IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 1, January 2011
ISSN (Online): 1694-0814
www.IJCSI.org

A Method for Designing an Operating System for Plug and Play Bootstrap Loader USB Drive

Dr. T. Jebarajan [1], K. Siva Sankar [2]

[1] Principal, V.V Engg College
Tamil Nadu, India.

[2] Lecturer, Noorul Islam University.
Tamil Nadu, India.

Abstract

This paper lays out different issues and solutions in the design of an operating system with an inbuilt kernel memory for data storage and USB (Universal Serial Bus) drive with bootstrap loader. This relates from the minimum features required for a program to become the kernel and how this kernel should be written into the boot sector of a hard disk drive depend upon the machine architecture, so that it gets loaded into the computer memory automatically and it restores the disk drive in to its original state. It highlights how this operating system can be made to support user specific authentication, keyboard, networking, peripherals, file system access etc. Most of the frequently used drivers are added to the kernel images. This also lays out the specifications for a shell to issue system commands and system utilities, interfacing FAT (File Allocation Table) file system for smooth boot of an operating system and how to communicate with another similar system using hardware device.

Keywords- *USB flash drive, plug and play, kernel memory, boot loader, shell*

1. INTRODUCTION

Any computer system can be considered to have four basic components [1] [2] - users, application programs, operating system and hardware. The hardware, comprising of central processing unit, memory and input output devices provides the basic computing resources.

The operating system provides a platform for proper use of these resources [6] [7]. It can be considered as a program that manages the computer hardware or as an intermediary between a user of a computer and the computer hardware. The application programs that run on top of the operating system provide the users with solutions they are looking for. This paper will explore the design of a small operating system for and plug and play USB device, which can be built using C and assembly language. In addition to this some modules are configured with this operating system.

2. BUILD ENVIRONMENT

In order to start building, a development box running on Windows or Linux with a C editor, C compiler, Nasm is required. It requires a target machine for the operating system. It supports a range of object file formats, including Linux and *BSD a.out, ELF, COFF, Mach-O, Microsoft 16-bit OBJ, Win32 and Win64. It will also output plain binary files. Its syntax is designed to be simple and easy to understand, similar to Intel's but less complex for fast accessing. The design and bootstrap strategy will vary with the underlying machine architecture. For this design, consider the target machine as an Intel x86 based hardware with minimum 1MB RAM, USB disk drive, keyboard and monitor.

3. DESIGN OF SYSTEM

A Linux-based system is a modular Unix-like operating system. It derives much of its basic design from principles established in linux. Such a system uses a monolithic kernel version 2.26, the Linux kernel, which handles process control, networking, peripheral and file system access. Device drivers are integrated directly with the kernel

Separate projects that interface with the kernel provide much of the system's higher-level functionality. The user land is an important part of most Linux-based systems, providing the most common implementation of the C library, a popular shell, and many of the common Unix tools which carry out many basic operating system tasks. The graphical user interface (or GUI) used by most Linux systems is built on top of an implementation of the X Window System.

In order to design such a system it can divide it into different logical modules [11], Boot Loader, Kernel, FAT File System, Reverse Mapping, Initial Ramdisk, and Packaging.

3.1 Boot Loader

Boot Loader program loads the kernel of the operating system into the main memory for execution [1] [4]. The Boot Loader must be of size 512 bytes and should reside in the first sector of the disk drive. The conventional

MBR code expects the MBR partition table scheme to have been used, and scans the list of (primary) partition entries in its embedded partition table to find the only one that is marked with the active flag. It then loads and runs the volume boot record for that partition.

