A

PROJECT REPORT

on

Glendix: A Plan 9 / Linux Distribution

for partial fulfillment for the degree of

Bachelor of Technology

in

Computer Engineering

(2007-08)

Contents

List of Figures

List of Tables

ABSTRACT

GNU/Linux is a popular free operating system in use today. GNU/Linux strives to be strictly compliant with POSIX standards, and is thus tied down with several requirements, ceasing to be innovative as far as operating system design is concerned.

Plan 9 from Bell Labs, on the other hand, was designed to be a from-scratch successor to UNIX. The Plan 9 operating system offers several new features that are both useful and efficient in today's era of personal computing: synthetic file systems, per-process namespaces and a fresh look at graphics and text editors are just a few.

The Linux kernel, however, is far more popular than Plan 9's and is better supported by developers and companies alike. Besides, Linux also has a vibrant community backing it and supports a lot more commodity hardware. This project aims to combine Plan 9 user-space utilities with the Linux kernel, to offer today's developer an exciting environment combining the best features of both worlds.

ACKNOWLEDGEMENTS

This project was born from earlier open source projects, so we would like to begin by thanking the Plan 9 and Linux communities for giving us such great software and support to work with. Specifically, we would like to thank Charles Forsyth, Russ Cox, Rene Herman and Al Viro, who contributed significantly to the project by offering their insightful comments, suggestions and help.

We would also like to thank Dr. Vijaylaxmi for her timely feedback and suggestions on the project. We would especially like to thank Dr. M.S. Gaur who kindly agreed to supervise the project and has provided unsurpassable support for our work.

Chapter 1

Introduction

An operating system is probably the most important component in any computer. Operating systems research is very important to the health of computing, as it forms the foundation for all other software.

When the term 'operating system' is used, we generally think of a complete package that allows us to use our computer. Technically, we may split an operating system into two synergic, yet distinct components:

- **The kernel**: This is the layer that is responsible for directly controlling hardware and providing a abstraction over all the different devices connected to a computer system. The kernel is also responsible for providing a basic framework using which software can be written. Some of the important functions of a kernel include - providing device drivers, managing & scheduling processes and implementing a network stack.
- **User-space applications**: These are applications with which the users directly interact. They use the abstracted interface provided by the kernel and provide useful functionality to the end user. They may or may not be considered as system software. Some examples are - C compiler, text editor, interactive shell and window manager.

A graphical depiction of role of a kernel in a traditionally 'layered' operating system in shown in Figure 1.1.

Before we delve into the details of our project, it may be worthwhile to first introduce the technologies that we are basing our project upon.

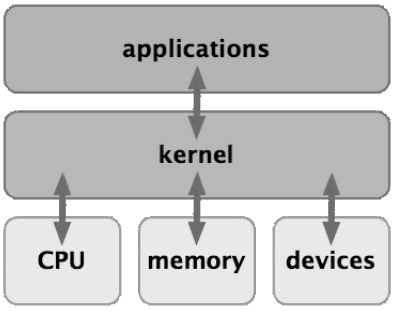

FIGURE 1.1: Role of a kernel

1.1 Plan 9

Plan 9 is the research successsor to Unix developed by Bell Labs as an attempt to build a system that was centrally administered and cost-effective using cheap modern microcomputers as its computing elements. The idea was to build a time-sharing system out of workstations, but in a novel way. Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers. For the central machines, the coming wave of shared-memory multiprocessors seemed obvious candidates. The main motto was: "To build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes" [6].

Hence Plan 9 incorporates and introduces lot of new things which are non trivial:

- **Everything is a file**: Resources are named and accessed like files in a hierarchical file system. This used to be a trivial thing which every OS incorporated, but as systems "progressed" the basic idea began to be lost. New complications, namely: Berkeley sockets, the X Window System and use of hardware-specific control mechanisms like *ioctl* were introduced due to claims that "everything did not map neatly to a file". Therefore, in Plan 9, *everything is file witout exception* [4]. You can read and write to any device or file using common functions.

- **The 9P protocol**: Since Plan 9 depends heavily on files, a simple file protocol was required. Plan 9 aims to provide users with a workstation-independent working environment. 9P (also know as Styx) [8] is the standard protocol to access all resources (local or remote) on Plan 9.

- **Unicode**: Plan uses UTF-8 encoding for all text in the system [9]. UTF-8 is a unicode encoding that is backward compatible with ASCII.
- **Union directories**: Plan 9 also introduced the idea of union directories, directories that combine resources across different media or networks, binding transparently to other directories. For example, another computer's */bin* (applications) directory can be bound to one's own, and then this directory will hold both local and remote applications, which the user can access transparently. Unix-style links are environment variables are no longer required!
- **Namespaces**: Disjoint hierarchies provided by different services are joined together into a single private hierarchical file name space in Plan 9 [7]. Also, every process has its own namespace, this increasing modularity, improving security and making application development a whole lot easier.

