LAP: a little language for OS emulation

Donn M. Seeley
Berkeley Software Design, Inc.

ABSTRACT

LAP, the *Linux Application Platform*, is a Linux emulation package for BSD/OS which uses a ‘‘little language’’ [Salu98] to describe transformations from Linux data types and values to BSD/OS data types and values, and *vice versa*. The little language simplifies and regularizes the specification of transformations, making the emulation easier to maintain. This paper describes the language and its place in the framework of LAP.

1. Introduction

The **Linux Application Platform** is a package of software that allows Linux [Linu00] applications to run under BSD/OS [BSDI00]. Although BSD/OS and Linux share a common executable file format (the Unix **ELF** format [Unix90]) and a common Unix-like programming interface (based on the IEEE **POSIX** interface [IEEE96]), they differ in the way that applications communicate to the operating system kernel, and they differ in the sizes and structure of their data types. The LAP software dynamically converts Linux data types into BSD/OS data types and *vice versa*, and it substitutes a BSD/OS kernel interface for the application's Linux kernel interface.

The LAP software uses *transformations* to convert data types, system call numbers and other parameters between Linux and BSD/OS. Transformations are written in a *transformation language*. The language looks a lot like C [ANSI89]; the inspiration for its syntax comes from **lex** [Lesk75]. The language specification tries to make the most common transformations also be the simplest ones to specify. Many transformations can be described by a simple C prototype declaration.

In the sections below, we'll discuss some of the design issues that we considered when writing the LAP software, and we'll introduce some of the features and describe how they work.

2. Motivation

2.1. Why do we need transformations?

Here are a few examples of the differences between BSD/OS and Linux which require transformation:

- BSD/OS follows the Unix Application Binary Interface standard [Unix90] for system calls, using the `lcall` instruction to transfer control to the operating system kernel. Linux uses the `int` instruction to transfer to the kernel, which generates a software interrupt, more like MS-DOS.
- The operating system assigns numbers to system calls, signals and error conditions, and BSD/OS and Linux use different numbers.
- The POSIX application programming interface requires certain abstract data types, which are defined using the C `typedef` mechanism. The BSD/OS and Linux systems use different real types for these data types for some interfaces. For example, the `uid_t` type describes user ID values; in BSD/OS, these values are unsigned 32-bit integers, while in Linux these are unsigned 16-bit integers.
- The POSIX API also requires certain aggregate data types which correspond to `struct` data types in C. The order of elements or the types of elements in a `struct` may be different in BSD/OS and Linux. For example, the file status structure `struct stat` in BSD/OS puts the timestamps before the sizes, while in Linux the sizes come first.

2.2. Why use a transformation language?

The most important reasons for using a transformation language are reliability and maintainability.

The transformation language makes transformations *reliable* by removing opportunities to commit errors in specifications. BSD/OS also has an emulation for SCO Unix SVr3.2; it uses a very primitive transformation table that required a lot of hand-composed support code. While the SCO emulation's transformations were reasonably well structured, using strict conventions for naming, parameter types and ordering, errors crept in, and the errors were sometimes very difficult to diagnose. A transformation compiler makes a lot of the ‘‘mechanical’’ work truly mechanical, reducing a class of errors.

The transformation language makes transformations *maintainable* by making them familiar and easy to read. The language looks like C and (for the most part) works like C; it's straightforward to read a transformation, and it's usually obvious what it does. When transformations are composed by hand, they are more difficult to read and understand, and hence the code is harder to fix. Of course the transformation compiler itself can have bugs, but its bugs usually affect groups of system calls rather than individual system calls, so they are easier to spot.

Other applications have used specialized languages for similar reasons. The original idea for the LAP transformation language comes from code template languages for compilers. The first such code template language that I used appeared in the Ritchie C compiler for 6th Edition Unix [Ritc79]; the Portable C Compiler [John79] and the GNU C Compiler [Free00] also have code template languages. One very direct inspiration was the master system call table in BSD-derived operating systems (`/sys/kern/syscalls.master`).

2.3. Design issues

The principal design goals for LAP were to:

- keep the specification simple
- keep the implementation simple
- make few modifications to Linux code
- maintain efficiency
- avoid burdening the operating system kernel

Keep the specification simple: If we can make the specifications for most transformations very simple, then we are likely to commit fewer errors when writing transformations. Also, the effort of writing transformations is reduced.

Keep the implementation simple: The code uses a **yacc** parser [John75] and a **lex** scanner to implement the transformation compiler. The output of the transformation compiler is more-or-less readable C, which we then compile using GCC. We use Berkeley DB [Olso99] to hold symbol databases. By using tools and keeping the code simple, we leave fewer opportunities for mistakes and spend less time on maintenance.

Make few modifications to Linux code: By interposing a relatively small number of low-level interfaces in front of actual Linux shared libraries, we reduce the chances that we will interfere with interactions between the application and the libraries, and we reduce the amount of work that we set for ourselves.

Maintain efficiency: We try to avoid design decisions that would cause us to add overhead by requiring us to block signals to protect internal data structures or make other expensive accommodations.

Avoid burdening the operating system kernel: We want to avoid changes to the BSD/OS kernel to support Linux emulation. This means that extra kernel resources don't need to be tied down for Linux emulation. It also means that installing new LAP software does not require an update to the kernel. It means that debugging is simpler, and that it's unlikely that an error in LAP software will cause the operating system to crash.

3. The LAP software

Before going into details of the transformation language, let's briefly look at the big picture - what is LAP and how does it work?

