PEDAGOGICAL TOOLS FOR SYSTEM SOFTWARE AND OPERATING SYSTEM COURSES USING XV6 KERNEL

A Project Report

Submitted by

Prashant Gonarkar	**111003026**
Dhanesh Arole	**111003044**
Prasannjit Gondachwar	**111003039**

in partial fulfilment for the award of the degree

of

B.Tech. Computer Engineering

Under the guidance of

Prof. Abhijit A. M.

College of Engineering, Pune

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,

COLLEGE OF ENGINEERING, PUNE-5

May, 2014

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,

COLLEGE OF ENGINEERING, PUNE

CERTIFICATE

Certified that this project, titled "PEDAGOGICAL TOOL FOR SYSTEM SOFTWARE AND OPERATING SYSTEM COURSES USING XV6 KERNEL" has been successfully completed by

Prashant Gonarkar	**111003026**
Dhanesh Arole	**111003044**
Prasannjit Gondachwar	**111003039**

and is approved for the partial fulfilment of the requirements for the degree of "B.Tech. Computer Engineering".

SIGNATURE	SIGNATURE
Prof. Abhijit A.M.	**Dr. J. V. Aghav**
Project Guide	**Head**
Department of Computer Engineering	**Department of Computer Engineering**
and Information Technology,	**and Information Technology,**
College of Engineering Pune,	**College of Engineering Pune,**
Shivajinagar, Pune - 5.	**Shivajinagar, Pune - 5.**

Abstract

The current widely used production level system software utilities such as *GNU* compiler, linker, interpreter and C library have been in active development for almost last 3 decades. But with the rise of *Linux* Kernel and *GNU* toolchain based operating systems in mid 1990s, these two softwares got evolved with strong mutual cohesion. System softwares such as *GCC*, *ld* and *Linux* kernel are heavily dependant on each other such that *Linux* kernel can only be compiled using *GCC* bundle where *GCC* runs on standard *libc* and only *GNU* linkage editor can link the programs generated by *GCC*. All of these tools carry out critical functionalities of operating system such as memory management, file system, resource management, access control and synchronization etc. So inherent required code for this functionality is highly optimised and written by leveraging several rare language features which makes these softwares very much complex. As a result it is always challenging job for educators to design a semester long, fundamental course of operating system covering theoretical and practical aspects of above mentioned tools. This project is aimed at delineating a framework for development of fresh operating system whose essential component should include platform specific C library, linkage editor and interpreter.

As a part of this project we demonstrated working of our proposed framework by porting an interpreter named *PicoC*, C library named *dietlibc* and small in house linkage editor to *Xv6* operating system. This report will elaborate on strategies to be used while porting system softwares to new kernel as well as on comprehensive measures that can be used in undergraduate as well as graduate courses of operating systems and system programming for better understanding of their basic functionality in pedagogy purposes.

Contents

List of Tables

List of Figures

List of Symbols

Chapter 1

Introduction

1.1 Need of developing a prototype Operating System:

After great success of *Linux* kernel, more and more developers (nearly 20391 for v3.5) across the world are finding greater interest in the particular model adopted for it's development. Also Operating System is conventionally regarded as the most difficult as well as intriguing piece of code. It is been observed through some of the formal methods of public opinion gathering and social network data that it is the most sought after software and college level course in undergraduate programs of computer science engineering. But rather a contradictory result shows that average age group of latest Linus kernels contributors is 36 years which is much more than expected age of 25 years. A deep inquiry into this fact poses a several question regarding the current method of teaching OS at entry level college courses.

Most modern Kernels provide many more system calls and many more kinds of services but due to their production values it is almost impossible to get the real crux of how these systems have been implemented in their code base. As a result though student grasps the theoretical knowledge about OS concepts, one remains marginally aloof of the core implementation details which restraints him from participating in active OS development.

Hence most fundamental aim of this project is to develop small prototype *Unix* like OS that can be essentially used for education purposes. The main reason behind developing such OS is to help wide spread student community in understanding the basics of OS through practical demonstrations. Such OS can be used by educators around the world for teaching basic working of system softwares and their interaction with different OS components.

Hence considering this we have developed a small OS which can be used and easily learnt too. But only developing such small OS wouldn't suffice. As a result the important part of this project is to develop some built in tool in such small educational OS to give real time insight of it. And to produce a comprehensive course material containing the documentation, tutorials and set of assignments that will probably give more deeper insight into code base of this OS.

1.2 Components of education OS:

The critical issue in this project is to decide which softwares to be ported into such educational purpose OS. Because considering the prime aim of this project it is very much necessary to maintain the overall size of code base relatively small enough for understanding of student level developers. In every OS, language processing activities arise due to the differences between the manner in which a software designer expresses the ideas and way computer instructions are understood by computer hardwares. So there is tight binding between system softwares which convert high level programming language instructs into machine language and OS which is responsible for executing that program. Hence this OS includes a small interpreter named *PicoC* and a standalone linkage editor developed specifically for this purpose.

1.3 Reason for porting Interpreter in education OS:

During the software development cycle, programmer make frequent changes to source code. When using a compiler, each time a change is made to the source, they must wait for the compiler to translate the altered source files and link all of the binary code files together before the program can be executed. The larger the program , the longer the wait. By contract a programmer using an interpreter does a lot less waiting, as the interpreter usually just needs to translate the code being worked on to an intermediate representation (or in other words Not translate it at all), thus requiring much less time before the changes can be tested since effects are evident upon saving the source and reloading the program. Interpreting a language given implementations some additional flexibility over compiled implementations. Features that are easier to implement in interpreters than in compilers include:

1. Platform Independence
2. Reflection and reflective use of evaluator
3. Dynamic typing
4. Smaller executable program size (since implementations have flexibility to choose instruction code)
5. Dynamic Binding.

Also in recent times, rise of agile software development movement and cross platform language implementation, languages such as Java and Python have become by default choice of many software architects because of their comparatively lesser time of development and source readability.

1.4 Reason for porting standalone Linkage editor to OS:

1. Understanding linkers will help you build large programs.Programmers who build large programs often encounter linker errors caused by missing modules, missing libraries, or incompatible library versions. Unless you understand how a linker resolves references, what a library is, and how a linker uses a library to resolve references, these kinds of errors will be baffling and frustrating.

2. Understanding linkers will help you avoid dangerous programming errors The decisions that Unix linkers make when they resolve symbol references can silently affect the correctness of your programs. Programs that incorrectly define multiple global variables pass through the linker without any warnings in the default case. The resulting programs can exhibit baffling run-time behavior and are extremely difficult to debug. We will show you how this happens and how to avoid it.

3. Understanding linking will help you understand how language scoping rules are implemented.For example, what is the difference between global and local variables ?

4. Understanding linking will help you understand other important systems concepts. The executable object files produced by linkers play key roles in important systems functions such as loading and running programs, virtual memory,paging, and memory mapping.

5. Understanding linking will enable you to exploit shared libraries. For many years, linking was considered to be fairly straightforward and uninteresting. However, with the increased importance of shared libraries and dynamic linking in modern operating systems, linking is a sophisticated process that provides the knowledgeable programmer with significant power. For example, many software products use shared

libraries to upgrade shrink-wrapped binaries at run time. Also, most Web servers rely on dynamic linking of shared libraries to serve dynamic content.

1.5 Reason for designing built in code review utilities:

As mentioned above the main purpose of this project is to make users aware of complex functionality of system softwares that they are using. Hence for that reason we have embedded several built in hooks into current code base of *Xv6* which gives user the step by step information of internal work flow of any given utility like linker, loader or interpreter. This enables her to track down the actual source of that utility very easily and modify it accordingly. This object code review utilities are very much useful for making best use of assignment section of this report.

Chapter 2

Literature Survey

2.1 Current educational purpose operating systems

We have studied some of the current available education purpose operating system OSes like *XINU*,*sos*,*linux kernel v0.01*,*Bluefire OS*,*MINIX*,*jos*, *Plan 9*. While studying all these operating systems we have followed guidelines based on following parameters:

1. language code should be *ANSI C*.
2. should not have more that 15000 lines of code.
3. should follow one of the measure unix standard specifications.
4. should be open source and able to compile on `qemu` emulator in major linux distributions.
5. should have past history of being used for pedagogical purposes.
6. should have well documented manual.

