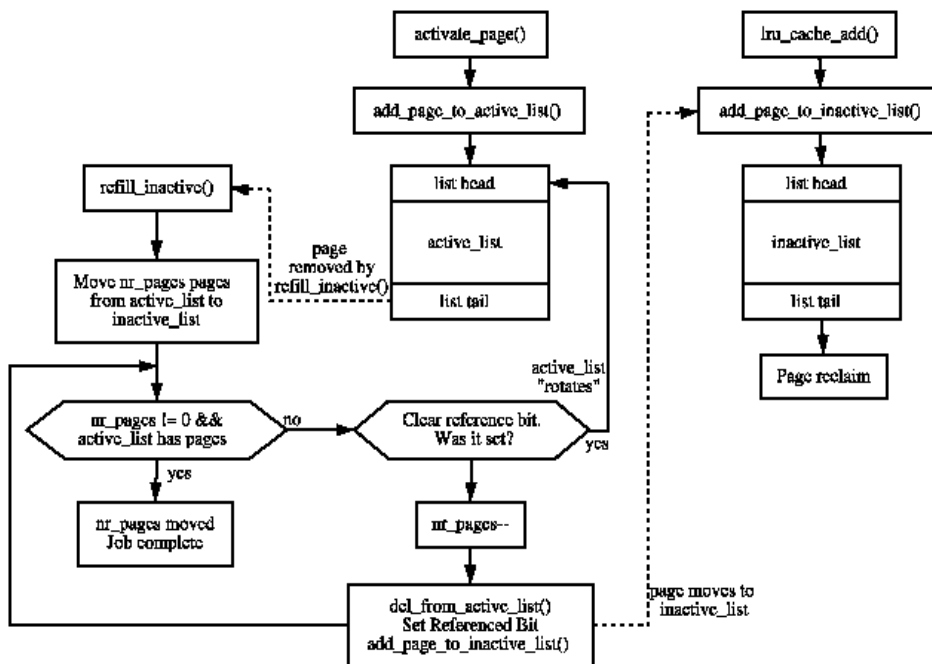
The paging system contains two main sections -

- **Policy algorithm** decides which pages to write out to the disk and when to write them
- **Paging mechanism** carries out transfer and brings the data back, in form of pages, into the main memory when needed.

## Page Replacement Algorithm used

The methods Linux uses to select pages are empirical in nature and the theory behind the approach is based on multiple different ideas. The Linux kernel employs an algorithm *similar* to the **LRU page replacement** policy.

- The LRU algorithm in Linux consists of two lists called the **active\_list** and **inactive\_list**. The **active\_list** contains the working set of all the processes and **inactive\_list** to contain the reclaim candidates. This LRU policy is a *global page replacement* policy.



- The lists resemble *LRU 2Q* where two lists called **Am** and **A1** are maintained. With *LRU 2Q*, pages when first allocated are placed on a FIFO queue called **A1(inactive)**. If they are referenced while on that queue, they are placed in a normal LRU managed list called **Am(active)**.

- When pages reach the bottom of the **active list**, the referenced flag is checked, if it

is set, it is moved back to the top of the list and the next page checked. If it is cleared, it is moved to the **inactive\_list**.

### Deviations from the classic LRU

- This policy does not satisfy the *inclusion property*. The location of pages in the lists depends heavily upon the *size of the lists* as opposed to the *time of last reference*.
- Also, the lists are almost ignored when paging out from processes as pageout decisions are related to their location in the virtual address space of the process rather than the location within the page lists

## Kernel Virtual Memory

Linux reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use.

The page entries pertaining to these pages are marked as *protected* and these pages are not visible or modifiable while the processor is running in the user mode.

This virtual memory area consists of two main regions -

- **Static area** : the core of the kernel and the pages allocated by the normal page allocator reside in this region. It contains page table references to every available physical page of memory in the system.
- **Remaining area** is a general purpose memory space area. Page entries in this range can be modified by the kernel to point to any other area in the memory. The kernel provides two facilities that allow kernel code to use this virtual memory. **vmalloc** function creates an arbitrary number of physical pages that may not be physically contiguous into a single virtually contiguous kernel memory. **vremap()** function maps a sequence of virtual addresses to an area of memory used by a device driver for memory mapped I/O.