LAP changes the execution environment for Linux applications so that they can satisfy their requests for services using the native BSD/OS kernel, rather than a Linux kernel. LAP's operation is based on *dynamic linking* [Ging89]. When a Linux application runs, the operating system loads it into virtual memory along with a separate program known as the dynamic linker or **ld.so**. The dynamic linker is responsible for loading *shared libraries*, which are collections of useful executable code that the application needs in order to communicate to the operating system (among other things). LAP replaces the Linux dynamic linker with a lightly modified version that can make native BSD/OS system calls, and interposes a library named **liblinux** in front of unmodified Linux libraries. The **liblinux** library overrides the Linux libraries and causes the application to make calls into the native BSD/OS kernel rather than into a Linux kernel.

Both the modified **ld.so** and **liblinux** are built from source code written in C, assembly language and the transformation language. A separate *transformation compiler* converts the transformation language into C, which is in turn compiled by the native C compiler. LAP does not create or interpret transformations at runtime. Apart from its own libraries, LAP does not use any native BSD/OS libraries - all other shared libraries in the LAP environment are unmodified Linux libraries.

4. The transformation language

The transformation language resembles C, with inspiration from **lex**. Source code in the transformation language is translated into C by the **transform** program. By convention, source code files for transformations end in the suffix `.x`, while headers for transformation sources end in `.xh`. Transformations describe how to convert between Linux data types and system calls, and BSD/OS data types and system calls.

As an example, here is a transformation for the `stat()` system call:

```
int stat(const char *name, struct stat *buf);
```

Given the appropriate `stat.xh` header, this transformation causes **liblinux** to do the following:

- Execute an `lcall` instruction to perform the BSD/OS `stat()` service, placing the resulting data in a buffer on the stack.
- Convert the BSD/OS `stat` structure into a Linux `stat` structure in memory belonging to the application.
- If there is an error, convert the BSD/OS error code into a Linux error code and store it in the Linux `errno` location.

The following sections provide a more detailed description of the transformation language.

4.1. Lexical structure

The basic elements of the transformation language are similar to C. Unlike C, there is no macro preprocessor; however, it is possible to "escape" to C code and write C preprocessor code in that context.

The language defines *keywords* that introduce statements or qualify declarations. All C keywords are reserved. Several keywords that are specific to the transformation language are introduced in the syntax section below , along with the statements that use them. The `typedef` statement provides a mechanism for defining new keywords, analogous to the C `typedef` statement.

There are *names* and *numbers* that work much like they do in C. Names are introduced in C-like contexts such as function names, parameter names, structure tags, structure members, and so on. Names follow the usual C rules - they must begin with a letter or an underscore, and may contain letters, digits or underscores. Numbers also follow C rules; the **transform** program simply passes numbers into its C output without interpretation.

The transformation language treats specially those names that begin with a *foreign* or *native* prefix. Names that start with `LINUX_`, `linux_` or `__bsdi_` cause the **transform** program to place restrictions on the automatic mapping between Linux names and BSD/OS names. See below for more information on this feature. The transformation language also recognizes the special prefix `__kernel_` on function names; more on that below as well.

Strings in the transformation language are used only to give the names of header files. They are surrounded by double quotes and they don't follow C rules for escapes (yes, very crude).

Various *punctuation marks* and "syntactic sugar" are recognized, including parentheses, commas, semicolons and braces. Certain punctuation implies an escape to C. As in **lex**, text that appears inside percent-brace pairs `%{ ... %}` is treated as literal C, and text that appears between simple braces following a function declaration is also treated as C. This C text is included in the output from the **transform** program without significant alteration.

Comments and *whitespace* are basically the same as C: text inside slash-star and star-slash `/* ... */` is ignored, and comments, spaces, tabs and newlines serve to break input into tokens, but are otherwise collapsed together and ignored.

4.2. Syntax

The syntax of the transformation language is organized into *statements*. The transformation language itself provides only declarative statements. Any imperative statements must be coded in C inside C escapes. Here is a summary of the statements. Literal text appears in `fixed width` font, while text that varies appears in *`slanted fixed width`* font.

Include

```
include "header"
```

The `include` statement causes text from the named header file to be inserted into the program text at the current location. Note that there is no "`#`" character at the beginning of the line. If a program needs to include a C header so that text in C escapes can use the header information, then that C header must be included using a C escape too; for example, `include` includes a transformation language header, while `%{ #include` a C header.

Typedef

```
typedef type-specifiers ... name;
typedef type-specifiers ... name {
   in(name) { ... }
   out(name) { ... }
};
```

The first form of the `typedef` statement looks much like a C `typedef`. It declares *`name`* as a type name. If *`name`* begins with `linux_`, then the type is a *foreign* type with no BSD/OS equivalent; otherwise, the **transform** program creates a mapping between the given type name in Linux and the type with the same name in BSD/OS. In the latter case, there really are two types, but the difference is hidden by the mapping feature. Inside C escapes, the BSD/OS type has the usual name while the Linux version of the type is prefixed with `linux_`. Note that the transformation language does not define the BSD/OS version of a type name; you must provide that yourself in a C escape, either by including the appropriate header file or by writing an explicit C `typedef` statement.

As an example, the statement `typedef unsigned short uid_t;` in the transformation language says that there is a Linux type named `uid_t` that corresponds to a BSD/OS type `uid_t`, and that it is equivalent to the basic C type `unsigned short` in Linux. When `uid_t` is used in a parameter list or a structure definition, the Linux value is automatically copied (as if by assignment) into the corresponding BSD/OS value on input (which has a type corresponding to `unsigned int`), and the BSD/OS value is automatically converted into the Linux value on output. For example, the transformation `int setuid(uid_t uid);` converts the Linux `uid` value into a BSD/OS `uid` value before calling the BSD/OS system call.