A comprehensive study of each of these operating system is presented below.

2.1.1 *XINU*

XINU [10] stands for Xinu Is Not Unix. Actually it shares concepts and even names with *Unix*, but the internal design differs completely. Xinu is a small, elegant operating system that supports dynamic process creation, dynamic memory allocation, network communication, local and remote file systems, a shell and device-independent I/O functions. As *XINU* does not follow any UNIX standards, it falls apart from mainstream of operating system pedology.

2.1.2 *SOS*

SOS [9] Simple Operating System, is an operating system kernel which aims at being simple to understand and that nonetheless covers concepts and functionalities of modern OSes. The OS code comes with a batch of articles published in the french magazine called Linux Magazine France, during 2004 and 2006. The main problem with SOS is most part of documentation is available in french and most of concepts are implemented in a standalone fashion, apart from standard *UNIX* prototype.

2.1.3 *linux-0.01*

[4] This is earliest version of *Linux kernel* release in early 1991. It comprise of 8000 lines of code only. This can be the best tool for novice developers to learn *Linux kernel* because current code base has evolved from this tiny release v0.01. But the problem with this vanilla kernel is dependency and compiling environment has been changed significantly since 1991. So this linux-0.01 version can not be complied on today's modern compilers. Even though it was written considering *GCC*, but *GCC* itself has changed a lot since 1991. This difficulties make linux-0.01 unsuitable for use of pedagogy.

2.1.4 *Bluefire OS*

Bluefire[1] is a didactic OS that was created to show every step of creating a bootstrapping OS. It's approximately 15,000 lines of code. It's include primary graphics which is been shown while booting steps. This operating system lacks of some core functionalities like synchronization, standard memory management. The design of this operating system is quiet different from standard *UNIX* prototypes.

2.1.5 *MINIX*

MINIX [5]is a free, open-source, operating system designed to be highly reliable, flexible, and secure. It is based on a tiny microkernel running in kernel mode with the rest of the operating system running as a collection of isolated, protected, processes in user mode. *MINIX* incorporates `microkernel` architecture which is difficult to understand for novice students. Also current *MINIX* version is now become a high end operating system and it's no more a pedagogical operating system now.

2.1.6 *plan 9*

Plan 9 [8] is a research system developed at Bell Labs starting in the late 1980s. Its original designers and authors were Ken Thompson, Rob Pike, Dave Presotto, and Phil Winterbottom. *Plan 9* is an operating system kernel but also a collection of accompanying software. The bulk of the software is predominantly new, written for Plan 9 rather than ported from Unix or other systems. The main problem with this operating system, as it has been evolved since 1980s, it has become a full fledge operating system instead of just a research system.

2.1.7 *xv6*

xv6 is a simple unix-like teaching operating system developed at MIT [13]. xv6 is re-implementation of Dennis Ritchies and Ken Thompsons Unix Version 6 (v6). xv6 is implemented in ANSI C for an x86 based multiprocessors. *Xv6* is simple enough to teach operating system walking through its code in a semester. xv6 has implemented most of modern operating system core functionalists which makes easier to start look through in oper- ating system code for a newbie. Xv6 is around 8000 lines code yet still contains the important concepts and organization of Unix. Current users of xv6 include MIT[6],Yale [11],Columbia university, IIT Delhi.

2.2 Components of prototype operating systems

The selected operating system should have one language process system process either interpreter or compiler. It should also have a minimal library on top of which utilities can run. We have studied such exiting solutions.

2.2.1 Introduction to *Picoc*

PicoC [7]is a very small C `interpreter` for scripting. It was originally written as the script language for a UAV's on-board flight system. It's also very suitable for other robotic, embedded and non-embedded applications. The core C source code is around 4500 lines of code. It's not intended to be a complete implementation of *ISO C* but it has all the essentials. When compiled it only takes very little code space and is also very sparing of data space. This means it can work well in small embedded devices. It's also a fun example of how to create a very small language implementation while still keeping the code readable. The overall structure and it's open source nature is the key reason to choose Picoc as a prototype interpreter in current xv6 os.

2.2.2 Introduction to *dietlibc*

Dietlibc [2]is a C standard library released under the GNU General Public License Version 2, but there are also commercial licences available. It was developed with the help of about 100 volunteers by Felix von Leitner with the goal to compile and link programs to the smallest possible size. *Dietlibc* was developed from scratch and thus only implements the most important and commonly used functions. It is mainly used in embedded devices. Dietlibc is easy to build on linux operating systems and have quite flexible licensing policy which makes it the best choice for using this as a standard library in current xv6 kernel.

Chapter 3

Design

As briefly explained in first 2 chapters that the main problem with today's production level system softwares is that their core logic is most of the time hard to grasp for student and novice developers because of amount of error checking code that is added into them. Though various hooks and verbose output options are embedded into them, their main intention is to help developers to debug new source code added into the current codebase rather than learning the original one. So while establishing a framework for improved learning experience of operating system and system software courses, we need to focus on the various stages through which these softwares pass through in order to generate final output. In case of operating system understanding such verbose output helps in understanding various modular parts of operating systems. So that if some new driver or let's say some new functionality is to be added into current production level software then this comprehensive knowledge of different modular part will help them in targeting source code of that particular subsystem.

This following section will provide list of such utilities or tools that we have included in xv6 considering the general difficulties faced by students. This list will particularly emphasize on those concepts that are traditionally missed out from most of the operating system textbooks but are of great importance.

3.1 Design of platform for *Picoc* interpreter

3.1.1 Workflow of *Picoc*

The basic outline of how Picoc interprets the code can be described briefly as follows:

1. It first Allocates enough space on heap for loading given program in memory.
2. It then tokenizes the given source code into different categories such as variables, function declarations, function definitions etc.
3. It then parses the given source code as per the language grammar to find out if there are any early stage syntax errors.
4. While parsing the source code it parses one statement at a time, and identifies different types of tokens in given statement.It also associates proper semantic meaning to those tokens.
5. If while parsing a given statement, it is found that a call to external function is made then definition of that function is first searched in local as well as global list of built-in function libraries.
6. Once that function definition is found, a wrapper for that particular library function is called. After some prepossessing, a standard library call inside that wrapper then does the rest of work and returns back. Hence control of flow goes back to interpreter parser engine again.

For example, if a call to `printf` is there in the source of given program then in first step *Picoc* tokenizes it as `Cprintf`. In second step it is associated with semantic meaning of function call to it. In third step actual definitions of `Cprintf` is searched in an list of by default included library files. Here in this case this is in stdio.c. So a call to `Cprintf` function from stdio.c will be replaced in place of `printf`. The implementation of `Cprintf` will then make a call to standard library version of `Printf`.

3.1.2 Adding library support for *Picoc*

Any intermediate stages of interpreter processing involves many crucial functionalities such as tokenization, symbol generation, semantic linking etc. These all functionalities make use of many of the standard string processing functions such as `strtok`, `strlen`, `strncmp` etc. But currently when we build *xv6* kernel these functions are not there in it's standard `C library`. Hence firstly we have implemented all of these functions specific to xv6.

3.1.3 Building *Picoc* interpreter in *xv6*

Right now *Picoc* can only be build using *GCC* compiler bundle. As *xv6* doesn't have that right now we have used following approach for building this interpreter.

1. First *Picoc* is built by giving `-C` option to *GCC* in standard *Linux* environment. This option creates relocatable object file of final executable.

2. This relocatable executable of *Picoc* is then linked with the standard library of *xv6*. So this arrangement makes sure that all the relocatable files are produced by *GCC* but are finally linked against library provided in xv6.

3. While linking these relocatable files against *xv6's* standard library all the function definitions that are required or functions which are called by *Picoc* but whose definition is not found in current library have to be added to current library.

3.2 Design of Linkage editor for xv6

During designing a built-in linker there is one crucial question to be tackled is of deciding type of linker to be built. There are two main types of linkers that are static and dynamic linkers.