## C. Execution and Loading of User Programs

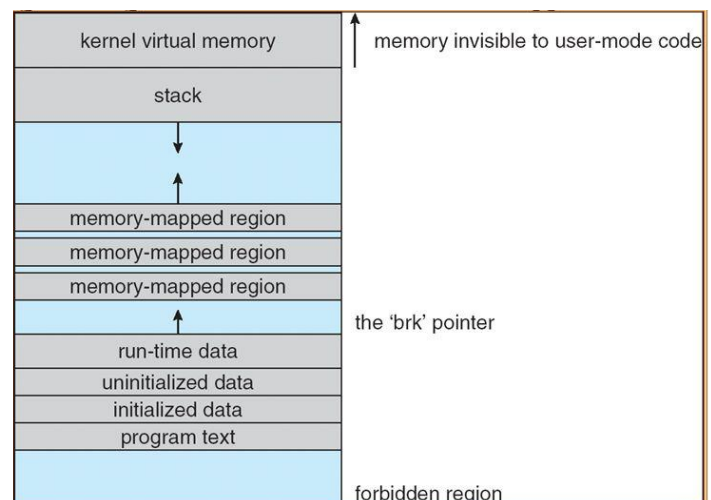
The Debian kernel's execution of user programs is triggered by a call to the **exec()** system call. This call commands the kernel to run a new program overwriting the execution context of the program which is called **exec()**.

The two main tasks of the kernel are to first *load the program* into the physical memory of the system and the second, *to execute the program* by linking the appropriate libraries and modules referenced in the program.

Since Debian is a Linux based operating system, it uses the **ELF(executable and linkable format)** as the default format and also has support for **a.out** binary format for the binary executables files present in the system. We now see how the mapping and the execution of the same are done by the OS.

### Mapping programs to memory

- Since the Linux kernel follows *demand paging*, the whole binary file is not loaded into the physical memory but the pages of the file are mapped to the virtual address space.
- The ELF-format consists of a header followed by several page-aligned sections. The ELF-loader maps these sections to separate regions of the virtual memory.



Memory mapping of a program into the physical memory

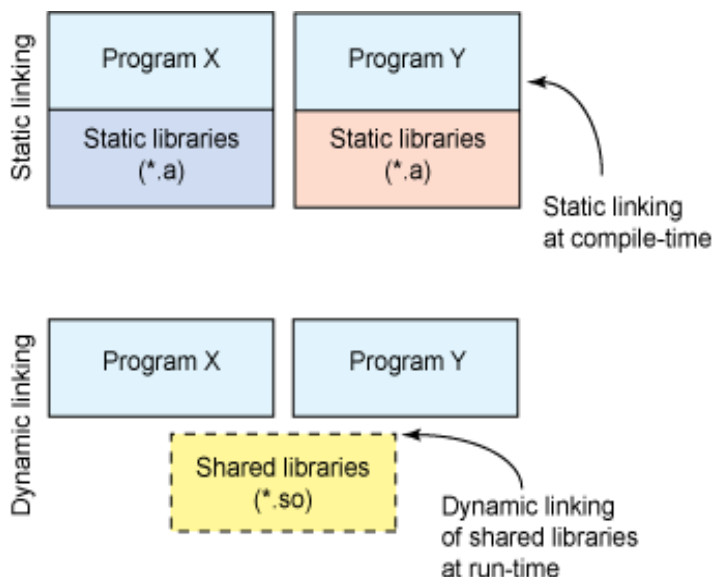
- In the top part the space is reserved for the address space of the kernel and the rest of the virtual memory is available for the programs.
- The loader's job is to now initialize the **stack** and **data / text** regions which are important for starting the execution of the program itself.
- The **stack** includes the copies of the arguments and the environment variables given to the program in the **exec()** system call. Other regions including the read-only, writable and uninitialized data are mapped towards the end of the memory available with the system.
- Each process has a pointer named **brk** (the break pointer), that points to the current extent of a variable data region. This variable region is thus expanded and contracted by changing the value of **brk** pointer using the **sbrk()** system call.

Once all these mappings have been set up, the loader initializes the process's program counter with the starting point in the ELF header and thus it is scheduled.

## Static and Dynamic linking

Since there is a lot of redundancy in the process of *static linking* of the external libraries used by a process, the Debian kernel uses *dynamic linking* to link the external system libraries used by the process during the run time.

The system libraries are loaded into the physical memory only once and any subsequent process referring to them does not have to provide the whole binary executable code for those libraries but rather a reference to the memory location where those libraries are present.



- Debian kernel implements dynamic linking in user mode using a special *linker library*.
- Every dynamically linked program contains a small piece of code which maps the link library into memory and runs the code that the routine, which is linked, contains.
- Link library determines which dynamic libraries are required by a program and thus maps these libraries in the middle of the virtual memory address space.
- These libraries are compiled as **position independent code(PIC)** and thus can run at any address in the memory.