The second form of the `typedef` statement allows you to specify *transformation functions* for the given integral type. The `in()` function is automatically called to convert Linux types into BSD/OS types, while the `out()` function is automatically called to convert BSD/OS types into Linux types. The parameter *`name`* represents the value to be transformed. The body of the function is given in C inside braces. One or both transformation functions may be omitted, in which case the value is transformed by assignment. As an example, the statement

```
typedef unsigned short dev_t {
    in(dev) {
        return (makedev(dev >> 8, dev & 0xff);
    }
};
```

specifies an input transformation for `dev_t` that converts Linux `dev_t` values into BSD/OS `dev_t` values using the BSD/OS `makedev()` macro. Note that a `typedef`'s transformation functions may be accessed directly inside C escapes by appending `_in()` or `_out()` to the type name; this is true of transformation functions in general.

Cookie

```
cookie type-specifiers ... name {
    name number;
    in(name) { ... }
    out(name) { ... }
};
```

The `cookie` statement is an enumeration statement that creates a type like a `typedef` and lists members of that type along with their Linux values. When an object of the given integral type appears in an input context and its value matches one of the enumerated values, that value is converted to the value with the corresponding name in BSD/OS. This is a fancy way of saying that cookies convert `#define` macros from Linux values to BSD/OS values and back. Inside C escapes, the Linux member names are prefixed with `LINUX_`. If a `cookie` member's name is given with a `LINUX_` prefix, the **transform** program assumes that there is no equivalent BSD/OS value; if the `cookie` type name itself is prefixed with `linux_`, **transform** assumes that none of the members have corresponding BSD/OS names (and it omits the `LINUX_` prefixes in C escapes). If a value of a given `cookie` type fails to match any of the listed numbers, the value is assigned without conversion - that means that you don't have to list names that have the same value in both Linux and BSD/OS. However, if there is an `in()` or `out()` function, it applies to unmatched values. This lets you take care of values that have no exact equivalent in Linux or BSD/OS. Note that you are responsible for supplying the BSD/OS cookie member definitions, usually by including the appropriate C header file inside a C escape.

As an example,

```
cookie int reboot_t {
    RB_AUTOBOOT          0x01234567;
    RB_HALT              0xcdef0123;
    LINUX_RB_ENABLE_CAD  0x89abcdef;
};
```

says (among other things) that `RB_AUTOBOOT` has the value `0x01234567` in Linux and that there is no direct BSD/OS equivalent for the Linux name `RB_ENABLE_CAD`. An object of type `reboot_t`, presumably the argument to `reboot()`, with value `0x01234567` would be converted to the BSD/OS value of `RB_AUTOBOOT`, which happens to be `0`. (Yes, Linux uses enumerated values rather than flags as arguments to `reboot()`.)

Flag

```
flag type-specifiers ... name {
    name number;
    name;
    in(foreign, native) { ... }
    out(native, foreign) { ... }
};
```

A `flag` works very much like a `cookie` but for flag bits rather than enumerated values. `Flag` values are tested for matches by logically `and`-ing against the appropriate Linux (on input) or BSD/OS (on output) value. If a match occurs, the corresponding BSD/OS (on input) or Linux (on output) value is logically `or`-ed in. Bits that aren't matched are copied unchanged, so you don't need to list flag values that are identical on both Linux and BSD/OS. A given input can match more than one flag value. If you provide a transformation function, it gets both the raw value and the converted value, so that you can use a complicated rule to add (or subtract) bits from the converted value after all of the specific conversions are made. If you specify a flag name without a value, **transform** assumes that the name is a BSD/OS name with no equivalent Linux value. If a member name has a LINUX_ prefix, **transform** assumes that the name is a Linux name with no equivalent BSD/OS value. Bits that have no equivalent are not copied by default; this is a handy way to clear bits that aren't supported and don't significantly affect the semantics. Inside C escapes, the Linux flag member names are prefixed with LINUX_.

Here's an example:

```
flag unsigned int cflag_t {
    LINUX_CSIZE 0000060;
    HUPCL 0002000;
    CRTS_IFLOW;
    in(f, n) {
        return (n | (f & LINUX_CSIZE) << 4);
    }
    out(n, f) {
        return (f | (n & CSIZE) >> 4);
    }
};
```

This flag encodes the flag bits for the c_cflag field of a termios structure. It says that the HUPCL bit under Linux has the value 02000 rather than 0x4000 as it does under BSD/OS. The BSD/OS CRTS_IFLOW bit has no equivalent under Linux, and we clear it by default in any conversion. The LINUX_CSIZE field is also cleared by default, but the transformation functions copy it to and from the BSD/OS CSIZE field, so the information isn't lost. Notice how the transformation functions must be careful to preserve the bits that were already converted when returning a value.

Struct

```
struct name {
    type-specifiers ... name;
    type-specifiers ... name[number];
    in(foreign, native, length) { ... }
    out(native, foreign, length) { ... }
};
```

A struct statement in the transformation language declares a Linux structure and guides its transformation into a BSD/OS structure (or the reverse). Structure members are declared like they are in C. A structure member whose name begins with linux_ is assumed to have no BSD/OS equivalent, and it doesn't get converted automatically. Unlike flags or cookies, structs have no defaults - all of the members must be listed, and if the BSD/OS version of the structure contains a member that is not present in the struct specification in the transformation language, **transform** assumes that no such member appears in the Linux version of the structure. Inside C escapes, the member names look exactly the way that they are declared - no prefixes are automatically prepended. (We can do this because structure member names have a scope local to the given structure.)

