Glendix: A Plan9/Linux Distribution

Anant Narayanan
Shantanu Choudhary
Vinay K. Pamarthi
Manoj S. Gaur

Malaviya National Institute of Technology, Jaipur, India

ABSTRACT

We describe our approach of bringing the Plan 9 userspace to the Linux kernel in order to spread the use of Plan 9 tools amongst the Linux developer community.

1. Introduction

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 and thereby ceases to be innovative as far as operating system design is concerned. Plan 9 [1], 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 very compelling to a developer in today's era of of personal computing.

The Plan 9 kernel, however, supports only a bare minimum of hardware. That is one of the primary reasons of its unpopularity for day-to-day use. The Linux kernel, on the other hand, has had years of development behind it, and enjoys the support of several hardware components and developers alike.

We propose Glendix, a general purpose operating system that aims to combine the Plan 9 userspace with the Linux kernel, to offer today's developer an exciting environment for application development on personal computers and embedded systems alike.

The primary motivating factor here is to promote the Plan 9 style of application development to the large base of developers that Linux already has. A secondary factor is to eliminate the need for *GNU* [2] based userspace software, by replacing them with their lightweight Plan 9 counterparts, which are just as functional and portable. The resulting distribution would be a lightweight Linux based operating system.

In this paper, we describe the approach taken by us to create Glendix. We begin with a review of the different approaches possible, and then describe the chosen methodology, along with significant challenges and how we overcame them. We conclude with a summary of what has been done so far and a few notes on future work.

2. Review

From a broad perspective, there are two kinds of compatibility we can create between programs on Plan 9 and Linux. In this section, we discuss source and binary compatibility, and what they mean in the context of Glendix.

{anant@kix.in,choudhary.shantanu@gmail.com,pamarthi.vinay@gmail.com,msgaur@mnit.ac.in}

2.1. Source compatibility

"Plan 9 from User Space" (also known as *plan9port)* [3] is an existing software package for POSIX compliant operating systems that consists of ports of several Plan 9 applications. While most of Plan 9's libraries have also been ported, the solution is not completely perfect. For example, taking the source for a Plan 9 program and recompiling it using *plan9port* may not result in correctly working binaries all the time.

One of the approaches we reviewed early on during the project was very similar to *plan9port.* The most significant advantage for this approach is that Plan 9 applications can be run on a variety of UNIX clones (not just Linux) after a recompile.

However, this would require us to write POSIX equivalents of all the Plan 9 libraries, which seemed like a step backward. The additional constraint of having to recompile the program for each target environment was not very appealing (what if the sources were not available?), and thus we chose to reject this approach.

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.* We wanted to ensure that it wouldn't matter where the program was compiled, it should run as expected on both Plan 9 and Linux.

While this approach seems ideal, the Linux kernel provides the capability to support new binary formats, such as Plan 9's *a.out.* In order for this approach to work, we have to make Linux behave exactly as a Plan 9 kernel would, as far as applications are concerned. There are two primary channels for an application to access functionality provided by the Plan 9 kernel: system calls and file servers. If we were to provide suitable implementations of both in the Linux kernel, userspace applications should be oblivious to the fact that the underlying kernel is Linux and not Plan 9, which is exactly what we want.

We decided to adopt this approach because it was interesting and seemed to achieve our stated goals in a clean manner.

3. Methodology

In this section we discuss the implementation details of an *a.out* binary loader for Linux and Plan 9 style system call handling.

3.1. Loader

We will not describe the structure of a Plan 9 executable, which is already documented [4] in *a.out*(5). 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.

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 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, because that is central to our objective. The entry point of *exec* lives in the architecture-dependent tree of the source files, but all the interesting code is part of `fs/exec.c`. The toplevel function, `do_execve()`, performs some basic error checking, fills the "binary parameter" structure `linux_binprm` and looks for a suitable binary handler. The last step is performed by a seperate function `search_binary_handler()`, The function finds

the appropriate binary handler by scanning a list of registered binary formats, and passing the `binprm` structure to all of them until one succeeds. If no handler is able to deal with the executable file, the system call returns the ENOEXEC error code.

Linux is also compatible with the standard Unix behavior of supporting exectuable text files that begin with `#!`. Such files are executed with the help of an intepreter which is specified immediately after the `#!` symbol. For this purpose, a binary format specialized in running interpreter files (`fs/binfmt_script.c`), is included. The function is designed to be reentrant, and `binfmt_script` checks against double invocation. The ability to invoke an intepreter in a binary format handler helps us greatly, as we shall see later.