3.2.1 static linking and dynamic linking

In static linking, linker copies all of the library routines used in the program in executable image. This requires more memory space for every program than dynamic linking. But it is to accommodate and doesn't require extensive kernel support while loading a program in memory. It also doesn't require run-time library version to be available in memory all the time during running of program.

Whereas in dynamic linking only name of shared library is placed in executable image. Actual linking with the library routines is not performed at compile time. All such shared libraries are placed in memory at some predefined address space. So when the program is run it can refer to those routines even though they are not part of it's final executable. [3]

Though dynamic linking is more advantageous and is the modern day de-facto standard, we have designed static linker for xv6. This choice is made so that original kernel code of xv6 remains simple and easy to read. Also another aspect of dynamic linking is that it requires some complex memory management and process management schemes that will make allow two or more processes two share same piece of common code. So by abiding to basic principle of this project which aims at readability of source code we have opted for static linker in this version of xv6. This is done so that student will easily get hands on experience of tweaking source code of linker without much hassle. Refer assignment sections.

3.2.2 Algorithm for symbol resolution used in linker

During the symbol resolution phase, the linker scans the relocatable object files from left to right in the same sequential order that they appear on the compiler drivers command line. (The driver automatically translates any .c files on the command line into .o files.) During this scan, the linker maintains a set E of relocatable object files that will be merged to form the executable, a set U of unresolved symbols (i.e., symbols referred to, but not yet defined), and a set D of symbols that have been defined in previous input files. Initially, E, U , and D are empty.

1. For each input file f on the command line, the linker determines if f is an object file or an archive. If f is an object file, the linker adds f to E, updates U and D to reflect the symbol definitions and references in f , and proceeds to the next input file.

2. If f is an archive, the linker attempts to match the unresolved symbols in U against the symbols defined by the members of the archive. If some archive member, m, defines a symbol that resolves a reference in U , then m is added to E, and the linker updates U and D to reflect the symbol definitions and references in m. This process iterates over the member object files in the archive until a fixed point is reached where U and D no longer change. At this point, any member object files not contained in E are simply discarded and the linker proceeds to the next input file.

3. If U is nonempty when the linker finishes scanning the input files on the command line, it prints an error and terminates. Otherwise, it merges and relocates the object files in E to build the output executable file.

3.2.3 Algorithm used for relocation in linker

Compilers and assemblers generate code and data sections that start at address 0. The linker relocates these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location. Relocation consists of two steps:

Relocating sections and symbol definitions

In this step, the linker merges all sections of the same type into a new aggregate section of the same type. For example, the .data sections from the input modules are all merged into one section that will become the .data section for the output executable object file. The linker then assigns run-time memory addresses to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules. When this step is complete, every instruction and global variable in the program has a unique run-time memory address.

Relocating symbol references within sections

In this step, the linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses. To perform this step, the linker relies on data structures in the relocatable object modules known as relocation entries, which we describe next.

relocation entries

When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate location is unknown, it generates a relocation entry that tells the linker how to modify the reference when it merges the object file into an executable.

relocation types

ELF defines 11 different relocation types, some quite arcane. We are con- cerned with only the two most basic relocation types

R_386_PC32:

Relocate a reference that uses a 32-bit PC-relative address. Recall from Section 3.6.3 that a PC-relative address is an offset from the current run-time value of the program counter (PC). When the CPU executes an instruction using PC-relative addressing, it forms the effective address (e.g., the target of the call instruction) by adding the 32-bit value encoded in the instruction to the current run-time value of the PC, which is always the address of the next instruction in memory.

R_386_32: Relocate a reference that uses a 32-bit absolute address. With absolute addressing, the CPU directly uses the 32-bit value encoded in the instruction as the effective address, without further modifications.

```
foreach section s {
foreach relocation entry r {
  /* ptr to reference to be relocated */
  ptr_for_modify = s_section_start + r.offset;

  /* Relocate a PC relative reference */
  if (r.type == R_386_PC32){
    /* ref s run-time address */
    refaddr = ADDR(s_section_start) + refaddr;
    *ptr_for_modify = (ADDR(reloc_entry_r_symbol) +
        (*ptr_for_modify - ptr_for_modify));
  }
  /* Relocate an absolute reference */
   if (relocation_entry.type == R_386_32)
      *ptr_for_modify = (ADDR(reloc_entry_r.symbol) + *ptr_for_modify);
}
}
```

3.3 Filesystem modifications

Originally xv6 operating system would used to support at max 64 KB of file size. In order to port new linker and other system utilities to it we need to extend the filesize limit of this structure. Following section will give an overview of underlying filesystem structure along with the required changed that are to be done in order to maximize max file size to 8MB.

3.3.1 Original xv6 structure

The file system must have a plan for where it stores inodes and content blocks on the disk. To do so, xv6 divides the disk into several sections. The file system does not use block 0 (it holds the boot sector). Block 1 is called the superblock; it contains metadata about the file system (the file system size in blocks,the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks in use. Most of the remaining blocks are data blocks, which hold file and directory contents. The blocks at the end of the disk hold a log that is part of the transaction layer. As a result the max size of any file on this filesystem can't execed 70kB. This can be given by below formula:

max file size = [`NDIRECT` * `BSIZE`] + [`NINDIRECT` * `BSIZE`]

which becomes 6KB + 64KB = 70 kB.

where `NDIRECT` is the number of direct blocks which is 12. `BSIZE` is size of each Block on disk 512 Bytes. `NINDIRECT` is number of indirect blocks which is currently 512/4 i.e. 128. The original file system is shown in figure 3.1

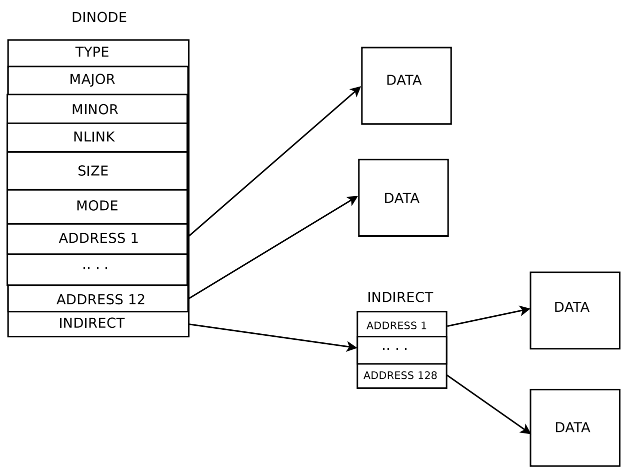


Figure 3.1: Original file system

3.3.2 Changes made to increase max file size limit

The size of max file size of every inode can be increased by adding more number of `NINDIRECT` blocks to current implementation of filesystem. So as to increase max file size to 8MB we added 1 double indirect block in the inode's on disk structure. Details of code base changes are discussed in implementation section. The double indirect block is block whose every entry represents a block address of another block. Each entry of every such block represents the block address where actual data is stored. So this gives more space to every inode.

But to maintain the overall memory size of inode structure we have cut down 1 NDIRECT block from disc inode content. So that max filesize in modified filesystem can be given as:

max file size = [`NINDIRECT * BSIZE` + `NDINDIRECT * NINDIRECT * BSIZE`]

which become 8 MB + 64KB   8MB.