When converting structures, each member is converted using a transformation that is appropriate for the type of the member, or if no transformation for that type is available, it is copied by assignment. Arrays are always copied by assignment (actually, by a memcpy() call). It is important to note that **transform** doesn't transform structures, but rather structure *pointers*; the direction and size of the transformation are derived from context.

After the specific members have been converted, any transformation functions are applied. The parameters to the transformation functions are a pointer to the source structure, a pointer to the destination structure and the length of the destination structure. If the last member in a structure is an array, **transform** assumes that the structure has variable length and it copies everything from the start of the array to the end of the structure as determined by the length parameter. Structure transformation functions have void type, since the parameters are passed by reference.

Here is an example of a struct statement:

```
struct sockaddr {
    familycookie_t sa_family;
    char sa_data[14];
    in(f, n, len) { n->sa_len = len; }
};
```

The familycookie_t type is a cookie type that converts socket family values from Linux numbers to BSD/OS numbers and back. Because the structure ends with an array, it is considered a variable-length structure and the sa_data field fills out the structure to the given length len. The input transformation fills in the BSD/OS sa_len field using len, whose value was supplied elsewhere.

Function

```
type-specifiers ... name(parameters, ...);
type-specifiers ... name(parameters, ...) =
    syscall-name;
type-specifiers ... name(parameters, ...) =
    errno-cookie;
type-specifiers ... name(parameters, ...) =
    number;
type-specifiers ... name(parameters, ...)
    { ... }
```

where *parameters* can be:

```
type-specifiers ... name
const type-specifiers ... name
volatile type-specifiers ... name
cookie-member-name
flag-member-name
```

A function statement is a transformation that converts a Linux function call into a BSD/OS function call. Function statements look similar to prototype function declarations and function definitions in C, but they have different meanings.

All of the function statement formats require a return type, a function name and a parameter list. The return type doesn't have to be a transformable type; it may be any C type, including a pointer, as long as all of the type names have been declared. The function name should match a name in the Linux C library. **Transform** uses a database of library symbols to generate all of the aliases for a known symbol, so the simplest version of the symbol name is usually the right one. If the name is identical to the name of a BSD/OS system call, the body of the function may be omitted, in which case **transform** arranges to call the BSD/OS system call automatically. There may be zero or more parameters. Each parameter is either a declaration for a name, or a `cookie` or `flag` member name. Declared parameters look much like they do in C, except that the name of the parameter is mandatory even when the function statement has no body and looks like a C declaration. Here is a simple example:

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

This definition creates a mapping for the `read()` function. It replaces the definitions for the Linux names `read`, `__read` and `__libc_read`. It calls BSD/OS system call number 3 (`SYS_read`) with the given parameters and returns the result. If the return value is -1, it converts the error number in `__bsdi_errno` into a Linux error number in `errno`.

The parameter list may optionally be followed by an assignment or a C escape. An assignment is a shorthand for certain common function bodies. An assignment from a system call name tells **transform** to make a call to the given BSD/OS system call rather than using the name of the function as the name of the system call. An assignment from an `errno` cookie says to return an error condition (-1 for integer valued functions, NULL for pointer valued functions) and set the Linux `errno` variable to the given value (translated to a Linux value). An assignment from a number tells **transform** to make the function return that constant value; it's useful for turning functions into no-ops. Finally, if you provide a C escape, it will be used as the body of the function in C. **Transform** will still look for error returns and translate `errno` unless you mark the function definition with the type qualifier `noerrno`.

The basic point of the transformation language is to allow you to specify transformations of function parameters and return values using transformable type names. If the parameter and return value transformations are sufficient to handle the transformation for the function, then you can generally omit the function body; this is the simplest and most common definition in the transformation language. If you need to do more work, you can write your own function body. Inside the function body there are a few rules that you must follow, which (unfortunately) are not enforced by **transform**, which does not process the C code in the body. Functions must be re-entrant; if they need to allocate memory dynamically, they should do it on the stack using stack variables and/or the `alloca()` function. To make a BSD/OS system call inside a function body, you must do an indirect call through the `__bsdi_syscall()` function. The `__bsdi_syscall()` function works just like the BSD/OS `syscall()` function - it takes a syscall number from `<sys/syscall.h` and a list of parameters, and it performs the corresponding BSD/OS system call. Any helper functions that you provide in C escapes must have names that use the `__bsdi_` prefix so that they do not collide with Linux function names. The `__bsdi_syscall()` function sets the `__bsdi_errno` variable, ***not*** the `errno` variable, which is a Linux variable. **Transform** generates code to translate `__bsdi_errno` to `errno` automatically, so in general it isn't necessary to refer to `__bsdi_errno` explicitly.

There are several interesting features of parameters beyond the obvious ones. Transformable structure pointers are quite special in many ways:

- The `const` keyword means something in addition to the usual C semantics when it is applied to a transformable structure pointer. A `const` transformable structure pointer is an input-only parameter - the structure gets converted from a Linux structure into a BSD/OS structure, copying it from the application's memory space onto the stack; however, no copying or condition is performed on return.
- A `volatile` transformable structure pointer is a read/write parameter - that is, it is transformed both on input and on output, unless there is an error.
- A transformable structure pointer parameter that doesn't have a `const` or `volatile` qualifier is output-only. The BSD/OS system call places its data in a BSD/OS structure allocated on the stack, and that structure is automatically converted on return into the corresponding Linux structure.
- If a structure definition ends with an array and a function definition contains both a transformable structure pointer *and* an integral parameter with whose name consists of the prefix `length_` plus the name of the structure pointer parameter, then the structure is considered to be variable length and it is assumed to have the number of bytes indicated by

the length parameter. The length parameter may also be a pointer to an integral type, in which case it is dereferenced before it is used. The value of the length parameter is used when converting the array member (as described above) and it is also passed to the structure's transformation functions, if it has any.