Plan 9 is also a lot smaller and lighter than most other operating systems. A standard Plan 9 ISO is around 250MB, which includes a window manager, text editor, C compiler and other development tools. Since Plan 9 is mainly a research vehicle, we don't usually see it being used on standard desktop environments, or even regular servers.

Many of Plan 9's ideas have been adopted by other operating systems. For instance, the */proc* synthetic filesystem (used to probe into internal kernel data structures) has been implemented by Linux, and the concept of union directories is catching on too.

The Plan 9 operating system includes both a kernel and a set of user-space applications.

1.2 Linux

The term "Linux" is actually a little ambiguous. As discussed before, an operating system can refer to both the kernel as well as user-space applications. Linux is a kernel, the portion responsible for managing system resources. Linux, by itself, is not of much use - you'd need a whole set of user-space applications to be able to use the computer.

The Linux kernel was initially conceived and assembled by Linus Torvalds in 1991. Early on, the MINIX community contributed code and ideas to the Linux kernel.

At the time, the GNU Project had created many of the components required for a free software operating system, but its own kernel, *GNU Hurd*, was incomplete and unavailable. The BSD operating system had not yet freed itself from legal encumbrances. This meant that despite the limited functionality of the early versions, Linux rapidly accumulated developers and users who adopted code from those projects for use with the new operating system. Today, the Linux kernel has received contributions from thousands of programmers.

Hence, when we refer to Linux as a complete operating system, we include all the user-space applications provided by GNU. By putting Linux and GNU utilities together, we obtain a "Linux distribution", a complete and usable operating system. Prominent Linux distributions include *Debian*, *Ubuntu*, *Gentoo* and *Fedora*.

Even though the code for Plan 9 is licensed under an open-source licenses, Linux is one of the most prominent examples of successful open source software. This has resulted in it being adopted by a large amount of developers and corporations alike. The popularity of Linux, has in turn, significantly improved its hardware support - Linux runs on a lot more hardware than the Plan 9 kernel. However, Linux is simply a clone of Unix, and offers nothing new in terms of functionality or innovation.

Any further references to the word 'Linux' in this report shall refer to the kernel only.

1.3 What is Glendix?

Now that we know what Plan 9 and Linux are, it's time to discuss Glendix. The name of our project is derived from the two words 'Glenda' and 'Tux'. Glenda the rabbit is the mascot of the Plan 9 operating system, while Tux the penguin is the mascot for the Linux kernel.

We believe Plan 9 has a lot to offer in terms of features and functionality to the end-user. However, while the Plan 9 kernel is an excellent example of kernel design, it lacks in terms of device drivers. Plan 9 does not run on several commodity hardware, thereby severely reducing its adoption rate. Most people run Plan 9 in virtual machines, not on actual hardware. Linux, on the other hand has had years of work by thousands of developers put into it. It runs on significantly larger amounts of hardware than the Plan 9 kernel.

In this project, we decouple Linux from GNU utilities, and port Plan 9 user-space applications to run on the Linux kernel. In summary, we are combining the Plan 9 user-space with the Linux kernel-space - resulting in a "hybrid" operating system. We think this would offer the best of both worlds - great hardware support with a cutting-edge application development environment.

Figure 1.2 represents a graphical depiction of the Glendix's composition.

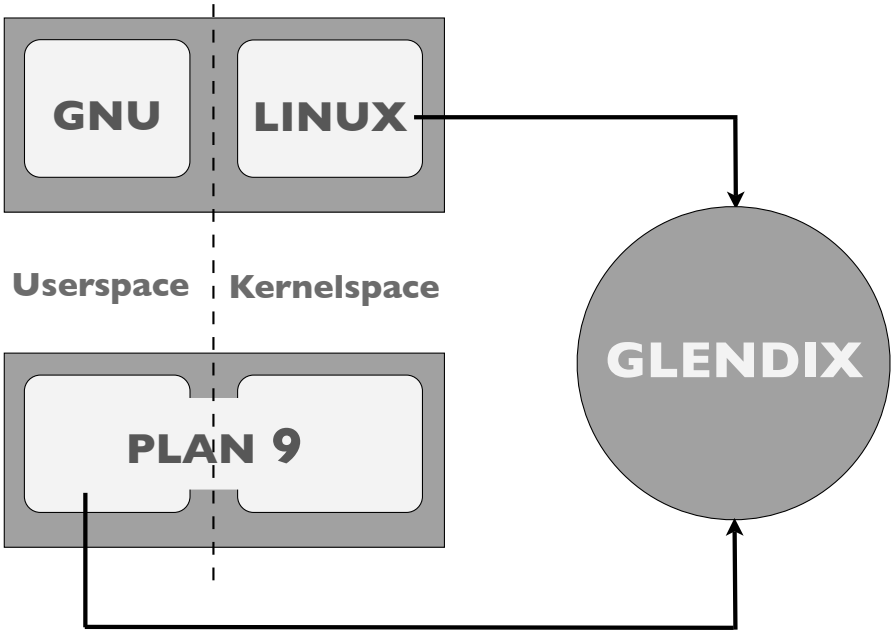

FIGURE 1.2: Composition of Glendix

The primary goal of the project is to create a Linux based operating system that includes the most important user-space applications from Plan 9. For brevity, we restrict our work to only the Intel x86 architecture. All technical discussions in this report apply only to that platform.