The modified file system is shown in figure 3.2

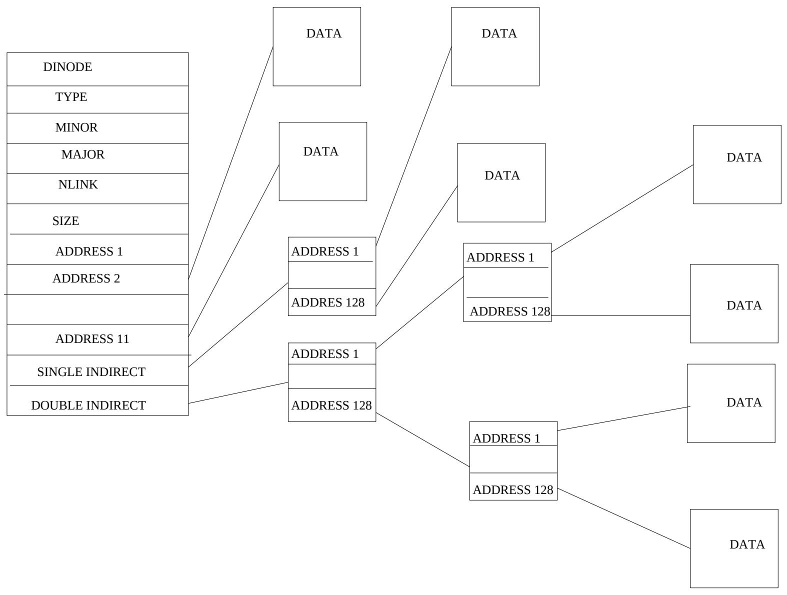


Figure 3.2: Modified file system

3.4 Standard library in C

The current version of *xv6* uses a static library. All of the library functions are made part of kernel. It doesn't make use of library archives. If an application wants make use of these library routines then it has to be built by using a shell script given in Appendix B. There are two options while adding these definitions in current library. First is to implement these functions by own or second one is to use some already existing `POSIX` standard library. Here in this case we have used a library named *dietlibc*. List of functions that are ported to current library of *xv6* which are borrowed from *dietlibc* can be found Appendix A.

Chapter 4

Implementation

4.1 Porting *Picoc* to *xv6*

4.1.1 New functions added to port *Picoc* to *xv6*

Once *Picoc* is invoked it reads an input file through `FILE` structure mechanism. This function of reading original source file is done by `PicocPlatformReadFile()`. Similarly to generate intermediate files or final output file it makes an extensive use of functions such as `fprintf(), sprintf(), fputs()`.As it makes use of these functions such as `fopen(), fread(), fclose()` we have implemented these functions in current *xv6* library. As mentioned above *Picoc* can handle the various data types such as `INT,FLOAT,CHAR * etc`. So to perform arithmetic operations on data types such as `float` and `double` it needs to make use of complex exponential operations. For example, while printing `float` datatype to standard output it calls `PrintFP()` which uses two mathematical functions `Pow() and exp()`. Hence these two functions are also ported to current version of *Xv6*. Their implementations are borrowed from *dietlibc*.

4.1.2 Library of Picoc

Considering the general work flow of Picoc as explained in chapter 3. For providing appropriate environment to Picoc we have provided complementing sets of library functions whose wrappers are supported by *Picoc* during run time. In picoc currently we have just provided limited API that includes following functions:

1. printf
2. scanf
3. getchar
4. putchar
5. strcmp
6. strlen
7. strlen

along with the support to standard data types such as int,double,float,char etc. When an instance of picoc is running it shows output at different stages of interpretation like symbol table generation, tokenization and library call wrappers. The main reason to limit the library API to such small scale so that it will create a scope for students to understand the current structure and add further functionality into it.

4.2 Increasing Max filesize limit of filesystem in xv6

4.2.1 Original block allocator in *xv6* filesystem

File and directory content is stored in disk blocks, which must be allocated from a free pool. xv6s block allocator maintains a free bitmap on disk, with one bit per block. A zero bit indicates that the corresponding block is free; a one bit indicates that it is in use. The bits corresponding to the boot sector, superblock, inode blocks, and bitmap blocks are always set [12]. The block allocator provides two functions: The first one is `balloc` it allocates a new disk block, and second one is `bfree` which frees a block. `Balloc` starts by calling `readsb` to read the superblock from the disk (or buffer cache) into sb. `balloc` decides which blocks hold the data block free bitmap by calculating how many blocks are consumed by the boot sector, the superblock, and the inodes (using BBLOCK). The loop considers every block, starting at block 0 up to sb.size, the number of blocks in the file system. It looks for a block whose bitmap bit is zero, indicating that it is free. If balloc finds such a block, it updates the bitmap and returns the block. The race that might occur if two processes try to allocate a block at the same time is prevented by the fact that the buffer cache only lets one process use a block at a time. `Bfree` finds the right bitmap block and clears the right bit.

4.2.2 changes made to on-disk Inode structure

The original on-disk inode structure, struct dinode, contains a size and an array of block numbers. The inode data is found in the blocks listed in the dinodes addrs array. The first NDIRECT blocks of data are listed in the first NDIRECT entries in the array; these blocks are called direct blocks. The next NINDIRECT blocks of data are listed not in the inode but in a data block called the indirect block. The last entry in the addrs array gives the address of the indirect block. Thus the first 6 kB (NDIRECTBSIZE) bytes of a file can be loaded from blocks listed in the inode, while the next 64kB (NINDIRECTBSIZE) bytes can only be loaded after direct blocks are completely filled.

In extension to this we have added one more NDINDIRECT block to current inode structure. Each entry in this block represents block address of another block. Each entry of such blocks gives address of another block which actually stores data. Now the main task is to allocate blocks from this entry. This is handled by modified block allocation strategy explained in following section. See Appendix F for further detail.

4.2.3 Modified block allocation strategy

The function bmap manages the representation. So that higher-level routines such as readi and writei can interact with in memory inode structure. Bmap returns the disk block number of the n'th data block for the inode ip. If ip does not have such a block yet, bmap allocates one. The function bmap begins by picking off the easy case: the first NDIRECT blocks are listed in the inode itself. The next NINDIRECT blocks are listed in the indirect block at ip.addrs[NDIRECT]. Bmap reads the indirect block and then reads a block number from the right position within the block. If the block number exceeds NDIRECT+NINDIRECT, because of new changes in filesystem now it will try to find that data block in ip.addr[NDINDIRECT]. Hence it gives filesystem wider length. See Appendix F for inode structure.

4.3 Linkage editor

4.3.1 Implementation of static linker in xv6

Static linkers such as the Unix ld program take as input a collection of relocatable object files as a command-line arguments and generate as output a fully linked executable object file that can be loaded and run. The input relocatable object files consist of various code and data sections. Instructions are in one section, initialized global variables are in another section, and uninitialized variables are in yet another section. A linker concatenates blocks together, decides on run-time locations for the concatenated blocks, and modifies various locations within the code and data blocks. Linkers have minimal understanding of the target machine. The compilers and assemblers that generate the object files have already done most of the work.

Implementation of new linker in *xv6*

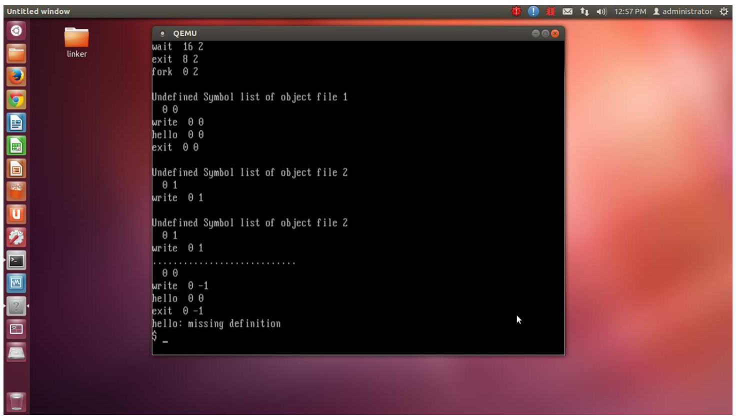
The newly modified linker first reads the given object files and for each object it generates two lists, one is of defined symbols and other is of undefined entries using `get_undefined_entries()`. These lists are then processed subsequently in further stages such as symbol resolution and relocation.

4.3.2 Symbol resolution

For Symbol resolution, symbol references are resolved from each pair of object files. `resolve_undefined_reference()` takes up as an argument a list of defined symbols of some object file and another list of undefined symbols of any other object file to be linked together. It then finds out if the entry from undefined list is present in the defined list if so, it moves that symbol from list of undefined entries to defined. If such undefined entry is resolved then it sets the `argindex` flag of that symbol to -1. Finally after all the pairs of files are resolved, `definition_not_found()` is called. This function traverses undefined list of every object file and checks if the `argindex` flag of all of these symbols is set to -1. If an entry is found whose symbol entry doesn't have `argindex` set to -1. It implies that some symbol reference is not resolved and prints out appropriate error message. See Appendix D for code of these functions and symbol structure definition.