Some system calls like `ioctl()` and `fcntl()` change their parameter types or their return types depending on the value of a "command" or flag parameter. The transformation language allows you to define each of these variants separately. You simply specify a `cookie`or `flag` member name for a particular parameter, and if the function is called with that parameter matching that value, then the body of that function definition is executed. The first matching definition applies. You must always supply a *generic* function definition that uses the appropriate `cookie` or `flag` type for that parameter, and the body of that function definition is executed when the `cookie` or `flag` value fails to match any of the specific values in other function definitions for the same function. The feature is hard to describe in words but easy to show in examples; here's one:

```
cookie int linux_pers_t { PERS_LINUX 0; };
int personality(PERS_LINUX) = 0;
int personality(linux_pers_t p) = EINVAL;
```

This code defines a `cookie` that lists "personality" values for the Linux `personality()` system call. We only support the Linux personality, so only the `PERS_LINUX` member is interesting. If the application calls `personality(0)`, the definition for `personality(PERS_LINUX)` matches, and the system call appears to return 0. If the application calls `personality()` with any other value for the personality parameter, the system call will appear to return -1 and `errno` will be set to the Linux equivalent of `EINVAL`.

Here is a somewhat more complex example:

```
flag int openflags_t { ... };
cookie int fcntl_t { ... };
struct flock { ... };
openflags_t fcntl(int fd, F_GETFL, int ignore);
int fcntl(int fd, F_SETFL, openflags_t oflags);
int fcntl(int fd, F_GETLK, struct flock *fl);
int fcntl(int fd, F_SETLK,
    const struct flock *fl);
int fcntl(int fd, F_SETLKW,
    const struct flock *fl);
int fcntl(int fd, fcntl_t cmd, int arg);
```

In this example, if the second parameter matches `F_GETFL`, then the first `fcntl()` definition applies; it converts the BSD/OS `open()` mode flags into Linux mode flags on return. If the second parameter is `F_SETFL`, the third parameter is converted from Linux mode flags into BSD/OS mode flags before we call the BSD/OS `fcntl()` system call. The third, fourth and fifth definitions show how a transformable structure pointer parameter is converted on output (third) and input (fourth and fifth, respectively). The generic function definition causes all remaining `fcntl()` cookie values to be passed unchanged to the BSD/OS `fcntl()` system call; this is appropriate when the BSD/OS cookie value is identical to the Linux cookie value and the parameters and return value do not require transformation, or when the application supplies an illegal cookie value that the BSD/OS `fcntl()` call can reject. Note that there is really just one `fcntl()` function in the **liblinux** library - all the `fcntl()` definitions are merged into a single function.

The __kernel_ feature

Function names that begin with `__kernel_` are treated specially. LAP has support for raw Linux system call traps. By default, when it detects a Linux system call trap, LAP marshals its arguments and transfers control to the function with the same name as the Linux kernel call. Sometimes it isn't appropriate to do this - for example, the kernel system call may have a different name from the function in the Linux C library, or it may treat its parameters differently. In that case, you may define a function with the `__kernel_` prefix to handle just system call traps.

For example, the Linux `llseek()` system call has a different kernel interface from the Linux C library interface:

```
linux_loff_t llseek(int fd, linux_loff_t offset,
    int whence) { ... }
int __kernel__llseek(int fd, unsigned long o_high,
    unsigned long o_low, linux_loff_t *result,
    int whence) { ... }
```

The kernel version of this function swaps the high and low words of the offset and returns its value using a reference parameter, unlike the C library version.

5. The implementation

5.1. The transformation compiler

The **transform** program compiles transformation sources and creates C output files. The program is about 3,300 lines of C, Yacc and Lex source code. An additional program named **afdb** builds databases for **transform**; it's about 250 lines of C.

The **transform** program is a single-pass compiler. It parses statements and then emits most code in place, including C escapes. `Typedef` and `struct` declarations are converted into C `typedef` and `struct` declarations, respectively, while `cookie` and `flag` members become C `#define` directives. `In()` and out() transformation functions become static inline functions with names that begin with the type name (for typedef types), with the type name plus _default (for cookie or flag types) or with the tag name (for structures); the function names end in _in() and _out(), respectively. Cookie and flag transformations turn into switch statements or sequences of if statements (respectively), inside inline functions, with a call to the *type*_default_in() or *type*_default_out() function at the end, as appropriate. Function statements become C inline functions with numeric suffixes to distinguish the different alternatives. At the end of processing, the compiler emits static functions which test incoming arguments and call the appropriate function alternatives. The compiler also generates assembly escapes that serve to map the static container functions onto the names that the Linux C library uses. (The inline function and assembly escape syntax are extensions in the GNU C compiler.)

The **afdb** program processes the dynamic symbol table from the Linux C library and produces Berkeley DB btree files that map addresses to function names and function names to addresses. When the transformation compiler sees a function name, it looks the name up in the database, locates all of the aliases, then generates assembly escapes that duplicate the Linux aliases in the emulation library.

5.2. The transformations

Currently there are about 3250 lines of code written in the transformation language.