Chapter 2

Review

In order for us to proceed on the project, it was important to understand all existing technologies thoroughly and convince ourselves of the feasibility of our stated objectives. In this chapter, we present to you a review of all software related to the project.

2.1 GNU vs. Plan 9

The first and foremost question that we needed to ask ourselves was: "Why is there a need to replace GNU"? While GNU software certainly has its merits (such as being the first 'free software' suite), it has several disadvantages that we found are limiting in certain applications. GNU software is widely being used in desktop and server systems, where memory and processing power is in plenty, but is not even considered for embedded applications. Why?

Before we can answer that question, we must begin with an assumption: *Complexity is bad*. When a computer program can be kept simple, it must. Given two pieces of software that perform the same task, the one with less complexity should be preferred as it will be more maintainable, portable, and in general be easy to develop. [5]

GNU based software severely lacks in this respect. All of GNU software is unnecessarily complex, in the name of portability. On the other hand, Plan 9 applications are equally portable and perform the same functions, but are drastically less complex. To put this into context, we present to you a comparison of the lines of code in the source of various GNU and Plan 9 applications in Table 2.1

Table 2.1: Comparison of GNU & Plan 9 applications

Application	Lines of Code	
	GNU	Plan 9
cat	786	36
echo	276	40
wc	695	309
tee	219	75
ls	4489	321
mv	449	236
tar	17352	1267
C Compiler	843322	49447
C Library	790322	20039
Debugger	376392	5514

As you can see, all Plan 9 applications are much smaller than their GNU counterparts. The difference is even more significant as the complexity of the task itself increases, the GNU C Compiler (GCC) source code is 17 times KenCC's (the Plan 9 suite of C compilers) size! We would like to emphasize here that Plan 9 supports just as many processor architectures as GNU, and provides the same set of features (if not more). Clearly the complexity of GNU based software is larger than Plan 9's by a staggering amount.

Additionally, all Plan 9 binaries are statically linked; which means that all code needed to execute that particular program is present in the executable. The executable only needs to be laid out in memory, and no additional requirements are necessary. GNU binaries, on the other hand, are usually dynamically linked; which means that the portions of library code used need to be loaded at runtime. While the debate of whether static or dynamic libraries are better rages on, it is generally accepted that statically linked binaries are easier to debug and are preferred for embedded applications.

2.2 Possible Approach

After a review of the current state of Plan 9 applications and the Linux kernel, there were two basic approaches to solving the problem of running Plan 9 applications on Linux. We discuss both approaches here.

2.2.1 Source Compatibility

An existing software package called 'Plan 9 from User Space' (also known as 'plan9port') is a port of many Plan 9 programs from their native Plan 9 environment to Unix-like operating systems. The primary goal of this project is to allow Unix users to benefit from Plan 9 applications. Development in the 'Plan 9' style is not very polished in plan9port, as source-level compatibility is (as of yet) not entirely possible. You cannot, for example, take the source for a Plan 9 program and recompile it within a plan9port environment and expect it to work.

One of the approaches we reviewed early on during the project was achieving source-level compatibility for all Plan 9 applications by building on plan9port. The most significant advantage for this type of approach is that Plan 9 applications will be able to run on a variety of Unix clones (not only Linux) just by a recompile of the program. These include operating systems like FreeBSD, Mac OS X, NetBSD, OpenBSD and SunOS.

However, this approach means we would have to write POSIX equivalents for all source libraries offered by Plan 9. This seemed like a step backward. The additional constraint of having to recompile the program in the new environment was not very appealing, and thus we chose to reject this approach.

2.2.2 Binary Compatibility

A more appealing solution was to achieve binary-level compatibility of all Plan 9 applications. The mantra here was "compile once, execute everywhere". At the end of the project, we wanted to ensure that it wouldn't matter where the program was compiled. The program should run as expected on both Plan 9 and Linux.

While this approach seems ideal, it turns out that Linux actually offers us the flexibility to achieve this kind of binary-level compatibility. In order for this approach to work, we have to make Linux behave exactly as a Plan 9 kernel would when requested by a Plan 9 application. After all, executables are just a series of instructions for the CPU.

The only channel of interaction between a user-space application and the kernel is the *system call*. System calls are the primary entry point for user-space applications to notify the kernel that they want something done. If we can program Linux to intercept these system calls and perform the required action before returning

a value, user-space applications will be oblivious to the fact that the underlying kernel is Linux, not Plan 9! We decided to adopt this approach because it was interesting and seemed to achieve our stated goals in a clean manner.