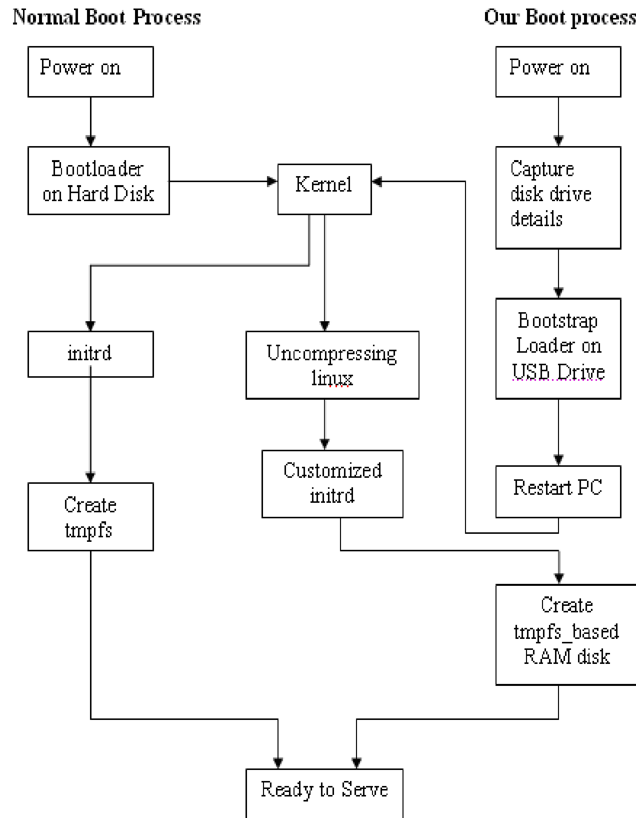

Fig.1. Boot Process comparison

The procedure of Boot Loader is as follows:
Check the boot signature 0AA55h at 510,511-th bytes of the first sector of Boot Disk. If boot signature is present, it loads the code present in the first sector (512bytes) to memory address 07c00h. Next the code at 07c00h is executed. This code then tries to find the available physical memory and divides it into 64KB pages. After that, 2 KB boot stack is allocated at (A0000-512) h and stack pointer is setup. Then Space for IVT and BIOS routines are reserved, and kernel is loaded at 00600h. The kernel that is to be loaded can be an EXE, BIN or COM file. Search for this kernel file will be conducted in the Root Directory (19th sector of the Boot Disk). On getting the file, it is allocated properly with all needed segments and memory pointers. If the kernel is in BIN or COM format it will have a single segment with all DS, CS, ES, SS integrated. If the kernel is in EXE format, it will have separate code, data, extra and stack segments. In such cases the exe header will be ripped off and proper relocation factors are added as needed. After this, the loader loads the kernel into memory.

3.2 Kernel

Kernel is a nothing but a program that resides in the memory, takes in user inputs, process those user inputs and give out a suitable response. Kernel can be EXE, BIN or COM file. Though the tasks done by the kernel varies with design [5], and many operating systems delegate system functions to different layers or child programs, the absolute minimal functions common to all other sub-systems should be kept in the kernel. In this design, the shell is the main component. This shell program should perform the following functions.

User Authentication – The first routine in the shell should be to authenticate the user. The username and password entered by the user is checked against stored username/password combinations in the system and if found valid, the user is given a 'prompt' from where he can issue system commands.

Command Interpreter - The user input parser is nothing but the first component of the Command Interpreter. Once the parser identifies a command as valid, an action has to be taken corresponding to the command. For example, if the command is copy, the shell may call the File Manager subsystem to read the source file and then issue another command to create and write the contents just read as a new file. Similarly all the system commands that the operating system supports can be implemented

IO Functions - Internal working of the Operating System can be based on the basic display and keyboard driver routines [2] [4]. These routines are implemented using BIOS Interrupts [4]. For Keyboard, the routines for reading a keystroke, converting it to number format, checking for keystroke presence etc are included. For Video, the routines for displaying a character, message, error message, printing at specific location on the screen, video settings, color settings etc are included. String manipulation routines are also implemented for simplicity of the high level operations.

Memory Location Routine
10h - Video Support Routines
12h - RAM size
15h - Delay Support Routines
16h - Keyboard Routines
19h - Reboot Functions
1Ah - CMOS Support Routines

That is to say, when a user presses a key, the keyboard support routes at 16H kicks in, based on the context, then appropriate action is taken. For example if the user is in typing on the shell, the video support routines at 10h will be called to display the character typed onto the screen. Any program that implements the above functions qualifies to be a kernel. Below is the pseudo code for the simplest of all kernels. The only function done by this operating system kernel is to display a message after

booting and it is invoked by the script called initrd (initial RAM disk).

```
int main()
{
char *vidmem = (char *) 0xb8000;
vidmem[0] = 'O';
vidmem[1] = 0x7;
vidmem[2] = 'S';
vidmem[3] = 0x7;
return 0;
}
```

Note that char *vidmem = (char *) 0xb8000 is the memory mapped location of video memory.