Most of the transformations are straightforward. Many functions require only the default transformation, with no transformable parameters or transformable result and no function body; in that case we just make the equivalent BSD/OS system call, and transform the BSD/OS *errno* value if there is an error. A number of functions apply very simple transformations on parameters. Some functions perform minor API changes; for example, Linux has two kernel interfaces for the select() function, one of which takes different parameters from the standard API, and one of which has a different name, and the transformation language serves to map parameters correctly.

A few transformations are more complex. The ioctl() transformations are large because Linux ioctl() cookies and termio/termios structures are different from BSD/OS, even though the semantics are very similar. For socket functions, the Linux kernel provides a single system call that multiplexes all of the BSD-style socket calls using cookies, so the socket support is a little bit complicated. The getdirentries() and getdents() transformations are complicated because Linux dirent structures have seek-offset members that are not in BSD/OS dirent structures, and because the different sizes of the Linux and BSD/OS dirent structures require code to repack them. A similar issue with re-packing applies to getgroups() and setgroups(), which require a separate kernel implementation because the Linux kernel gid_t data type is a 16-bit integer while the BSD/OS type is a 32-bit integer. (The problem doesn't strike the C library API for getgroups() and setgroups() because the GNU C library that Linux uses has a 32-bit uid_t type like BSD/OS. There are a few cases like this where the GNU C library API is closer to the BSD/OS API than to the Linux kernel API and we try to take advantage of this when we can.)

5.3. The libraries

We build three shared library objects: the dynamic linker, the emulation library and a dummy C library.

The *dynamic linker* is built from the GNU C library source code and linked with transformation language source code so that it can make native BSD/OS system calls. We configure the dynamic linker slightly differently from Linux so that it looks for its ld.so.cache file in /linux/etc rather than /etc, which causes it to use different libraries from the native BSD/OS dynamic linker.

We make the *emulation library* from the transformation sources plus some assembly and C code. We supply code to do call tracing at the Linux API level. We add code to implement BSD/OS system call stubs without polluting the Linux C library namespace. We add initialization code that programs the hardware interrupt descriptor table so that Linux system call interrupts are dispatched to an address in the emulation library, and we generate a dispatch table that sends system call interrupts to the appropriate handler. The transformation compiler itself generates stub code for each system call that marshals arguments and calls the C transformation function; the dispatch table jumps to the stubs. We build the dispatch table using an **awk** script that processes the Linux header file that defines Linux system call numbers.

The *dummy C library* replaces the Linux C library in the Linux library path. The purpose of the dummy C library is to load the emulation library ahead of the real Linux C library in the symbol search path, so that no matter how an application tries to load the Linux C library, it will always get the emulation library too. We use the **ELF** DT_AUXILIARY feature to implement this trick. The dummy C library would not have any code of its own if it were not for a peculiar rule about library initialization. It seems that the dynamic linker initializes libraries in reverse order of their loading; that means that it initializes the emulation library after it initializes the real Linux C library. But we have to arrange to dispatch Linux system call interrupts before we can execute any code from the real Linux C library, so the emulation library needs to run its initialization first. The dummy C library initialization is performed before both the real Linux C library initialization and the emulation library initialization, however, so we get around this problem by arranging for the dummy C library to call an initialization function in the emulation library.

5.4. How does it really work?

It's hard to tell how the emulation really works just by reading descriptions of its pieces. Here's a brief description of what happens when you actually run a program.

When a Linux application starts up on BSD/OS, the normal **ELF** loader in the operating system loads it with the modified Linux dynamic linker. The application uses the dynamic linker to load the shared libraries that it needs. All of the shared libraries that it sees are real Linux shared libraries, with one exception: when the application asks for the Linux C library, it also gets an emulation library.

Let's say the application needs a service from the Linux C library; for example, it does a `stat()` call to find out the size of a file. The application doesn't define `stat()` itself, so the dynamic linker looks for an implementation of `stat()`. Because of the way the libraries were loaded, the dynamic linker looks in the emulation library before it looks in the Linux C library, and it uses the emulation library's `stat()` function. The emulation function allocates room on the stack for a BSD/OS `stat` structure and calls the BSD/OS `stat()` function with the stack buffer as an argument. If the BSD/OS `stat()` call succeeds, the emulation function copies and converts the elements of the buffer into the Linux `stat` structure that was passed in and returns 0 for success. If the `stat()` call failed, the emulation function converts the BSD/OS error number into a Linux error number and stores the result in the Linux `errno` location.

Current Linux shared C libraries are statically linked internally, so a `stat()` call inside the C library works a little differently. The Linux C library moves the system call number (106) and the two parameters into registers and executes the Intel `int $0x80` instruction to generate a software interrupt. The Intel hardware transfers control directly to a dispatch routine in the emulation library. The dispatch code performs a computed goto using the system call number, resulting in a branch to the automatically generated stub for `stat()` in the emulation library. This code pushes the parameters on the stack and calls the same `stat()` emulation function that the application used in the example above. On return, if there was an error, the stub code copies the negated `errno` value back into the result register.

(This example oversimplifies the specific situation with `stat()` slightly - see the appendix for more details.)

6. Conclusions

6.1. Comparisons to other work

Of course there are many ways to emulate other operating systems and Linux on other Unix-like systems in particular. I want to mention a couple other emulations done in a different style, and compare them to LAP.