2.3 Steps of Operation

In order to achieve binary-level compatibility, we need to write code that performs the following:

- **Load the executable**: Write a loader that understands the Plan 9 executable format and loads it into memory. The loader also sets the registers to an appropriate state before allowing the CPU to process the instructions.
- **Intercept system calls**: Whenever an instruction signaling the request of a system call is executed by the CPU, it must be intercepted and handled.
- **Implement all system calls**: The actual system call functionality as provided by the Plan 9 kernel must be performed by the Linux kernel, and an appropriate return value returned.

In the following chapter, we discuss the methodology by which we achieve the above.

Chapter 3

Methodology

3.1 Loader

In this section we describe the Plan 9 executable format [1] (known as *a.out*, not to be confused by another executable format of the same name used in old versions of Unix). We then proceed to discuss how we wrote a loader for the Linux kernel to parse and prepare for execution of executables in the *a.out* format.

3.1.1 a.out

An executable Plan 9 binary file has up to six sections: a header, the program text, the data, a symbol table, a PC/SP offset table (MC68020 only), and finally a PC/line number table. The header, given by the structure in Listing 3.1, contains 4 byte integers in big-endian order.

Sizes are expressed in bytes. The size of the header is not included in any of the other sizes.

When a Plan 9 binary file is executed, a memory image of three segments is set up: the TEXT segment, the DATA segment, and the STACK. The text segment begins at a virtual address which is a multiple of the machine-dependent page size. The text segment consists of the header and the first *text* bytes of the binary file. The *entry* field gives the virtual address of the entry point of the program. The data segment starts at the first page-rounded virtual address after the text segment. It consists of the next *data* bytes of the binary file, followed by *bss* bytes initialized to zero. The stack occupies the highest possible locations in the core

image, automatically growing downwards. The BSS segment may be extended by the *brk_* system call.

```
typedef struct Exec {
    long magic;     /* magic number */
    long text;      /* size of text segment */
    long data;      /* size of initialized data */
    long bss;       /* size of uninitialized data */
    long syms;      /* size of symbol table */
    long entry;     /* entry point */
    long spsz;      /* size of pc/sp offset table */
    long pcsz;      /* size of pc/line number table */
} Exec;
#define    _MAGIC(b)    ((((4*b)+0)*b)+7)
#define    A_MAGIC      _MAGIC(8)     /* 68020 */
#define    I_MAGIC      _MAGIC(11)    /* intel 386 */
#define    J_MAGIC      _MAGIC(12)    /* intel 960 */
#define    K_MAGIC      _MAGIC(13)    /* sparc */
#define    V_MAGIC      _MAGIC(16)    /* mips 3000 */
#define    X_MAGIC      _MAGIC(17)    /* att dsp 3210 */
#define    M_MAGIC      _MAGIC(18)    /* mips 4000 */
#define    D_MAGIC      _MAGIC(19)    /* amd 29000 */
#define    E_MAGIC      _MAGIC(20)    /* arm 7something */
#define    Q_MAGIC      _MAGIC(21)    /* powerpc */
#define    N_MAGIC      _MAGIC(22)    /* mips 4000 LE */
#define    L_MAGIC      _MAGIC(23)    /* dec alpha */
```

Listing 3.1: a.out header structure

The next *syms* (possibly zero) bytes of the file contain symbol table entries, each laid out as a 4-byte *value*, followed by a byte of *type* and ending with a variable length null-terminated string *name*. The *value* is in big-endian order. The *type* field is one of the characters in Table 3.1 with the high bit set. The symbols in the symbol table appear in the same order as the program components they describe. The symbol table is used only by the debugger and is not laid out in memory.

3.1.2 How a File Gets Executed in Linux

Linux already supports a variety of executables - ranging from ELF (the native Linux executable format) to COFF. Hence, the foundation for adding support for a new executable format had already been laid, we simply had to use the tools that the kernel offered us. [3]

One of the roles that kernel modules can accomplish is adding new binary formats to a running system, so we chose to write a kernel module for the Plan 9 executable format. The single biggest advantage of writing a kernel module for this purpose

TABLE 3.1: a.out symbol types

Character	**Type**
T	text segment symbol
t	static text segment symbol
L	leaf function text segment symbol
l	static leaf function text segment symbol
D	data segment symbol
d	static data segment symbol
B	bss segment symbol
b	static bss segment symbol
a	automatic (local) variable symbol
p	function parameter symbol

is that we didn't have to recompile the kernel and reboot every-time we made a change to the loader - thanks to Linux's dynamic module loading/unloading facilities.

Let's take a look at how the *exec* system call is implemented in Linux. This is an interesting part of the kernel, as the ability to execute programs is at the basis of system operations.