4.3.3 Relocation

Before relocation, linker calculates total size of resultant binary along with individual size of data section and text section. Then `do_relocation()` takes an argument each object file and it's text section, data section and their respective position offset in final executable. This is calculated by taking into consideration modulo 32 byte order. That is for second object file in the list it's text section would start exactly at the address



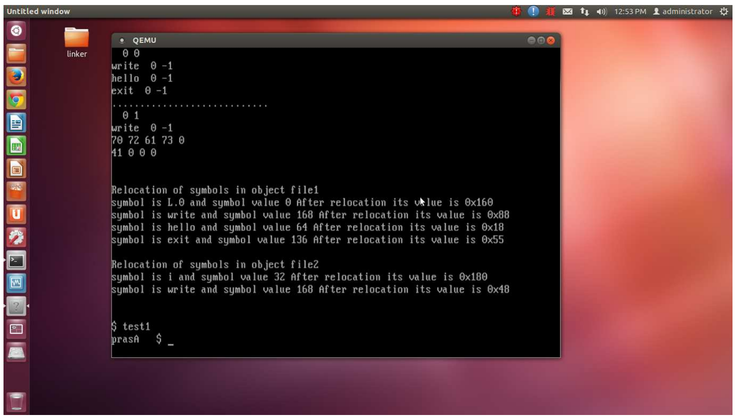
```
wait  16 2
exit  8 2
fork  0 2

Undefined Symbol list of object file 1
  0 0
write  0 0
hello  0 0
exit  0 0

Undefined Symbol list of object file 2
  0 1
write  0 1

Undefined Symbol list of object file 2
  0 1
write  0 1
............................
  0 0
write  0 -1
hello  0 0
exit  0 -1
hello: missing definition
$ _
```

Figure 4.1: Linker output for missing symbol test case



```
  0 0
write  0 -1
hello  0 -1
exit  0 -1
............................
  0 1
write  0 -1
70 72 61 73 0
41 0 0 0


Relocation of symbols in object file1
symbol is L.0 and symbol value 0 After relocation its value is 0x160
symbol is write and symbol value 168 After relocation its value is 0x88
symbol is hello and symbol value 64 After relocation its value is 0x18
symbol is exit and symbol value 136 After relocation its value is 0x55

Relocation of symbols in object file2
symbol is i and symbol value 32 After relocation its value is 0x180
symbol is write and symbol value 168 After relocation its value is 0x48


$ test1
prasA   $ _
```

Figure 4.2: Linker output for positive test case

modulo 32 which is next to after end of text section of first object file. Then it reads relocation table of that object file and finds out it's entry in corresponding list of defined symbols. Then as mentioned in the design section it identifies the type of symbol i.e either `R_386_32` or `R_386_PC32`. Depending upon it's type it calculates the new offset of the corresponding symbol and puts that value at the address given by that symbol's relocation table entry.

Finally all the sections of different object files are merged together and value of `entry point` is set in `Program header`. Also total filesize of final executable is updated.

Final working output of linker is shown in figures 4.1 and 4.2

Chapter 5

Assignments for Operating system course

This chapter discusses a series of assignments that can be incorporated into operating system coursework. Educators making use of these assignments are expected to use modified xv6 kernel specifically developed for this project. All the assignments are designed in such a way that it will help students to get more insight into critical modules of operating system.

5.1 Assignment 1: Adding library function with corresponding system call

5.1.1 Problem Statement

Adding a new library function along with its corresponding mapping system call. The current *xv6* kernel doesn't have support for position seeking in given file. Students are expected to add this functionality by adding corresponding system call and library interface function for the same.

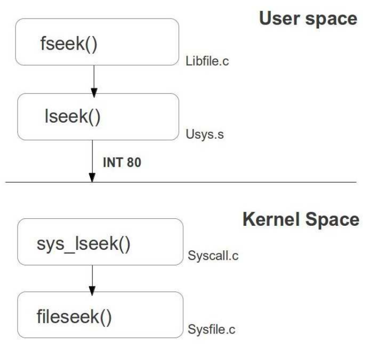


Figure 5.1: Assignment 3:Solution

5.1.2 Aim of assignment

This assignments will help students in understanding working flow of library functions and their mapping with system call underlying in kernel space.

5.1.3 Solution

The current xv6 lacks of `fseek` library function. Students are expected to add fseek library function with its wrappers functions in user space as well as corresponding system call in kernel space. The implementation of the code will be as shown in the figure.

Implementing `fseek` library function

This library function can be implemented by writing a wrapper function in library which internally calls to `lseek()`. There might be some processing code or error checking that can be done itself in `fseek()` at user space level. `lseek()` is also a part of library which causes software interrupt `int 0x80`, result of which it lands into kernel space. For this implementation we haven't done much in user space level, but in real world library functions there is lots of code divided in multiple wrapper functions.

Implementing `lseek` system call

For adding such functionality in kernel, we need to add a system call which will handle `fseek` call. A specific number has to be assigned to system call, which is expected to pass in `eax` which calling interrupt. The control reaches to same system call whose number had been passed in register. For that `sys_lseek()` function is implemented in kernel space. It again calls to `fileseek` function internally which actually seeks the required position in the file.

5.1.4 Code

```
 int fseek(FILE *stream, int off, int whence){
  return lseek(stream->desc, off, whence);
 }
globl lseek;
lseek:
  movl $SYS_ ##name, %eax;
  int 0x80;
  ret
int sys_lseek(void){
  struct file *f;
  int n,mode;
  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argint(3, &mode) < 0)
    return -1;
  return fileseek(f, n, mode);
}
int
fileseek(struct file *f, int off, int mode)
{
    if(f->readable == 0)
            return -1;
    if(f->type == FD_PIPE)
              return -1;
    if(f->type == FD_INODE){
        if(mode == 0 && off < f->ip->size)
```

```
            f->off = off;
        else if(mode == 1 && (f->off + off) < f->ip->size)
                f->off = f->off + off;
        else if(mode == 2)
            ;// Not implemented
        else
        return -1;
        return off;
    }
            return -1;
}
```

5.2 Assignment 2: Scheduling

5.2.1 Problem Statement:

Scheduling policy is backbone of the modern day pre-emptive kernels. This assignment involves using different scheduling policy for the Xv6 kernel other than currently used Round Robin scheduling algorithm.

5.2.2 Aim of assignment:

To introduce students to different scheduling policy and evaluate their performances.

5.2.3 Solution:

There are variety of scheduling policy discussed in Operating system literature. Students are first advised to go through them and then implement some of them as per guidelines mentioned below.

Round Robin

This is the default policy used in current Xv6 operating system. But currently xv6 schedules process after every clock tick. This can be changed by setting up some pre-processor variable while building up the kernel. So the trap mechanism will invoke scheduler only after those many clock ticks. That is kernel should provide functionality so that Xv6 can use different time quantum for round Robin policy. After this students are expected to perform different experiments mentioned below in evaluation section.

Two Queues Scheduling

This scheduling policy is based on concept of *priority scheduling*. In such scheduling mechanism while creating process proc struct will have an added field which will mention the priority of that process. In general any application level process will have high priority but most of the *UNIX* implementation provide `nice` system call which allows user to lower the priority of given process.

For designing of actual scheduler for such scheme, one of queues will hold all the high priority processes and other will hold all the lower priority processes. Then scheduler should be invoked after some predefined time quantum as mentioned above. On each invocation scheduler will schedule the process with RUNNABLE state from high priority queue. But it will schedule the process from other lower priority queue on alternate invocation of scheduler only. Such priority scheduling mechanism has to ensure that there are on no high priority process with RUNNABLE state before scheduling any process from low priority queue.
This can be easily achieved by using static flag variable in scheduler function in current xv6 code.

Two Queues guaranteed fair scheduling

This is an advanced version of *Two Queues Scheduling policy* whose major drawback is that, in some scenarios it may starve low priority processes forever. So remedy for this solution is to use fair scheduling policy which calculates the ratio given below for every process:

$$\frac{rtime}{current\_time - ctime} \tag{5.1}$$

and schedules the process with lowest priority.

Here `rtime` is running time of process and `ctime` is creation time of process. These time measures are nothing but the number of clock ticks which can be calculated by adding and extra time field in `proc struct` and updating it's value for every clock tick.