- The Skunkworks folks at **SCO** have a very neat emulator that they call **lxrun**. **Lxrun** is a program that loads a Linux program into its address space and catches the `SIGSEGV` signal that the SCO Unix operating system sends to the program when the Linux code executes a software interrupt instruction (`int $0x80`). This is analogous to the way that LAP redirects the hardware interrupt descriptor table, but it uses unprivileged software instead, so it requires no changes to the kernel at all, although it's a little slower. **Lxrun** can handle statically linked programs as well as the obsolete Linux **a.out** executable format, unlike LAP, and it requires no changes at all to the dynamic linker and no futzing with libraries. It's a really lightweight implementation. LAP improves on it by reducing the overhead of software interrupts, reducing overhead again by interposing the library interface to system calls when possible, and by loading itself automatically rather than requiring a separate loader program (at the expense of modifying the dynamic linker to use BSD/OS system calls). LAP's transformation language should also make it easier to maintain.
- The **FreeBSD** project decided to implement Linux emulation in its kernel. All of the transformations are performed in privileged mode, and the memory for the emulation is dedicated. This is a heavyweight implementation in terms of the amount of code required and its effect on the kernel, but it does permit precise emulation of (for example) signal semantics. While this is nice, I feel that operating system kernels are already absurdly fat, and given that LAP can be reasonably complete and efficient operating outside the kernel, that's a virtue. Certainly the transformation-driven approach to emulation could be applied to an in-kernel emulation if we felt that it would be useful.

Another out-of-kernel approach that we could have taken was the microkernel plus OS server approach that was used in Mach 3 and later versions of Mach [Golu90]. In Mach, not just the libraries but entire machine-independent part of the operating system runs outside the kernel, and the emulation communicates with the kernel using IPC calls. That strategy would clearly be overkill for a Linux emulation on a BSD Unix system, however, since Linux and BSD are so similar at the API level.

It's worth pointing out that while the transformation compiler was written in such a way to make it easy to retarget for (say) FreeBSD, I would not consider it a flaw if it were never retargeted. It's nice to be able to generalize tools, but LAP benefits from using a little language regardless of whether it is general.

6.2. General results

The emulation is quite successful. We can run a number of interesting Linux applications, and they run quite efficiently. Among the programs we have tested are the Adobe Acrobat Reader v4, Netscape Communicator v4.7, and WordPerfect v8. (In fact, this document was composed using Netscape Composer for Linux running under BSD/OS.) Only very minor BSD/OS kernel modifications were required, and the kernel contains no emulation code itself, so we avoided any significant kernel bloat. The implementation is remarkably robust so far; we have had to make very few bug fixes after the initial coding and testing. I attribute this to the small size and the simplicity of the specification.

As far as performance goes, LAP seems to be more than adequate, but the impact is difficult to measure in a meaningful way. I thought about trying to measure some application running under native Linux and comparing it to the same application running under LAP on native BSD/OS on the same hardware, but the different kernels, filesystems and other factors would surely confound the result - it would be more of a measure of Linux versus BSD than the overhead of LAP. However, I can provide a vague idea of how much time is spent in LAP when running a program. I ran an instruction tracer on the Linux **ls** program running `ls -l` under LAP and counted the number of instructions that were executed in the range of addresses occupied by LAP. For an 8-item directory listing, LAP used 3423 instructions out of 714797 total, or about 0.5%. Because of the way that LAP interposes itself in front of the Linux C library, the impact is actually a bit less than it seems, because LAP replaced code that would have been executed in the **ls** program under Linux. When running an X-based program under LAP such as Netscape (as I am doing right now), the overhead is not perceptible.

6.3. Bugs, omissions and other niceties

Not all Linux system calls are currently emulated. The sign that a system call hasn't been emulated is that your application prints a message and aborts. Almost all of the missing system calls are administrative calls, however, so we suspect that we won't encounter them in third-party applications. (For example, we don't support Linux NFS daemons; the BSD/OS native NFS daemons work just fine.)

By far the biggest user-visible omission is the lack of support for the Linux `clone()` system call and related user thread support. I am working actively on this issue and I hope to have news to report at the conference.

We don't support statically linked Linux programs. If we wanted to support statically linked Linux programs, we would adopt the SCO emulation technology. The kernel would load a statically-linked version of the emulation library into every statically linked Linux program. The emulation library would not interpose itself in front of Linux C library functions, but it would still catch software interrupts and process them in the same way that LAP currently does. Statically linked Linux programs are sufficiently rare that we have not seen a need for this feature yet.

Currently LAP does not do any mapping of data types that have narrower widths on Linux. If a UID on BSD is greater than 65,536, a LAP program may see a truncated value for that UID instead of the full 32-bit value in some situations.

I have not described the LAP support for older Linux **ELF** programs that use the 5th version of the Linux C library (sometimes called **libc5**). We do support those programs and the **libc5** emulation shares most of its source code with the rest of LAP. It works a little differently from the scheme described in this document, but I'm not going to explain it here. LAP does not (and probably will never) support Linux **a.out** programs.

The code for LAP is available on the BSD/OS contributed software CD-ROM. Like all software on that CD-ROM, it is freely redistributable. Feel free to use it and modify it as you please with the usual understanding that if it doesn't work for you or causes problems for you, BSDI doesn't take any responsibility.