3.2. Binary format handling

As mentioned before, Linux offers the ability to register new binary formats at runtime. The implementation is quite straightforward, although it involves working with rather elaborate data structures – either the code or the data structures must accommodate 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:

```
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 *);
};
```

The three methods declared by the binary format are used to execute a program file, to load a shared library and generate a core dump, respectively. 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 only need to implement `load_binary`. `load_shlib` is not required as all Plan 9 binaries as statically linked, and `core_dump` is mainly used to generate core dumps readable by the GNU debugger (which we do not want to use).

The binary format handler receives two important parameters by the kernel. The first contains a description of the binary file and the second is a pointer to the processor registers. The first argument, a `linux_binprm` structure, contains, in addition to other fields, the first 128 bytes of the binary file (which enable us to quickly check the *magic* number, and decide if we want to execute this binary or not). We also get addresses of the data pages used to carry around the environment and argument list for the new program.

3.3. Memory layout and padding

Once we've confirmed that the given executable is indeed an *a.out* file, we begin to load its contents into memory. The layout in memory is described in detail in *a.out*(5) but take note of the fact that the in-memory representation of a binary does *not* match with that of the contents of the file. 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 this extra padding.

This was our first major challenge. We noticed that all of the binary formats Linux

supports, actually do contain the padding in the file itself, and therefore, all their handlers use the (in)famous `mmap()` call to directly map the file to memory. We cannot use that approach because `mmap()` does not work on non page-aligned offsets, and the DATA section is bound to be at such an address in the file.

As a workaround, we use Linux's interpreter capabilities (discussed earlier) to invoke a userspace program whenever an authentic a.out executable is found. This userspace 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 later stages of the project, as will be discussed in the next section.

3.4. Top of Stack

The statement that system calls are the only way for Plan 9 userspace applications to interact with the kernel is not entirely true. The Plan 9 kernel initializes and maintains a special structure called `Tos`, which is also used to exchange data between the kernel and userspace:

```
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;
        vlong kcycles;
        vlong pcycles;
        ulong pid;
        ulong clock;
        /* top of stack is here */
};
```

As you can see, there are several fields important for process profiling, which need to made available when a binary in executed. The Plan 9 kernel initializes this area above the userspace stack and stores the address in the accumulator, from which userspace applications retrieve and store it in a global variable `_tos`. This is done by all programs linked with *libc.* Linux, however, resets the accumulator immediately after the loader finishes (to signal the return value of *exec*), so we can't use that register to notify userspace applications of the `Tos` address.

As a workaround, we used the padding program described in the previous section, to mangle the instruction that fetched the address from EAX and changed it to fetch the address from EBX instead (Linux does not modify EBX in any way between the loader's end and the program's beginning). The opcode for the MOV instruction is `0x89`. The first instruction in a typical Plan 9 userspace application, therefore, would usually be:

```
89 05 xx xx xx xx
```

where 'xx xx xx xx' denotes a 32-bit address corresponding to the global variable `_tos` in the DATA section.

We change this instruction to:

```
89 1D xx xx xx xx
```

in accordance with x86 opcode table [6] for MOV:

		r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
Address	Mod	R/M	Value of ModR/M Bytes (In Hex)							
[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

3.5. System call handler

Once the loader had been written, the next major task was to be able to intercept system calls. In Linux, system calls are invoked using the 0x80 interrupt, which raises the programmed exception with that vector. The calling process passes the system call number to identify the required system call in the EAX register. The kernel saves the contents of most registers in the kernel mode stack, hence other parameters to the system call (if required) are placed on subsequent registers. The handler is exited when the system call finishes, and the registers are restored. The return value of the system call is placed in the accumulator, where it is picked up by the calling process. An example of a 'Hello World' program in pure assembly for Linux is provided for clarity:

```
section .data
        hello: db 'Hello World!', 10
        helloLen: equ $-hello
section .text
        global _start
_start:
        mov eax, 4
        mov ebx, 1
        mov ecx, hello
        mov edx, helloLen
        int 80h
        mov eax, 1
        mov ebx, 0
        int 80h
```

Thankfully, the method of system call invocation in Plan 9 is not very different from what is described above. The only two big changes are: a) Plan 9 uses programmed exception vector 0x40 to notify the kernel, and, b) Plan 9 applications store arguments for the system call on the userspace stack, just like for any other function call. An example program for Plan 9 will make the differences clear:

```
DATA    string<>+0(SB)/8, $-"Hello0
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
```

Unfortunately, the Linux kernel was not built to support the interception of different interrupt vectors in a kernel module. The initialization is done at boot time, hence, for this part of the project, we had to directly edit the kernel source (as opposed to a module as done for the binary format loader).