5.2.4 Evaluation of different Scheduling schemes:

For evaluating different scheduling policy use `-D` option of gcc compiler while building xv6. Then by making use of conditional pre-processing constructs such as `ifdef` various scheduling schemes can be used.

For stress measurement of different policy write a user level program as test case. This program will fork 25 child process upon execution. Every child process will have unique `pid`. After that all the process which have odd numbered `process id` will lower their priority immediately.

Every child process will have code that prints it's `pid` 200 times and then calls `exit`. Till then parent process does wait. But the `wait` system call should be modified so that it can return the time for which it is waiting and the total time for which the child process was running. Depending upon these measures various inferences can be drawn.

5.3 Assignment 3: Implementing `semaphores` in xv6

5.3.1 Problem Statement

The currently xv6 uses concept of `spinlock` for synchronizing the process. This assignment aims at implementing `semaphore` using `xchg` hardware instruction.

5.3.2 Aim of assignment

The students are expected to implement multiple instance synchronization using `semaphores.`

5.3.3 Solution

The `semaphore` can be implemented with the help of structure `struct semaphore` which has two variables `value` and `lock.` Init function initializes `semaphore` variable to specified value i.e the number of instances available of that resource. `P()` and `V()` also called as up and down functions respectively performs action on `semaphore` to acquire and release the resource. The updation of `val` variable has to be atomic. Multiple process trying to get the semaphore should not enter into the critical section code that updates value of `val`. So the code in the `P()` and `V()` functions is critical and it has to be protected during concurrency multiple process. These critical sections are protected by hardware instruction `xchg` which guarantees atomic operation of the instruction. Hence, here mutual exclusion is implemented using `xchg`. Just for sake of simplicity instead of maintaining the waiting queue which sent for sleep and then woke up, the process just allowed to spin while acquiring a lock.

5.3.4 Code

```
struct semaphore {
  int val;
  int lock;
```

```
};
init(struct semaphore *s, int val) {
  s->lock = 0;
  s->val = val;
}
P(struct semaphore *s) {
  while(1) {
    while (xchg(1, &s->lock) == 1); //implement acquire spinlock using xchg
      if (s->val > 0) {
        s->val--;
        break;
      xchg(0, &s->lock); //release spinlock
      }
}
V(struct semaphore *s) {
  while (xchg(1, &s->lock) == 1); //implement acquire spinlock using xchg
    s->val++;
    xchg(0, &s->lock);
}
```

5.4 Assignment 4: Process Memory Swap

5.4.1 Problem statement:

Current memory map of Xv6 allows only one process to be resident in RAM at a time. That is whole physical memory is allotted to only single process. But instead of this most modern day operating system uses a concept of swapping memory architecture. So this assignment includes implementing memory swapping mechanism in Xv6.

5.4.2 Aim of assignment:

Aim of this assignment is to introduce students to process memory map, demand paging, swapper mechanism, swap filesystem.

5.4.3 Solution:

To implement swapping mechanism, one has to create a swapper process. It is a dedicated process that actually selects which process to swap out,in and when to do it so. This is universal process like init or scheduler, but the care must be taken that this process starts running before any other process gets created. Hence swapper process must be create before scheduler process.

Once the swapper process is created, it executes following general algorithm for swap out mechanism:

1. Select the process to swapped out and when to swap it out.

2. Find it's process `struct` entry and then create a swap file with same name as that of it's process id. After that write every page of that process from it's page table entry to corresponding swap file.

3. Unmap the pagetable entry of that process and set it's state as SWAPPED OUT in it's process struct entry.

Once the swapper process is created, it executes following general algorithm for swap in mechanism:

1. After the pagefault is generated, a trap mechanism should invoke the swapper process, providing `pid` of the process to be executed.

2. Swapper process should find the relevant swap file from disk.

3. Create a new page table and map all the pages from the swap file to the pages from

this page table.

4. Remove the swap file.

5.4.4 Design of Swapper process

The important task in designing the swapper process is to decide which process to be swapped out or swap in and when to do it. For this task a preliminary approach would be periodic memory check of main memory usages by all the process.

Triggering a swapper

This approach involves periodically checking ratio of free pages to the allocated pages in main memory. If this ratio falls below certain threshold level then swapper should invoke a function to swap out some process from main memory to swap file system.

Selecting a process for swapping out

The process that is currently in memory and has it's state as RUNNABLE can only be swapped out. The decision of which process to be swapped out can be made depending upon some criteria such as process with minimum no of used pages, Or process with maximum number of used pages.

Selecting a process for swapping in

This is very much a trivial choice. As this will depend upon which process has caused page fault. That is while executing certain process, kernel's loader will notify which desired process is not currently present in memory and hence a hardware trap is generated. So a trap mechanism should then find out that process from swap file system and execute the algorithm for swapping in that process.

5.5 Assignment 5: Modification to linker

5.5.1 Problem Statement (scope resolution)

Current version of linker provided with *xv6* doesn't take into consideration variable binding of different variable. This assignment includes adding scope resolution capability to the linker which differentiates between local and global variables of program so as to resolve conflict between variables having same names but different bindings in two different object files.

5.5.2 Aim of assignment:

This assignment will help students understand how variable is actually store in memory of program depending upon it's binding such as global, local, static etc. And how that variable is resolved during linking with other object files.

5.5.3 Solution:

In the current version of linker, for every given object file `resolve_undefined_reference()` finds out if the undefined entry in that object file is found in list of defined objects from any other object file. But it doesn't take into consideration whether required defined object from any other file is declared locally or globally. If it is defined locally then it can't be used as external reference. So for solving this problem one has to check the binding of particular variable while resolving references. That is the particular undefined entry is resolved only if it defined globally in any other provided object file. To find the type binding of given variable make use of `ELF32_ST_BIND()` macro.

5.5.4 Problem Statement (order of object file argument)

Current algorithm of *xv6* can handle function call references only from files which are provided as argument to linker program after the file from which functions are called. That is object file given as `argv[1]` can only call functions defined in `argv[2]` or greater than that. Here students are expected to modify current implementation so as to allow more than 3 files to be linked together and every other file can call function from any other file irrespective of it's order.

5.5.5 Aim of assignment

To understand the concept of Mathematical set theory in traditional computer science. Current linker uses a 'list' data structure which actually stores the list of defined and undefined objects for every given object file argument. This implementation has a drawback that it can only link at max 3 files together. For fixing this problem student will have to design a new data structure for practical purpose which will be a big confidence booster and problem solving exercise for them.

5.5.6 Solution

The solution to this problem is to use a Set data structure. Rather than using individual lists of defined and undefined symbols for every object file, one should define global Set of defined symbol named `D` and another set of undefined symbol `U`. Then for each input object file, linker should read the symbol table entries of it and accordingly add defined and undefined symbol to `D` and `U` respectively. If it finds any undefined symbol any one of the object file and that symbol is present in set `D` then it should delete that entry from set `U`.This process of scanning each file should be repeated until `U` and `D` doesn't change even after scanning entire set of object files. This mechanism ensures that even if some function is present in `U` set if first traversal and it's definition is in files before it on the command line, it will be resolved during next traversal.

After this stage check out if set `U` is empty or not. If it is empty then that means all the undefined symbol entries have found their respective references. If it is not empty then that means there are some undefined symbol entries in some object file for which linker can't find it's definition or initialization.

5.6 Assignment 6: File system modification

5.6.1 Problem Statement:

The assignment is to speed up the current filesystem transaction.

5.6.2 Aim of assignment:

This assignment will give more deeper insight into the current design of xv6 filesystem. Some of the modern day filesystem concepts such as journaling, memory mapped filesystem, filesystem cache can be understood more easily. This assignment will be a good exercise of evaluating particular operating system module and then enhancing it's performance.

5.6.3 Solution:

Speed of retrieving content from filesystem depends upon how much data is retrieved in one I/O disk access. For current *xv6* filesystem each block is of size 1KB. So during one I/0 cycle it can only fetch 1KB of data from disk. So a solution to this is to increase a size of block to 4KB. So this involves changing the size of `uchar` data in `buf struct` to 4096.