3.3 FAT File System

BIOS interrupts are available for low level disk services like reading sectors from drive, writing sectors into the drive, formatting a track etc [3]. The operating system can invoke these bios services for disk activities whenever it has to read/write into the disk drive [9]. But, in order to do low-level reading and writing on a hard drive with a FAT file system, it is required that the address assigned to files/directories by the file allocation table is converted to the absolute sector address understood by BIOS. And for this conversion, a good understanding of the underlying structure of FAT and FAT chaining is required. Disk structure has got 4 logical parts: Boot Sector, File Allocation Table (FAT), Directory and Data space. Of these, the Boot Sector contains information about how the disk is organized. That is, how many sides does it contain, how many tracks are there on each side, how many sectors are there per track, how many bytes are there per sector, etc.

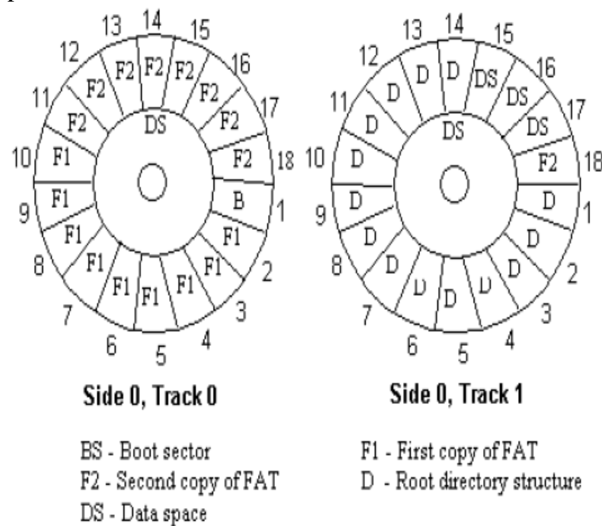

Fig.2. FAT files system architecture

The files and the directories are stored in the Data Space. The Directory contains information about the files like its attributes, name, size, etc. The FAT contains information about where the files and directories are stored in the data space. Fig.2 shows the four logical parts of a 1GB USB flash drive.The basic functions that should be supported by the operating system on the file system should be, List Files / Directories, Create Directory, Change Directory, Create File, Display File contents, Copy File, Rename File, Delete File, Modify File etc. It also makes sense to provide users with an in-built editor which can be used for creating and editing files structure.

FAT is more robust and it can relocate the root folder and use the backup copy of the file allocation table instead of the default copy. In addition, the boot record on FAT USB drives is expanded to include a backup copy of critical data structures. Therefore, FAT USB drives are less susceptible to a single point of failure than existing other file system specified drives.

3.4 Reverse Mapping

Reverse mapping, or RMAP, was implemented in the kernel to solve memory problem. Reverse mapping provides a mechanism for discovering which processes are using a given physical page of memory. Instead of traversing the page tables for every process, the memory manager now has, for each physical page, a linked list containing pointers to the page-table entries (PTEs) of every process currently mapping that page. This linked list is called a PTE chain. The PTE chain greatly increases the speed of finding those processes that are mapping a page, as shown in below

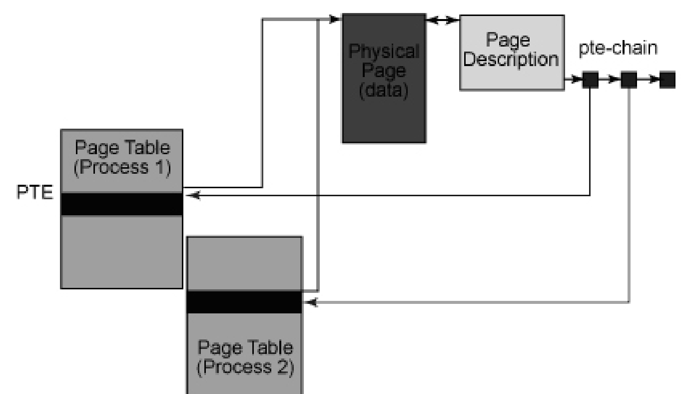

Fig. 3. Reverse mapping page description

The most notable and obvious cost of reverse mapping is that it incurs some memory overhead. Some memory has to be used to keep track of all those reverse mappings. Each entry in the PTE chain uses 4 bytes to

store a pointer to the page-table entry and an additional 4 bytes to store the pointer to the next entry on the chain. This memory must also come from low memory, which on 32-bit hardware is somewhat limited. Sometimes this can be optimized down to a single entry instead of using a linked list it should be compatible with one other.