The entry point of exec lives in the architecture-dependent tree of the source files, but all the interesing code is part of *fs/exec.c*. Within *fs/exec.c*, the toplevel function, *do_execve()*, is less than fifty lines of code in length. Its role is checking for errors, filling the "binary parameter" structure (linux_binprm) and looking for a binary handler. The last step is performed by *search_binary_handler()*, another function in the same file. The magic of *do_execve()* is contained in this last function which is very short. Its job consists of scanning a list of registered binary formats, passing the *binprm* structure to all of them until one succeeds. If no handler is able to deal with the executable file, the kernel tries to load a new handler via kerneld and scans the list once again. If no binary format is able to run the executable file, the system call returns the ENOEXEC error code ("Exec format error").

The main problem with this kind of implementation is in keeping Linux compatible with the standard Unix behaviour. That is, any executable text file that begins with **#!** must be executed by the interpreter it asks for, and any other executable text is run by */bin/sh*. The former issue is easily dealt with by a binary format specialized in running interpreter files (*fs/binfmt_script.c*), and the interpreter itself is run by calling *search_binary_handler()* once again. This function is designed to be reentrant, and binfmt_script checks against double invocation. The latter

issue is mainly an historical relic and is simply ignored by the kernel. The program trying to execute the file takes care of it.

All the magic handling of data structures needed to replace the old executable image with the new one is performed by the specific binary loader, based on utility functions exported by the kernel. We need to write precisely this type of loader.

3.1.3 Registration of Binary Formats

The implementation of *exec* is interesting code, but Linux has more to offer: registration of new binary formats at run time. The implementation is quite straightforward, although it involves working with rather elaborate data structures - either the code or the data structures must accomodate the underlying complexities; elaborate data structures offer more flexibility than elaborate code.

The core of a binary format is represented in the kernel by a structure called *linux_binfmt*, which is declared in the *linux/binfmts.h* file, also given in Listing 3.2.

```
struct linux_binfmt {
    struct linux_binfmt *next;
    long *use_count;
    int (*load_binary)(struct linux_binprm *, struct pt_regs *);
    int (*load_shlib)(int fd);
    int (*core_dump)(long signr, struct pt_regs *);
};
```

Listing 3.2: Linux binary format structure

The three functions, or "methods", declared by the binary format are used to execute a program file, to load a shared library and to create a core file. The next pointer is used by *search_binary_handler()*, while the *use_count* pointer keeps track of the usage count of modules. Whenever a process p is executing in the realm of a modularized binary format, the kernel keeps track of *use_count* to prevent unexpected removal of the module.

Of the three methods, we need only to implement *load_binary*. *load_shlib* is not required as all Plan 9 binares are statically linked, as discussed before; and *core_dump* is mainly used to generate core dumps readable by the GNU debugger (which we do not want to use).

3.1.4 The Binary Parameters

In order to implement a binary format that is of some use, the programmer must have some background information about the arguments that are passed to the loading function, i.e., *load_binary*. The first such argument contains a description of the binary file and the parameters, and the second is a pointer to the processor registers.

The second argument is useful because we must initialize the registers associated with the current process to a sane state. In particular, the instruction pointer must be set to the address where execution of the new program must begin. The function *start_thread* is exported by the kernel to ease setting up the instruction pointer.

The first argument, a *linux_binprm* structure contains the following fields:

- **char buf[128]**: This buffer holds the first bytes of the executable image. It is usually looked up by each binary format in order to detect the file type. We use this buffer to quickly check if an executable of the Plan 9 a.out format or not.
- **unsigned long page[MAX_ARG_PAGES]**: This array holds the addresses of data pages used to carry around the environment and the argument list for the new program. The pages are only allocated when they are used; no memory is wasted when the environment and argument lists are small. The macro MAX_ARG_PAGES is declared in the *binfmts.h* header.
- **unsigned long __user p**: This is a "pointer" to data kept in the pages just described. Data is pushed to the pages from high addresses to low ones, and p always points to the beginning of such data. Binary formats can use the pointer to play with the initial arguments that are passed to the program being executed. It's interesting to note that *p* is a pointer to user-space addresses, and it is expressed as *unsigned long __user* to avoid an undesired de-reference of its value. When an address represents generic data (or an offset in the memory "array") the kernel often considers it a long integer - like kernel-space addresses.
- **struct inode *inode**: This inode represents the file being executed.
- **int e_uid, e_gid**: These fields are the effective user and group ID of the process executing the program.

- **int argc, envc**: These values represent the number of arguments passed to the new program and the number of environment variables. Plan 9 doesn't use environment variables, so we ignore *envc*.
- **char *filename**: This is the full pathname of the program being executed. This string lives in kernel space and is the first argument received by the *execve* system call. Although the user program won't know its full pathname, the information is available to the binary formats, so they can play games with the argument list.
- **int dont_iput**: This flag can be set by the binary format to tell the upper layer that the inode has already been released by the loader.