Another drawback of `xv6` filesystem is it's block allocation policy. Currently while creating any file or directory, new blocks are allocated from the list of free blocks on disks. This poses problem of data locality. For instance, if there are 10 directories created in a filesystem and then new file is created in first out of them. Then every time that file is accessed filesystem will have to seek all the blocks in between them. So one way to tackle this problem is to maintain locality of `inodes` of every file in it's parent directory. To do so, one can allocate group of free blocks near directory `inode` whenever new directory is created. So when new file is added to that directory, free blocks are allocated from those blocks. This ensures lower seek time.

Chapter 6

Conclusions and Future work

As this report examines in detail about different aspects of developing a prototype operating system that can help in improving learning experience of individuals enormously. Apart from this it can be concluded that such efforts will help in achieving two other prime objectives. 1. It gives a deeper insight into the internal working of system softwares to the application level developer. So that in future he can write more efficient user level programs leveraging complete features of underlying OS and architecture. 2. Such prototypes can prove as a prominent tools for people developing an alternative solutions to current bottlenecks. These prototypes can be used as a sandbox for developing alternate mechanisms and then taking them to the production level software for further evaluation. This approach will help system level developers to make sure that their alternative solution is not against the basic fundamentals of operating system functioning which may happen when one directly tries to replace old mechanism in current codebase.

6.1 Framework for porting system softwares to fresh kernel

Though main objective of this project is to develop an educational purpose operating system for graduate/undergraduate course, it's very important to actually devise a framework that can be used in future to develop such prototypes for variety of different platforms such as `android` operating system, real time operating system.

As a part of this project we have outlined following some guidelines for developing such prototypes in variety of domains:

1. First step in developing such prototypes is that it should be easy to compile and use, comprising only one central Makefile.

2. While developing such prototypes it should be ensured that most of the error checking is purposefully skipped off. This helps in reducing the extra bloat added to production level software codebasee

3. Number of hooks like presenting intermediate state output should be done in codebase. These hooks and output on console helps user in understanding the exact phases throught which final results produced.

4. If some third party softwares are needed to be ported, then one has to make sure that they are compliant to open source software license like *GPL*,*AGPL*. This is very much essential for overall code re-usability and understandability.

5. All the third party software utilities should be considered only if they fall under all the criterion's mentioned in literature survey.

6. It is very much important to produce comprehensive documentation for given prototype using tools such `doxygen`.

6.2 Future work

6.2.1 complete standalone compiler bundle like tcc

Currently xv6 is equipped with a subset of C language interpreter named *Picoc*, an elementary linkage editor which can link together three relocatable object files and some other status utilities. But one important thing that it is lacking right now is a complete compiler tool chain. One of the main reason for this is lacking of such small and working compiler tool chain for educational purposes. As found in literature survey, some of the existing options are not feasible enough for this project. Hence the future step for this project is to develop such one small time compiler bundle.

Looking at this future prospect we have started working on *TCC* compiler bundle. *TCC* contains standalone translator, assembler and linker. If these efforts come true then xv6 can have utilities that will give overview of complete compilation process as per mentioned in framework described above.

6.2.2 prototype network stack for xv6

One of the future enhancements to this project would be to add an elementary network stack. This stack will help in understanding one of the most important subsystems of modern day operating systems.

6.3 *MOOC*

As mentioned in the desired framework for developing such educational purpose systems, one has to focus on innovative documentation methods. Also it is very much important on part of developers to represent these prototypes to intended users such as novice developers, students through comprehensive coursework.

Considering this fact we have started working on configuring *Openedx* platform, to host this new experimental course work through the medium of massive open online coursework ($MOOC$) medium. The main emphasis will be given on designing assignments that will help student understand the underlying concepts with appropriate depth and width.

Appendix A

Library Functions

Functions	Usage
acos.S	arc cosine function
asin.S	arc sine Function
atan.S	arc tangent Function
log.S	logarithmic function
pow.c pow.S	power function
floor.S	floor function
ceil.S	ceil function
cos.S	cosine function
sin.S	sine Function
tan.S	tangent Function
acosh.c	inverse hyperbolic Cosine function
asinh.c	inverse hyperbolic Function
atanh.c	inverse hyperbolic Tangent Function
exp.S	exponetial function
fmod.S	modulo function

Table A.1: Mathematical Functions and their usage

Function	File	Description
__stdin_is_tty	stdin.c	Check if terminal is available
__fflush_stdin	stdin.c	Flushes the standard input
__fflush_stdout	stdout.c	Flushes the standard output
__fflush_stderr	stderr.c	Flushes the standard error
vprintf	vprintf.c	write output to character string 'str'
strtol	strtol.c	converts initial part of the string in nptr to a long integer value
strtoul	strtoul.c	convert string to unsigned long long int value
printf	printf.c	prints output to standard output
__ltostr	__ltostr.c	converts unsigned long integer to character string
__lltostr.c	__lltostr.c	converts unsigned long long integer to character string
__isinf	__isinf.c	check if number can be consider as infinity
__isnan	__isnan.c	check if number is NULL
fwrite	fwrite.c	writes nmemb elements of data, each size bytes long
fputc_unlocked	fputc.c	writes the character c, cast to an unsigned char, to stream
fread	libfile.c	reads n member elements of data, each size bytes long
fwrite	libfile.c	writes n member elements of data each bytes long
fclose	libfile.c	function flushes the stream pointed to by fp and closes the underlying file descriptor.
fopen	libfile.c	opens file whose name is the string pointed to by path and associates a stream with it
fseek	libfile.c	sets the file position indicator for the stream pointed to by stream
fputs	libfile.c	writes the string s to stream, without its terminating null byte

Table A.2: Standard i/o Library Functions and their usage

Appendix B

Linking script

ld -N -e main -Ttext 0 -o _picoc clibrary.o heap.o lex.o parse.o platform.o stdio.o type.o expression.o include.o library_unix.o picoc.o platform_unix.o table.o variable.oulib.o usys.o printf.o umalloc.o libfile.o pow.o exp.o liblog.o

Appendix C

Source Code for New Library Functions Added

```
FILE *fopen(char *path,char* mode){
    int fd ;
      FILE *fp;
        fp  = (FILE *)malloc(sizeof(FILE));

          if(fp == (void *)0);
    //     printf("malloc problem\n");

      int m,seekset = 0;
        m = checkmode(mode,&seekset );
          fd = open(path, m);

            if(seekset)
            lseek(fd,0,2);

        fp->desc = fd;
          return fp;
int fread(void *ptr, size_t size, size_t nmemb, FILE *stream) {
  if(stream == (void*)0)
    return 0;
  int res;
  /*printf("fd = %d\n",stream->desc );*/
  res = read(stream->desc,ptr,size*nmemb);
  return res;
}


int fwrite(void *ptr, size_t size , size_t nmemb, FILE *stream){
  if(stream == (void*)0)
    return 0;
// if STDIO is buffered implementation should be added its simple direct implememtation
  int res;
  res = write(stream->desc, ptr, size*nmemb);
  return res/size;
}


int fclose(FILE *fp){
        int ret;
  if(fp == (void*)0)
    return 1;
  ret = close(fp->desc);
  free(fp);
  return ret;
 // exit(1);
}
}
int fprintf(FILE *stream,char *fmt, ...)
{
  char *s;
  int c, i, state, stringPointer;
  uint *ap;
  stringPointer = 0;

  state = 0;
  /* due to +1 it fetches the second argument*/
  ap = (uint*)(void*)&fmt + 1;
  for(i = 0; fmt[i]; i++){
```

```c
    c = fmt[i] & 0xff;
    if(state == 0){
      if(c == '%'){
        state = '%';
      } else {
        putc(stream->desc, c);
        stringPointer++ ;
      }
    } else if(state == '%'){
      if(c == 'd'){
        printint(stream->desc, *ap, 10, 1,&stringPointer);
        ap++;
      } else if(c == 'x' || c == 'p'){
        printint(stream->desc, *ap, 16, 0,&stringPointer);
        ap++;
      } else if(c == 's'){
        s = (char*)*ap;
        ap++;
        if(s == 0)
          s = "(null)";
        while(*s != 0){
          putc(stream->desc, *s);
          s++;
          stringPointer++ ;
        }
      } else if(c == 'c'){
        putc(stream->desc, *ap);
        ap++;
        stringPointer++ ;
      } else if(c == '%'){
        putc(1, c);
        stringPointer++ ;

      } else {
        // Unknown % sequence.  Print it to draw attention.
        putc(stream->desc, '%');
        putc(stream->desc, c);
```

```
          stringPointer += 2;
        }
        state = 0;
      }
    }
    return stringPointer;

}
```