`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 0x40. As for the ISR, we copied the same routine as for interrupt vector 0x80, with the exception of calling a custom system call implementation in the end: `sys_plan9()`, irrespective of the system call number in the accumulator. The ISR copies the register values to the kernel stack as usual, and triggers `sys_plan()` with appropriate arguments. We use the value of the `EBP` register to obtain the stack pointer in userspace and extract system call arguments using the `__get_user()` routine provided by Linux. These arguments are in turn passed to an internal system call implementation. Sometimes this means calling an existing Linux system call, but in many cases, we had to write one from scratch (eg: `sys_fd2path)`. A snippet of the `sys_plan9` function is as follows:

```
asmlinkage long sys_plan9(struct pt_regs regs) {
  .
  /* retrieving arguments from userspace stack */
  unsigned long *addr = (unsigned long *)regs.esp;
  /* check syscall number and invoke */
  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;
  }
}
```

4. Conclusion

By implementing 15 of the 39 system calls, we got a surprising number of applications to run. Examples include *8c, sed, grep, echo, cat, tar, cb, cal* and *dc,* among others. We believe that on completing all the system calls, Glendix will provide an excellent base for developers to start writing applications on Linux in the "Plan 9 way". The ability to run unmodified binaries in both operating systems is not provided by any other existing alternative, with the exception of 9vx (which is discussed in the appendix). The performance of these binaries will be the same as other native linux binaries because all the supporting infrastructure is built directly into the kernel.

Glendix, at this stage, serves as proof of concept that ideas from the Plan 9 system can be integrated into the Linux kernel. However, in order to achieve the goal of providing a complete ''Plan 9 experience'' to application developers, there is a lot more to be done, which is discussed in the following section.

5. Future Work

While most of the system calls from Plan 9 map more or less directly to their Linux counterparts, some features are unique Plan 9. Process and address space management along with per-process namespaces are the two most important aspects that affect the implementation of system calls.

Recently, the Linux kernel added support for per-process namespaces via the `CLONE_NEWNS` flag for its `clone` system call. Hence, Linux already contains primitives for namespace manipulation, even if they are not exposed to userspace applications directly. We believe that system calls such as `mount` and `bind` can be implemented using primitives already provided by the Linux kernel, and indeed, we are already working on them. `rfork`, on the other hand, is a little trickier, especially because of the specific combination of the `RFMEM` and `RFPROC` flags; which results in the creation of a new process sharing everything with its parent, except for the stack. For this particular permutation, it will be neccessary to dig deeper into the memory management primitives provided by the Linux kernel, but is entirely possible. In fact, since we are dealing with kernel code here, anything is technically possible, the only variation amongst the different system calls is the amount of code to be changed and/or written.

The other major feature to be emulated is that of the synthetic file systems provided by the Plan 9 kernel. Since Linux already supports such file systems (atleast partially - examples are `/proc` and `/sys`), we think it will not be hard to extend this to true Plan 9 filesystems such as `/net`. `/dev/draw` can be built on top the native Linux framebuffer device.

Once we implement all the system calls and synthetic file systems correctly, there should be no perceivable difference between the Glendix kernel and a Plan 9 kernel as far as an application is concerned. Source code and other details pertaining to the project are available on `http://glendix.org/`. Developers are encouraged to participate!

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.

Major portions of Glendix were executed as a final term project at the Malaviya National Institute of Technology. We would like to thank Dr. Vijaylaxmi for her timely feedback and suggestions.

References

[1] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''Plan 9 from Bell Labs'', Computing Systems, **8**, 3, Summer 1995, pp. 221-254

[2] ''GNU's Not Unix'', http://www.gnu.org/

[3] Russ Cox, ''Plan 9 from User Space'', http://swtch.com/plan9port/

[4] ''Plan 9 Programmer's Manual'', http://plan9.bell-labs.com/sys/man/

[5] Alessandro Rubini, ''Playing with binary formats'',

http://www.linux.it/~rubini/docs/binfmt/binfmt.html

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

[7] Bryan Ford, Russ Cox, ''Vx32: Lightweight User-level Sandboxing on the x86'', USENIX Annual Technical Conference, Summer 2008.

Appendix: Comparison to 9vx

Vx32 [7] is a user-mode library that was recently developed at CSAIL, MIT. The primary purpose of the library is to provide a safe and portable execution environment for untrusted x86 code. One of the interesting applications of this is the ability to run Plan 9 executables on all platforms that Vx32 supports (currently FreeBSD, Linux and Mac OS X). 9vx is the project that uses Vx32 to run an instance of the Plan 9 system.

On the surface, it may seem like the outcomes of 9vx on Linux and Glendix are similar, but there are many important differences. Vx32 can be compared in a very rough sense to a virtual machine, and thus there is a disjoint between the binaries running inside it, and the operating system it runs on. Glendix, however, aims to provide a more close coupling between Plan 9 applications and the Linux kernel, whether you trust the executables or not. Secondly, Vx32 is restricted to x86 binaries only. While this paper discusses only the x86 implementation of Glendix, we can easily extend it to cover other architectures as well, given the cross-platform nature of both Plan 9 binaries and Linux.