3.1.5 Memory Layout and Padding

Once we've confirmed that the given executable is indeed in the Plan 9 format, we begin to load the contents of file into memory as dictated by the a.out format. Figure 3.1 shows the required memory layout for the executable.

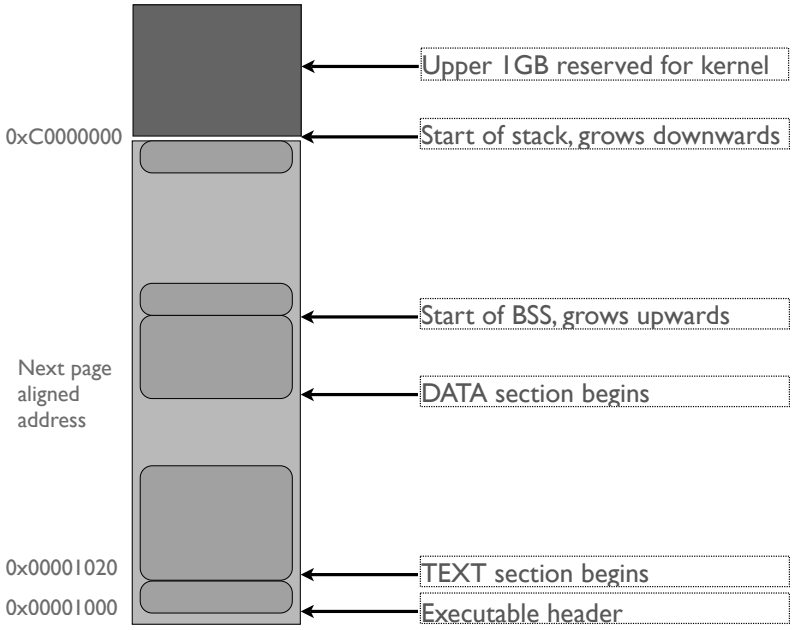

FIGURE 3.1: Memory layout for a.out

If you look at the figure, you'll note that there is a gap between the TEXT and DATA sections in memory, because of page-alignment. In the executable file, however, all sections are one after the other, so while copying the contents into memory we need to create the extra padding. Therein, lies the rub.

Linux follows a lazy memory allocation mechanism, where memory is allocated "just in time". Hence, all Linux executables use the *do_mmap()* function exported by the kernel to map an executable file to memory, the memory allocated may not actually exist until runtime. However, we cannot do that as the gap does not exist in the file, and memory mapping does not work on non page-aligned offsets.

As a workaround to this problem, we use Linux's interpreter capabilities to invoke a user-space program whenever an authentic a.out executable is found. This user-space program creates this extra padding in the file itself, which may then be memory mapped. This padding program also turned out to be extremely useful in the later stages of the project, as will be discussed in the next section.

3.1.6 Top of Stack

When we said earlier that system calls were the only way for Plan 9 user-space applications to interact with the kernel, we lied. The Plan 9 kernel initializes and maintains a special structure called Tos (Listing 3.3), which is also used to exchange data between the kernel and user-space applications.

```
struct Tos {
    struct              /* Per process profiling */
    {
        Plink   *pp;    /* known to be 0(ptr) */
        Plink   *next;  /* known to be 4(ptr) */
        Plink   *last;
        Plink   *first;
        ulong   pid;
        ulong   what;
    } prof;
    uvlong  cyclefreq;  /* cycle clock frequency if there is one, 0 otherwise */
    vlong   kcycles;    /* cycles spent in kernel */
    vlong   pcycles;    /* cycles spent in process (kernel + user) */
    ulong   pid;        /* might as well put the pid here */
    ulong   clock;
    /* top of stack is here */
};
```

Listing 3.3: TOS Structure

The Plan 9 kernel intializes this area above the user-space stack and stores the address in the accumulator, from which the user-space application retrieves it and stores in a global variable *_tos*. This works great in Plan 9, but Linux resets the accumulator immediately after the loader finishes (to signal the return value of

exec), so we can't use that register to notify user-space applications of the Tos address.

As a workaround, we used the padding program to mangle the instruction that fetched the address from *EAX* (the accumulator) and changed it to fetch the address from *EBX* instead (Linux does not modify *EBX* between the loader and the executable). The opcode for the *MOV* instruction is *0x89*. The first instruction in a Plan 9 user-space application, therefore, would usually be:

89 05 xx xx xx xx

where 'xx xx xx xx' denotes a 32-bit address (global variable _tos in the DATA section). We change this instruction to:

89 1D xx xx xx xx

in accordance with x86 assembly opcodes (Table 3.2) [2].

Table 3.2: 32-Bit addressing forms with ModR/M byte

		r32(/r)	**EAX**	**ECX**	**EDX**	**EBX**	**ESP**	**EBP**	**ESI**	**EDI**
Address	Mod	R/M	**Value of ModR/M Bytes (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[–][–]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	**05**	0D	15	**1D**	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F

3.2 System Call Handler

Once the loader had been written, the next major task was to be able to intercept system calls. Let us see how system calls are invoked by user-space applications in both Linux and Plan 9.