4. IMPLEMENTATION

4.1 Initial Ramdisk

The PC has only 256MB total RAM, maybe not even any swap partition or swap file, how on earth does this operating system avoid writing to the Flash drive during a session. This is one of the key architectural points of this approach. At boot up, pup_save.2fs is mounted read-only from where it is on the Flash drive, and its contents are not copied into RAM. Instead, a tmpfs (temporary file system) in RAM holds all new and changed files [15]. This is still actually very fast, as all the "working files" are in RAM. Periodically and at end of session, those "working files" are written back to the pup_save.2fs file. This approach has a much smaller initial ramdisk file, named initrd.gz (instead of image.gz), only about 1.1MB, and these accounts for a significantly faster boot time.

This operating system tackles the problem other way round, by always booting up in ramdisk-only the first time you boot on a PC, then at shutdown you are asked if you want to create a personal storage file with different storage option. This operating system mounts the persistent storage file pup_save.sfs at the top level, that is, on root directory ("/"). The read only compressed file with all the Puppy files, pup_xxx.sfs, mounts on root directory. The kernel is configured with a 12288KB maximum ramdisk and this is increased to 13824K for this approach, so the size should be mentioned in boot parameter and can have built in memory in the operating system. This design uses a tiny initial ramdisk, initrd.gz that is only 0.9M compressed. Installing extra applications, such as dotPups that install into /root/my-applications, do not add to initrd.gz. The initial ramdisk file remains fixed in size, and everything under root directory ("/") goes into pup_save.2fs, the squashfs file that gets attached later in the boot process. The initial ramdisk file that loads into the fixed-size-limit ramdisk. The boot sequence then creates a tmpfs ramdisk, which is variable in size, and will use as much RAM and swap space as available.

4.2. Architecture Overview

The way to understand the diagram is to view each of those layers as a complete filesystem, that is, a complete directory hierarchy from root directory ("/") down. These layers are laid one on top of the other, which is achieved by the unionfs filesystem. This file will be visible at the top layer. If the "off-blue" layer has the same file, it will not be visible, as it is overlaid by the same file on a higher layer. Depending on this version of Linux you are running, the method for creating the initial RAM disk can vary. The initrd is constructed using the loop device. The loop device is a device driver that allows you to mount a file as a block device and then interpret the file system it represents [18]. The loop device may not be present in this kernel, but you can enable it through the kernel's configuration tool by selecting Device Drivers > Block Devices > Loopback Device Support. The small, but necessary, set of applications are present in the ./bin directory, including nash (not a shell, a script interpreter), insmod for loading kernel modules, and lvm (logical volume manager tools).

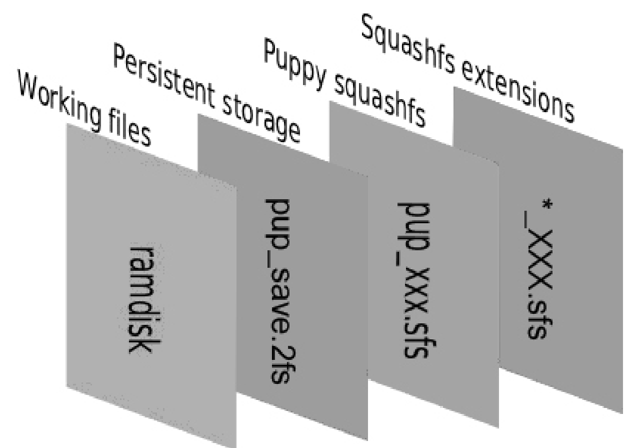

Fig. 4. Structure of Operating System

ramdisk	This is the tmpfs filesystem running in RAM, with new and updated files.
pup_save.2fs	This is the persistent storage, where all your data, settings, email, installed packages, etc., get saved permanently. The ".2fs" means that the file contains a FAT or ext2 filesystem.
pup_xxx.sfs	The built-in applications, window manager, scripts, everything. The ".sfs" means the file contains a squashfs compressed filesystem. The "xxx" is the version number without the dots.
*_xxx.sfs	These are additional squashfs files. The "*" can be anything. For example, devx_xxx.sfs is the complete environment for compiling C/C++ applications