# Disk Management

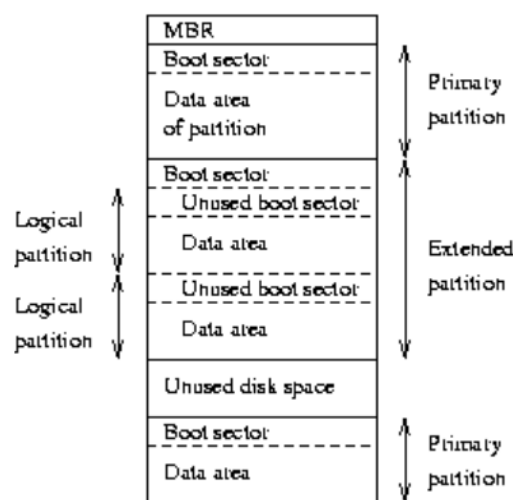
Disk management is a very broad set of responsibilities for the OS that involves creating partitions, managing the creation and maintenance of a file system for easy access to the data stored on a disk, keeping track of the free disk sectors and monitor for data corruption or loss in allocated sectors while also scheduling the disk I/O requests to maximize the disk bandwidth while minimizing the random access time.

## Disk Partitioning

Disk partitioning is dividing a disk into smaller sections that are treated as independent disks by an operating system that supports partitioning. Partitioning enables many features like:

- The ability to install multiple operating systems on a disk but in different partitions
- Each partition can have its own file system or be left unformatted/raw for special applications like large databases or for swap spaces.
- We can implement different security processes for each partition like encryption and disk locking algorithms without affecting the other partitions.

The Linux kernel supports disk partitioning and uses the MBR partition table implementation. In this, we can create upto 4 primary partitions, after which we need to create an extended partition which is a primary partition that is divided into logical partitions.



The MBR is stored in the first sector of the disk and each partition has a boot sector where there is a small program that loads the operating system installed in that partition into memory. During the boot process, the partition table present in the MBR is read by the BIOS and the first sector of the partition with an operating system installed (boot sector) is read.

In Debian, partitions are created during installation using the GUI partitioning tool. Even if we don't choose to create any partitions, Debian will create a dedicated swap partition automatically.

We can also create partitions at a later time in the command line with the help of the `fdisk` command. This command provides an interactive process to create a partition.

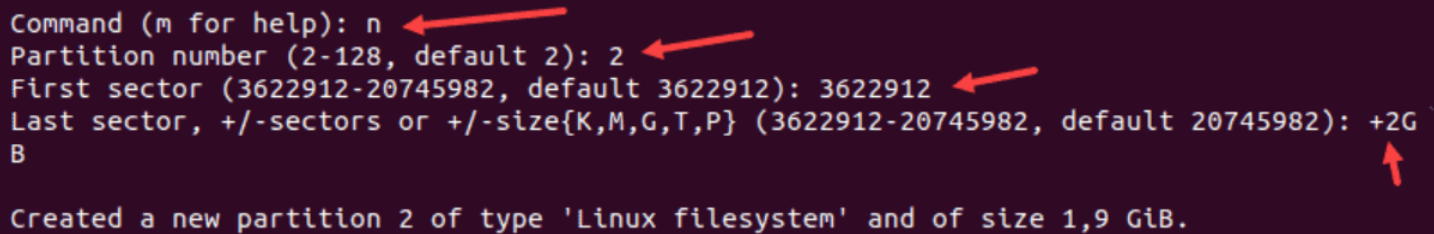
We start by executing the command:

```
sudo fdisk /dev/sdb
```

where, `/dev/sdb` represents the disk we are partitioning.

This runs the `fdisk` program where we are provided with an interactive command line where we can issue commands like `n` (to create a new partition) and `w` (to write the changes we have made into disk). We need to execute the command `n` to create a new partition.