Appendix D

Source Code for Newly Designed Linker

```c
typedef struct Node{
  char name[16];
  GElf_Sym *symbol;
  struct Node *next;
  int argindex;
}Node;

int definition_not_found(Node *undefined_object_file){
  Node *p;
  p = undefined_object_file;
  printf("............................\n");
  traverse(undefined_object_file);
  p = p->next;
  while(p != NULL){
    if(p->argindex >= 0){
      printf("%s: missing definition\n",p->name);
      return 1;
    }
    p = p->next;
  }
  return 0;
```

```
}


void get_undefined_entries(Elf *elf,Node **undefined_object_file1,Node **
    defined_object_file1,int argindex){

  /* segregate symbol table entries into defined and undefined
     argument : Elf *elf, from which symbol are to be separated out
     requirements: Global pointers from undefined symbols and defined symbols of type
    GElf_Sym ** should be declared
     return values: It returns none, but will malloc above mentioned arrays */
  int i,j,k,symbol_count;
  Elf_Scn *scn1 = NULL;
  GElf_Shdr shdr;
  Elf_Data *edata = NULL;
  GElf_Sym *sym  ;
  i = j = k = 0;

  while((scn1 = elf_nextscn(elf, scn1)) != NULL)
  {
    gelf_getshdr(scn1, &shdr);

    // When we find a section header marked SHT_SYMTAB stop and get symbols
    if(shdr.sh_type == SHT_SYMTAB)
    {
      // edata points to our symbol table
      edata = elf_getdata(scn1, edata);
      symbol_count = shdr.sh_size / shdr.sh_entsize;
      int counter = (int)sizeof(Elf32_Rel);

      /*  how many symbols are there?
        his number comes from the size of // the section divided by the entry size
        libelf grabs the symbol data using gelf_getsym()
       */

      for(i = 0 ; i < symbol_count ; i++){

        sym = (GElf_Sym *)malloc(sizeof(GElf_Sym));
        gelf_getsym(edata, i, sym);

        if(sym->st_shndx  == SHN_UNDEF){
//            if(ELF32_ST_BIND(sym->st_info) == 1)//this checks for local declaration
            insert(undefined_object_file1,sym,elf_strptr(elf,shdr.sh_link,sym->st_name),
    argindex);
        }else{
          if(ELF32_ST_TYPE(sym->st_info) != STT_FILE)
            insert(defined_object_file1,sym,elf_strptr(elf,shdr.sh_link,sym->st_name),
    argindex);
        }
      }
    }
  }
}
void do_relocation(Elf *elf,Node *defined1,char *text,char * data,int textsize, int
    start_text ,  int start_data){
  int symbol_count;
  int i;
  Elf_Scn *scn1 = NULL;
  GElf_Shdr shdr;
  Elf_Data *edata = NULL;
  GElf_Rel *rel,rel_mem;
  char *sym_name;
  Node *symbol_entry = NULL;
  char *refptr = NULL;
  int tmp;

  while((scn1 = elf_nextscn(elf, scn1)) != NULL)
  {
    gelf_getshdr(scn1, &shdr);

    if(shdr.sh_type == SHT_REL){
      // edata points to our symbol table
      edata = elf_getdata(scn1, edata);
      symbol_count = shdr.sh_size / shdr.sh_entsize;
      int counter = (int)sizeof(Elf32_Rel);
      for(i = 0 ; i < symbol_count ; i++){
        /*get every entry from relocation table*/
        rel = gelf_getrel(edata,i,&rel_mem);

        /*get it's newly modified address from defined1 symlist*/
        sym_name = get_symbol_name(elf,(int)GELF_R_SYM(rel->r_info));
        symbol_entry = (Node *)ispresent(defined1,sym_name);
        printf("symbol is %s and symbol value %d\n",sym_name ,(int) symbol_entry->symbol
    ->st_value);
        if(symbol_entry == NULL){
          printf("Undefined reference to symbol %s\n",sym_name);
          return ;
        }

        /*check relocation table entries type from relocation table*/
        if(ELF32_ST_TYPE(rel->r_info) == R_386_PC32){
               /*if it is a function*/
            refptr = text + (int)rel->r_offset + start_text;
          if(symbol_entry->symbol->st_shndx == 1){
            /* it from text section */

            tmp  = (int)(( symbol_entry->symbol->st_value) - (int)rel->r_offset -
    start_text + *(int*)refptr );
            printf("value of tmp in 1 is %x value is %d\n", tmp,(int)symbol_entry->
    symbol->st_value);
            //printf("value in text pc32  tmp = %d and rel offset = %d\n",(int)tmp,(int)
    rel->r_offset);
            *(int *)refptr =  tmp;
          }else if(symbol_entry->symbol->st_shndx == 2){
            tmp = (int)(  textsize - symbol_entry->symbol->st_value -(int)rel->r_offset
    +*(int*)refptr);
            printf("value of tmp in 1 is %x value is %d\n", tmp,(int)symbol_entry->
    symbol->st_value);
        //    *(int *)refptr =  tmp;
          }
        }else if(ELF32_ST_TYPE(rel->r_info) == R_386_32){
          /*if it is a variable*/
          refptr = text + (int)rel->r_offset + start_text;
          if(symbol_entry->symbol->st_shndx == 1){
            /* it from text section */
          //  refptr = text + (int)rel->r_offset;
             tmp =  (int)( symbol_entry->symbol->st_value);
            *(int *)refptr =  tmp;
          }else if(symbol_entry->symbol->st_shndx == 2){
          //  refptr = data + (int)rel->r_offset;
            tmp = (int)(  textsize + (int)symbol_entry->symbol->st_value);
            *(int *)refptr =  tmp;
          printf("tmp is %x\n",tmp);
          }
          printf("value of tmp is %x and textsize %d and value = %d \n",tmp,textsize,(
    int)symbol_entry->symbol->st_value);
        }else{
          printf("%d type not supported\n",(int)ELF32_ST_TYPE(rel->r_info));
          exit(1);
        }
      }
    }
  }
}
```

Appendix E

Object File Definitions

```
typedef struct {
  Elf32_Word st_name;
  Elf32_Addr st_value;
  Elf32_Word st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32_Half  st_shndx;
} Elf32_Sym;

typedef struct {
  Elf32_Addr r_offset;
  Elf32_Word r_info;
} Elf32_Rel;
typedef struct {
  Elf32_Addr r_offset;
  Elf32_Word r_info;
  Elf32_Sword r_addend;
} Elf32_Rela;
```

Appendix F

New In memory Inode Structure of xv6 File System

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses  *changes made
};
```

Bibliography

[1] Bluefire os. Website. http://code.google.com/p/blue-fire-os/.

[2] Dietlibc c library. Website. http://www.fefe.de/dietlibc/.

[3] Indian university knowledge base. Website. http://kb.iu.edu/data/akqn.html.

[4] Linux kernel version 0.01. Website. https://www.kernel.org/pub/linux/kernel/Historic/.

[5] MINIX 3 official. Website. http://www.minix3.org.

[6] Mit's xv6 operating system course. Website. http://pdos.csail.mit.edu/6.828/2012/.

[7] Picoc small c interpreter. Website. http://code.google.com/p/picoc/.

[8] Plan 9. Website. https://www.kernel.org/pub/linux/kernel/Historic/.

[9] Sos simple operating system. Website. http://sos.enix.org/en/MainPage.

[10] Xinu web resource. http://www.xinu.cs.purdue.edu.

[11] Yale's xv6 operating system course. Website. http://zoo.cs.yale.edu/classes/cs422/2014/.

[12] Robert Love. *Understanding the Linux Kernel 3rd edition*. OReilly Media, 2001.

[13] Robert Morris Russ Cox, Frans Kasshoek. *xv6 a simple, Unix-like teaching operating system*. MIT, June 2009.