While running this operating system, the outlook seen is one filesystem, which is the top layer. Thus you see /usr/lib/libgdkxft.so and you don't care what layer it is actually on. An exciting alternative to the squashfs extensions is to use an existing installed Linux distro as the bottom layer.

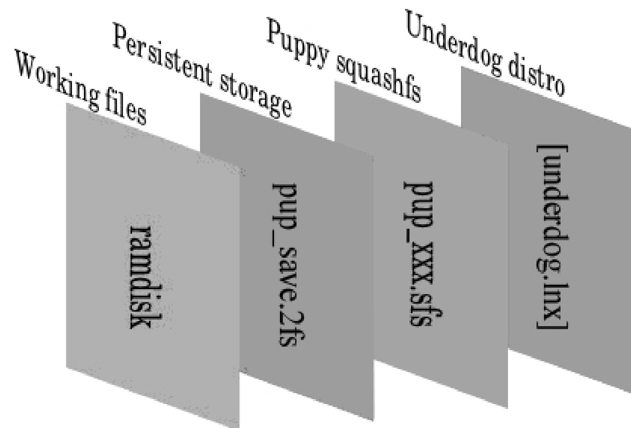

Fig. 5. Operating System Distro as the bottom layer

There are other variations, and it has a "state variable" named PUPMODE that shows what state, the operating system currently used. There is a file, etc/rc.d/PUPSTATE that has the PUPMODE variable defined in the following modes

- PUPMODE 5
- PUPMODE 12
- PUPMODE 13
- PUPMODE 2
- PUPMODE 77

4.3. PUPMODE 5

This is the configuration the very first time that operating system is booted from USB Flash drive. The first time that you plug in the USB and boot up, there is no persistent storage, and the "union" consists of only two layers, the top "working files" and the pup_xxx.sfs squashfs filesystem that has all the operating system files [18]. These two layers appear overlaid at root directory; however they can be viewed individually, at their respective mount points. So, we describe this approach but not touching the hard drive at all. You can run applications, configure, download, install packages, but it is all happening in the tmpfs ramdisk, so not getting saved. The way that we have been using pupmode is to create a "pup100" file on the USB drive, which has a FAT file system. This file is copied into RAM at boot up, if there is enough RAM, thus avoiding writes to the Flash drive during a session. Then the files are copied back at shutdown.

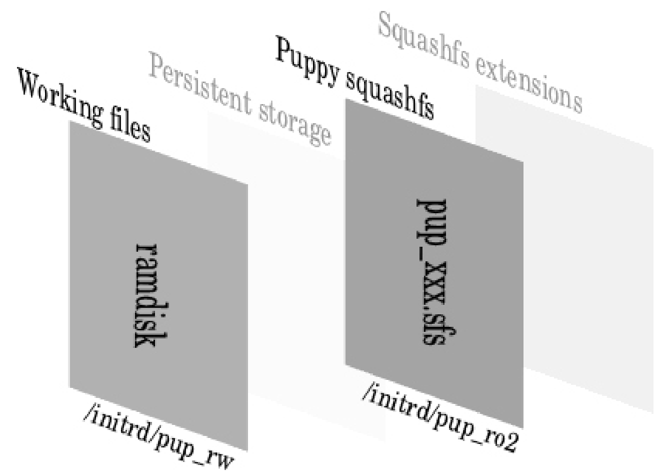

Fig. 6. First time boot configuration of USB flash drive

The amount of space you have in the ramdisk depends on how much RAM is in the PC. The really interesting part is when you decide to end the session and shutdown the PC. The shutdown script, which is actually /etc/rc.d/rc.shutdown, will execute and will bring up a dialog window asking you to save the session with different allocation of memory size. Whatever directories and files that have been created in the ramdisk can now be saved [18]. The choice of storage location depends on whether a partition is a Linux filesystem, or FAT filesystem, a file called pup_save.2fs can be created in USB flash drive and stored periodically every time.