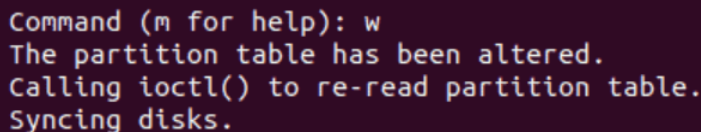
We are then prompted to choose the number that will be assigned to this partition, followed by the starting sector of this partition and then the last sector number. We specify the last sector as a displacement from the starting sector number or as a capacity of the partition and the tool automatically calculates the required number of sectors.



```
Command (m for help): n
Partition number (2-128, default 2): 2
First sector (3622912-20745982, default 3622912): 3622912
Last sector, +/-sectors or +/-size{K,M,G,T,P} (3622912-20745982, default 20745982): +2GB
Created a new partition 2 of type 'Linux filesystem' and of size 1,9 GiB.
```

Linux filesystems use 1K blocks (2 sectors) thus, each partition is recommended to have an even number of sectors in each partition.

Once the partition is created, it is still in memory. In order for this to persist, we have to write it to the disk using the `w` command.



```
Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

## Disk Formatting

This is the logical formatting process that involves creating a file system for the partition to use. There are many different file systems supported by the kernel, like `ext3`, `ext4`, `btrfs` and even `NTFS` through the `ntfs-3g` project. We can choose any of these file systems to format a disk partition.



We can format the disk using a GUI tool or through the command line using the `mkfs` (make filesystem) command.

We run the command as:

```
sudo mkfs -t ext4 /dev/sdb1
```

Where, `sudo` runs the command with root privileges

`mkfs` is the command

`ext4` is the file system we have chosen

`/dev/sdb1` is the representation of the partition

## Mounting a Partition

Once a partition has been created, and optionally formatted (we can have partitions with no file system for special uses like for swap spaces), we need to mount it to the hierarchical directory structure at some place to be able to use it.

The directory where a mass storage device/partition is mounted on the root directory structure is called a mount point. Thus, we choose a mount point (by convention it is chosen as a directory in the `/mnt` directory) and then use the `mount` command to add it to the root directory structure. The steps are:

```
mkdir /mnt/sdb1, to create the directory that serves as the mount point
```

```
sudo mount -t auto /dev/sdb1 /mnt/sdb1
```

Where, `/dev/sdb1` represents the partition and `/mnt/sdb1` is the mount point.

Now through this, mount point we can access the newly created partition.

## Disk Scheduling

Debian leaves the disk scheduling operations up to its LINUX kernel. The LINUX kernel, being an open-source project, has many different I/O schedulers developed for it by the community. The user can install and use any of these schedulers based on their use case. This allows the users of Debian to have the best disk throughput for any environment, like workstations, database servers, application servers, handheld devices etc with any type of disk storage options like spinning magnetic storage devices or flash storage and solid state devices with no moving parts.

One can see the scheduler being used by their system by examining the contents of the `/sys/block/sda/queue/scheduler` file.