3.2.1 System Call Behavior in Linux

Linux maintains dual modes, namely: user mode and kernel Mode. System calls provide the means for a user program to ask the operating system to perform task reserved for the operating system on the user program's behalf. The various steps to invoke a system call in Linux are:

1. System calls must be invoked by executing the *INT 0x80* assembly instruction, which raises the programmed exception having vector 128.
2. When a user mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function.
3. The calling process passes a parameter called the system call number to identify the required system call; the *EAX* register is used for that purpose.
4. The kernel saves the contents of most registers in the Kernel Mode stack. Hence, other parameters to the system call (if required) are put on the registers by the calling processing before giving the interrupt.
5. The kernel handles the system call by invoking a corresponding C function in kernel-space called the system call service routine.
6. The handler is exited when the system call finishes, and the registers are restored. The return value of the system call is stored in the accumulator, where it can be picked up by the user-space application.

As an example, consider the "Hello World" program in pure assembly for Linux (Listing 3.4)

```
section .data
    hello: db 'Hello world!',10    ; 'Hello world!' plus a linefeed character
    helloLen:  equ $-hello         ; Length of the 'Hello world!' string
section .text
    global _start
_start:
    mov eax,4               ; The system call for write (sys_write)
    mov ebx,1               ; File descriptor 1 - standard output
    mov ecx,hello           ; Put the offset of hello in ecx
    mov edx,helloLen        ; helloLen is a constant
    int 80h                 ; Call the kernel
    mov eax,1               ; The system call for exit (sys_exit)
    mov ebx,0               ; Exit with return code of 0 (no error)
    int 80h
```

Listing 3.4: HW in assembly for Linux

3.2.2 System Call Behavior in Plan 9

Thankfully, the method of system call invocation by Plan 9 user-space applications is not very different from that described earlier. Plan 9 differs from the Linux procedure in only two big ways:

1. Plan 9 uses programmed exception vector 40 (0x64) to notify the kernel.
2. Plan 9 applications store arguments for the system call on the user-space stack, just like for any other function call.

An example program for Plan 9 will make the differences clear. (Listing 3.5)

```
DATA    string<>+0(SB)/8, $"Hello\n\z\z"
GLOBL   string<>+0(SB), $8

TEXT    _main+0(SB), 1, $0

MOVL    $1, 4(SP)
MOVL    $string<>+0(SB), 8(SP)
MOVL    $7, 12(SP)
MOVL    $-1, 16(SP)
MOVL    $-1, 20(SP)

MOVL    $51, AX
INT     $64

MOVL    $string<>+0(SB), 4(SP)
MOVL    $8, AX
INT     $64
```

Listing 3.5: HW in assembly for Plan 9

3.2.3 Programmed Exception Handling

Unfortunately, the Linux kernel was not built to support the interception of different interrupt vectors in a kernel module. This initialization is done at boot time, hence, for this part of the project, we had to directly edit the kernel source.

arch/x86/kernel/traps_32.c is where programmed exception gates are created. The routine *set_system_gate()* is provided by the kernel to set an Interrupt Service Routine (ISR) for a particular exception vector. We used that function to set a gate for interrupt vector 40. As for the Interrupt service routine, we copied the same

ISR as for interrupt 80, except that we call a custom system call implementation at the end: *sys_plan9*.

Let us analyze what happens in this case. The ISR copies the register values to the kernel stack as usual, and triggers *sys_plan9* with the appropriate arguments. We use the *EBP* register to obtain the stack pointer in user-space and extract the system call arguments by using the *__get_user()* routine provided by Linux. These arguments are in turn passed to an internal system call implementation (most of the time, a Plan 9 system call maps directly to one provided by Linux) and the value returned accordingly.

A snippet of the *sys_plan9* function is provided in Listing 3.6.

```
asmlinkage long sys_plan9(struct pt_regs regs)
{
    .
    .
    /* retrieving syscall arguments from user-space stack */
    unsigned long *addr=(unsigned long *)regs.esp;

    /* check syscall number and invoke appropriate system call service routine */
    switch (regs.eax) {
        .
        .
        case 51: /* pwrite */
                arg1 = *(++addr);
                arg2 = *(++addr);
                arg3 = *(++addr);
                addr = addr + 2;
                offset = (loff_t) *(addr);

                if (offset == 0xffffffff)
                    retval = sys_write(arg1, (const char __user *)arg2, arg3);
                else
                    retval = sys_pwrite64(arg1, (const char __user *)arg2,
                                          arg3, offset);
                break;
        .
        .
    }
}
```

Listing 3.6: System call routing

3.3 System Call Implementation

Table 3.3 lists all the system calls provided by the Plan 9 kernel. Not all Plan 9 system calls map directly to Linux equivalents. We focused only on implementing

system calls that were essential to get a basic Plan 9 toolchain running on Linux. We implemented 15 of the 39 system calls (the calls beginning with a '_' are not to be implemented, they are legacy calls from previous editions of Plan 9), and got a surprising number of applications to run.