4.4 PUPMODE 13

Configure an operating system to a USB Flash drive, perhaps by using the operating system Universal Installer program,you will have a bootable drive with the files vmlinuz (the Linux kernel), initrd.gz (the initial ramdisk), pup_xxx.sfs (squashfs filesystem with all the os files) and syslinux.cfg (Syslinux config file). The situation is just like booting from a live-CD on first boot of this operating system and it will be in PUPMODE 5, as no persistent storage has yet been created. On first shutdown, as described in the PUPMODE 5 section above, you will create a persistent storage called pup_save.2fs file [18]. On the second boot, operating system will discover the persistent storage and boots. In the case of the persistent storage on Flash memory, which is the second layer, operating system will save everything from the top layer to the second layer every 30 minutes.

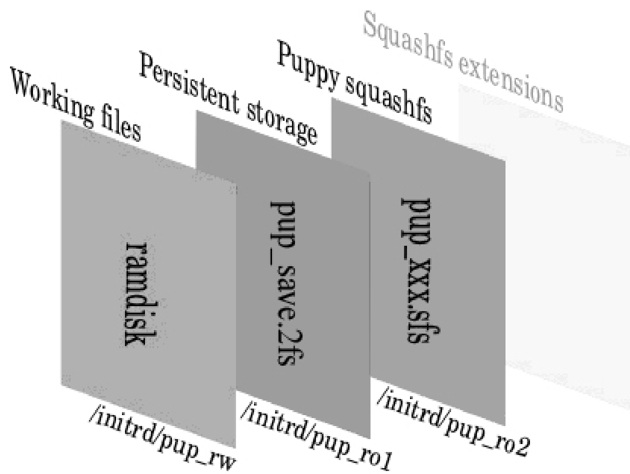

Fig. 7. Second time boot configuration of USB flash drive

From the "unionfs" point of view, the second layer is mounted read-only, it is only the top layer that is written to, however this design can able to "flush" the top layer down to the next layer at periodic intervals [18]. This has an option to select the user storage in and operating system, according to this user can use two storage one in operating system and another in normal disk drive. The updating made in the operating system is stored in that particular space allotted for it.

4.5 Flash technology

However, there is a downside to flash technology, and that is it is not designed for unlimited writes. That is, you can save onto it just so many times, and then it will collapse [16].

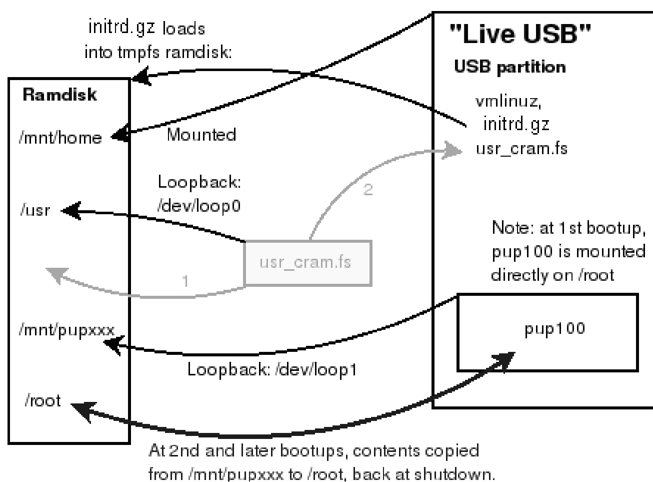

Fig. 8. Layout of Operating System at second and later boot ups from a USB device

Operating systems is especially designed to have no writes to the Flash drive during a session, enormously extending its life span. When operating system boots from USB, the steps are much the same as for the live-CD. The kernel vmlinuz is loaded into RAM, initrd.gz is uncompressed and loaded into a ramdisk, the ram disk is responsible of loading all the operating system modules. Take a look at this fig.8 that boots the operating system from the USB drive and the structure remains same for upcoming extraction.