```
→ jha-9 cat /sys/block/sda/queue/scheduler  
[noop]
```

The most common disk scheduling algorithms available are:

### 1) NOOP

NOOP I/O scheduler is a very simple scheduler that places all the incoming I/O requests from all the processes running on the system, regardless of the type of operation (read, write, seek etc), into a FIFO queue. The only attempt to improve throughput by this scheduler is request merging. This scheduler has less overhead and is good for storage devices without mechanical components like SSDs and flash storage.

### 2) Anticipatory

This scheduler tries to anticipate which block will be requested next in any future I/O requests and thus tries to re-order the current request and uses a one-way elevator (Circular SCAN) to improve locality and reduce disk seeks.

This scheduler is very beneficial for systems where read operations are very common like with web servers. Infact, a considerable increase in throughput is observed with the Anticipatory I/O scheduler in Apache Web Servers.

### 3) Deadline

This scheduler intends to prevent starvation of any I/O requests caused due to re-ordering of requests during scheduling by providing each request with a deadline. It maintains two queues, one for read requests and the other for write requests sorted by their block numbers. In addition it maintains two other queues for read and write requests sorted based on the shortest deadline first.

This scheduler gives higher priority to the read queue as mostly, processes are blocked until the data from a read operation is available (Thus, read requests are given deadlines of 500ms). While, write operations are cached in main memory so the process continues execution even if the data hasn't been written to the disk yet (Thus, write requests are given deadlines of 5 seconds).

When servicing requests, it first decides which queue to service and checks if the deadline of the 1st request in the deadline queue has passed. If so, that request is serviced immediately. If the deadline has not been reached, then the scheduler serves a batch of requests that have been sorted according to their block number so there are minimum disk seeks.

This scheduler is mostly used with real time systems where servicing I/O requests can't be delayed too much.

#### 4) CFQ (Completely Fair Queuing)

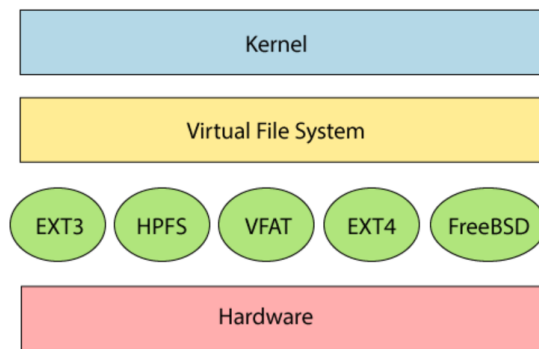
This is the current default scheduler of the LINUX kernel but it can be changed during the boot time by changing the GRUB configuration file at /boot/grub/menu.lst. This scheduler focuses on servicing disk I/O requests based on the processes that are making those requests. It maintains a per process FIFO queue and then provides these queues with a time slice to access the disk. After the time slice expires, the next processes queue has access to the disk to perform its disk operations.

The CFQ scheduler gives all users of a device an equal number of I/O requests over a time interval and thus is good for multiuser/multitasking systems.

Debian's kernel supports many file systems like the ext file systems, btrfs, ntfs, nfs, xfs, zfs etc and the user can format their drives/partitions with any file system they want. But over the years, the default file system for LINUX distributions has been the ext (extended) file system. There have been 4 versions of this called ext, ext2, ext3 and ext4.

## VFS

Since the kernel supports so many different file systems, the operating system has a layer of abstraction between the different filesystems and the applications that standardizes the API for requesting file system operations.



## Ext4

This is the most recent and the default file system used by Debian. It uses the MBR for partition identification, uses 48-bit internal addressing with 4KB blocks thus, allowing file sizes of upto 16 Terabytes and disk sizes of upto 1 Exabytes. It implements the directory structure using a hashed B-tree and keeps track of free sectors using a linked list. It maintains a list of bad blocks in a table and uses journaling to keep the disk in a consistent state in case of a failure during a disk write.

This file system improved on its predecessor, suffering very little with fragmentation problems by using **extents** (a large block of free space containing several adjacent sectors) and **delayed allocation**. In delayed allocation, the kernel does not actually allocate sectors to a file until it is ready to commit the data to the disk. This allows it more time to find suitable locations on the disk to reduce fragmentation. However, this improvement comes at a cost where the chances of data loss are increased as it has to stay in the cache for much longer than with ext3.

This file system also allows Debian to now be used in scientific and mission critical processes as its timestamps for file creation, modification and access are correct upto nanoseconds compared to the seconds of ext3.

### ntfs-3g

A major problem with LINUX distributions like Debian was that files created using the windows NTFS file system were difficult to work with in a LINUX environment. This is because the NTFS file system is a proprietary piece of software and thus providing support for it in open-source projects like the LINUX kernel is difficult. Since Windows is a common operating system, used in many places, this was a major limitation of operating systems like Debian.

However, eventually at least read support for the NTFS file system was built into the LINUX kernel by the community by guessing how the NTFS system worked (write support however is still not available).

ntfs-3g is an open source project that aims to provide full support for the NTFS file system in any LINUX environment. To use this, one needs to install the ntfs-3g package and then mount a ntfs partition using the command

```
ntfs-3g /dev/(partition-name) /mnt/(mount-point name)
```

Similarly, other filesystem operations like formatting a partition, creating and labelling volumes etc are also supported for the NTFS file system through this package.

# References

---

- **Books**

Operating System Concepts by *Avi Silberschatz, Peter Baer Galvin, Greg Gagne*

- **Websites and Documentations**

- <https://www.kernel.org/doc/html/latest/>
- <https://linuxjourney.com/>
- <https://www.debian.org/doc/>
- <https://opensource.com/>
- <https://www.informit.com/>
- <https://unix.stackexchange.com/>
- <https://www.networkworld.com/>
- <https://tldp.org/LDP/sag/html/partitions.html/>
- <https://wiki.debian.org/>
- <https://wiki.archlinux.org/index.php/NTFS-3G/>

- **Articles**

For thread management

<https://www.informit.com/articles/article.aspx?p=370047&seqNum=3>

For CFS scheduler

<https://opensource.com/article/19/2/fair-scheduling-linux>

For IPC

<https://opensource.com/article/19/4/interprocess-communication-linux-storage>

For Disk Management

<https://www.networkworld.com/article/3221415/linux-commands-for-managing-partitioning-troubleshooting.html>

For File Systems

<https://opensource.com/article/18/4/ext4-filesystem>