Table 3.3: Plan 9 system calls

Number	**System Call**	**Number**	**System Call**
0	sysr1	26	_wstat
1	_errstr	27	_fwstat
2	bind	28	notify
3	chdir	29	noted
4	close	30	segattach
5	dup	31	segdetach
6	alarm	32	segfree
7	exec	33	segflush
8	exits	34	rendezvous
9	_fsession	35	unmount
10	fauth	36	_wait
11	_fstat	37	semacquire
12	segbrk	38	semrelease
13	_mount	39	seek
14	open	40	fversion
15	_read	41	errstr
16	oseek	42	stat
17	sleep	43	fstat
18	_stat	44	wstat
19	rfork	45	fwstat
20	_write	46	mount
21	pipe	47	await
22	create	50	pread
23	fd2path	51	pwrite
24	brk_	25	remove

Some system calls require us to write completely new code, as no existing system call (of the 300-odd ones that Linux offers!) provides the same functionality. An example is the *sys_fd2path* call, which returns the absolute path corresponding to a file descriptor (Listing 3.7).

```
asmlinkage long sys_fd2path(int fd, char __user *buf, unsigned long size)
{
    int error;
    struct vfsmount *mnt, *rootmnt;
    struct dentry *dentry, *root;
    struct file *file;

    char *page = (char *) __get_free_page(GFP_USER);

    if (!page)
        return -ENOMEM;

    file = fget(fd);
    if (!file)
        return -EBADF;

    mnt = mntget(file->f_vfsmnt);
    dentry = dget(file->f_dentry);
    fput(file);
    read_lock(&current->fs->lock);
    rootmnt = mntget(current->fs->rootmnt);
    root = dget(current->fs->root);
    read_unlock(&current->fs->lock);

    error = -ENOENT;
    /* Has the file been unlinked? */
    spin_lock(&dcache_lock);
    if (dentry->d_parent == dentry || !list_empty(&dentry->d_hash)) {
        unsigned long len;
        char * cwd;

        cwd = __d_path(dentry, mnt, root, rootmnt, page, PAGE_SIZE);
        spin_unlock(&dcache_lock);

        error = -ERANGE;
        len = PAGE_SIZE + page - cwd;
        if (len <= size) {
            error = len;
            if (copy_to_user(buf, cwd, len))
                error = -EFAULT;
        }
    } else
        spin_unlock(&dcache_lock);

    dput(dentry);
    mntput(mnt);
    dput(root);
    mntput(rootmnt);
    free_page((unsigned long) page);

    return error;
}
```

Listing 3.7: The fd2path System Call

Chapter 4

Conclusions and Future Work

We were successfully able to run a number a number of Plan 9 applications on Linux using the given methodology. Some of the prominent utilities that were tested are: *8c* (The C compiler for x86), *cat*, *echo*, *cal* (Calendar), *dc* (Precision calculator), *cb* (Code beautifier), *diff*, *grep*, *mk* (Makefile-like system for Plan 9), *tar* and *wc*.

We believe that this will provide an excellent base for developers to write large distributed applications and embedded system, with all the advantages mentioned in the introduction of this report.

Our work, however, does not end here. We hope to continue implementing more system calls, and introduce some of the more interesting features of Plan 9 to Linux:

- Per-process namespaces: The CLONE_NEWNS flags for the Linux *clone* system call may come in handy.
- Window-manager: For *Rio* (the Plan 9 window manager) to work in Linux, we need to create a synthetic filesystem in */dev/draw* that wraps over either the X11 system or Linux's native framebuffer.
- Other applications: such as *awk* and *sam* also depend on certain virtual filesystems which have to be created on the kernel side. This may not be a big problem, as Linux already provides */proc* as a synthetic filesystem.
- Provide Plan 9-like FS services: especially */net* as a wrapper over BSD-style sockets.

Bibliography

[1] *Plan 9 Programmer's Manual, Volume 1.* AT&T Bell Laboratories, 1995.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2A. 2006.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2005.

[4] T.J. Killian. Processes as files. *Proc. of Summer USENIX*, 1984.

[5] Karl J. Lieberherr. Controlling the complexity of software designs. *Proc. of the Internal Conference on Software Engineering*, 2004.

[6] Rob Pike, Dave Presotto, Sean Doward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. Technical report, AT&T Bell Laboratories, 1995.

[7] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *Operating Systems Rev.*, 27, 1993.

[8] Rob Pike and Dennis M. Ritchie. The styx architecture for distributed systems. Technical report, AT&T Bell Laboratories, 1999.

[9] Rob Pike and Ken Thompson. Hello world (in unicode). *Proc. of Winter USENIX*, 1993.