Operating system with no writes to flash device. usr_cram.fs will find in the USB partition and will mount it on /usr. If there is enough RAM, it will copy usr_cram.fs into the ramdisk and then mount it on /usr. This will slow down bootup slightly, but will improve running speed even if the operating system does leave usr_cram.fs on the USB drive and mounts it from there onto /usr, that is not a problem as /usr is read-only [18]. There will be no writes to /usr, so the lifetime of the Flash drive is not compromised.

4.6 Packaging

Once it has the kernel with all required functionalities ready, it requires writing it into a medium like USB disk drives. Remember that the boot loader always loads the executable in one specific sector of the drive. So it is important that to place the program in correct sector for the boot loader to find it and load it into memory [19]. Tools like masm and debug allows writing the executable program to specific sectors that specify.

5. CONCLUCSION

The design parameters of an operating system, with minimal components are described by considering all issues. This is of great thrust to completely design and develop the underlying principles of an operating system and boot strap loader in USB drive. And this understanding of the working platform is critical to developing better software that runs on it.

REFERENCES

[1] Silberschatz, Galvin, and Gagne, “Operating System Concepts,” Wiley, Seventh Edition Wiley, 2006
[2] A S Tanenbaum, “Operating System Concepts,” 3rd ed., Oxford:Clarendon, 1992
[3] Dominic Giampaolo, “Practical File System Design with the Be File System,” Morgan Kaufmann publishers, 1999
[4] William Stallings "Computer Organization and Architecture: Designing for Performance", Prentice Hall, 2009

[5] Butler W. Lampson, and Howard E. Sturgis "Reflections on an operating system design", Communications of the ACM, Volume 19, Issue 5, pp. 251-265 , 1976
[6] A. Messer, and T. Wilkinson, “Components for operating system design”, Proceedings of the 5th International Workshop on Object Orientation in Operating Systems, IEEE Press, 1996
[7] Christine Morin, “Design and Implementation of First Advanced Version of LinuxSSI”, INRIA, Campus de Beaulieu, France, 2008
[8] William Stallings, “Operating Systems: Internals and Design Principles,” Prentice Hall, Fifth Edition, 2005
[9] Lex Stein, "Stupid File Systems Are Better", Proceedings from the Eighth Workshop of Hot Topics in Operating Systems, IEEE Press, 2005
[10] A Bruce Carlson, Paul B Crilly, and Janet Rutledge, “Communications Systems,” Mc Graw Hill, 2001
[11] Craig Larman, Victor R. Basili, “Iterative and Incremental Development: A Brief History”, IEEE Computer Society Press, Volume 36, Issue 6, pp. 47-56, 2003
[12] Jesshope C, Shafarenko A, "Concurrency engineering", Proceedings from IEEE Computer Systems Architecture Conference,” IEEE Press, 2008
[13] Tanenbaum, A.S, Herder, J.N, Bos, H, “Can we make operating systems reliable and secure?,” Computer, Volume 39, Issue 5, pp 44-51, IEEE Press, 2006
[14] Geer, D “The OS Faces a Brave New World,” Computer, Vol 42, Issue 10, pp 15-17, IEEE Press, 2009
[15] W. Tukey, “Bias and confidence in not-quite large samples,” *Annals of Mathematical Statistics*, vol. 29, p. 614, 1958.
[16] B. Efron, “The jacknife, the bootstrap, and other resampling plans,” in *Proc. Conf. Rec. SIAM*, Philadelphia, PA, 1982.
[17] Y. D’Asseler, C. J. Groiselle, H. C. Gifford, S. Vandenberghe, R. Van De Walle, I. L. Lemahieu, and S. J. Glick, “Evaluating human observer performance for list mode PET using the bootstrap method,” *IEEE Trans.Nucl. Sci.*, submitted for publication.
[18]http://www.puppylinux.com/development/howpuppyworks.html
[19] C. J. Groiselle, Y. D’Asseler, H. C. Gifford, and S. J. Glick, “Performance evaluation of the channelized Hotelling observer using bootstrap list-mode PET studies,” in *Proc. IEEE Medical Imaging Conf.*, Portland, OR, 2003, pp. 2511–2515.