Appendix - an extended example

types.xh:

```
/*      BSDI $Id: types.xh,v 1.2 1999/04/14 22:42:57 prb Exp $  */

/*
 * Linux has an awkward problem which we have to worry about here...
 *
 * The GNU C library defines basic types that don't match Linux kernel types.
 * The library applies transformations to these types when passing them
 * to and from the Linux kernel. By convention, the transformation routines
 * call stubs named __syscall_*() to perform the syscall using Linux
 * kernel data types. It's stupid to transform these data types twice,
 * especially when the GNU C data types are generally wider and hence
 * closer to BSD types, so we interpose our transformation routines over
 * the GNU C library routines rather than the Linux __syscall_() routines.
 */

%{
#include <sys/types.h>
%}

/*
 * Here are the GNU C library types.
 */
typedef unsigned long long dev_t {
        in(dev) { return (makedev(dev>> 8, dev & 0xff)); }
        out(dev) { return (major(dev) << 8 | minor(dev)); }
};

typedef char *caddr_t;
typedef long clock_t;
typedef unsigned int gid_t;
typedef unsigned long ino_t;
typedef int key_t;
typedef long long linux_loff_t;
typedef unsigned int mode_t;
typedef unsigned int nlink_t;
typedef long off_t;
typedef int pid_t;
typedef int ptrdiff_t;
typedef unsigned int size_t;
typedef int ssize_t;
typedef long time_t;
typedef unsigned int uid_t;

/*
 * For documentation purposes, here are the actual Linux internal types,
 * where they differ from the GNU C library types.
 */

typedef unsigned short linux_kernel_dev_t;
typedef unsigned short linux_kernel_gid_t;
typedef unsigned short linux_kernel_mode_t;
typedef unsigned short linux_kernel_nlink_t;
typedef unsigned short linux_kernel_uid_t;

typedef unsigned int u_int;
typedef unsigned short u_short;
typedef unsigned long u_long;
```

stat.xh:

```
/*      BSDI $Id: stat.xh,v 1.2 1999/04/14 22:38:59 prb Exp $   */

/*
 * Transforms for stat.h.
 */
```

```
%{
/* Don't use timespecs. */
#define _POSIX_SOURCE           1
#include <sys/stat.h
#undef _POSIX_SOURCE
#define old_stat        stat
%}

cookie int linux_stat_ver_t {
        _STAT_VER_LINUX_OLD     1;
        _STAT_VER_SVR4          2;
        _STAT_VER_LINUX         3;
};

struct stat {
        dev_t st_dev;
        unsigned short linux_pad1;
        ino_t st_ino;
        mode_t st_mode;
        nlink_t st_nlink;
        uid_t st_uid;
        gid_t st_gid;
        dev_t st_rdev;
        unsigned short linux_pad2;
        off_t st_size;
        unsigned long st_blksize;
        unsigned long st_blocks;
        time_t st_atime;
        long linux_unused1;
        time_t st_mtime;
        long linux_unused2;
        time_t st_ctime;
        long linux_unused3;
        long linux_unused4;
        long linux_unused5;
};

struct old_stat {
    unsigned short int st_dev;
    unsigned short int linux_pad1;
    unsigned long int st_ino;
    unsigned short int st_mode;
    unsigned short int st_nlink;
    unsigned short int st_uid;
    unsigned short int st_gid;
    unsigned short int st_rdev;
    unsigned short int linux_pad2;
    unsigned long int st_size;
    unsigned long int st_blksize;
    unsigned long int st_blocks;
    unsigned long int st_atime;
    unsigned long int linux_unused1;
    unsigned long int st_mtime;
    unsigned long int linux_unused2;
    unsigned long int st_ctime;
    unsigned long int linux_unused3;
    unsigned long int linux_unused4;
    unsigned long int linux_unused5;
};
```

stat.x:

```
/*      BSDI $Id: stat.x,v 1.2 1999/04/14 22:47:01 prb Exp $    */

/*
 * Transformation rules for <sys/stat.h> syscalls.
 */
```

```
include "types.xh"
include "stat.xh"

/*
 * Linux's kernel types don't match its user types.
 * The GNU C library performs transformations from the kernel types
 * to the user types. We interpose the GNU C library stubs rather
 * than the Linux kernel stubs, so that we don't transform twice.
 */
int __syscall_stat(const char *name, struct old_stat *buf)
{
        return (__bsdi_syscall(SYS_stat, name, buf));
}

int _xstat(_STAT_VER_LINUX_OLD, const char *name, struct old_stat *buf)
{
        return (__bsdi_syscall(SYS_stat, name, buf));
}
int _xstat(_STAT_VER_LINUX, const char *name, struct stat *buf)
{
        return (__bsdi_syscall(SYS_stat, name, buf));
}
int _xstat(linux_stat_ver_t v, const char *name, struct stat *buf) = EINVAL;

int _fxstat(_STAT_VER_LINUX_OLD, int fd, struct old_stat *buf)
{
        return (__bsdi_syscall(SYS_fstat, fd, buf));
}
int _fxstat(_STAT_VER_LINUX, int fd, struct stat *buf)
{
        return (__bsdi_syscall(SYS_fstat, fd, buf));
}
int _fxstat(linux_stat_ver_t v, int fd, struct stat *buf) = EINVAL;

int _lxstat(_STAT_VER_LINUX_OLD, const char *name, struct old_stat *buf)
{
        return (__bsdi_syscall(SYS_lstat, name, buf));
}
int _lxstat(_STAT_VER_LINUX, const char *name, struct stat *buf)
{
        return (__bsdi_syscall(SYS_lstat, name, buf));
}
int _lxstat(linux_stat_ver_t v, const char *name, struct stat *buf) = EINVAL;

/*
 * The other stat.h calls...
 */

int chmod(const char *path, mode_t mode);

int fchmod(int fd, mode_t mode);

int mkdir(const char *path, mode_t mode);

int umask(mode_t m);
```

References

[ANSI89] American National Standard for Information Systems, *Programming Language - C*, ANSI X3.159-1989.

[BSDI00] http://www.bsdi.com/

[Free00] *Using and Porting GNU CC, for Version 2.95*, R. Stallman, Free Software Foundation, 2000.

[Ging89] "Shared Libraries in SunOS," R. Gingell, M. Lee, X. Dang, M. Weeks, in *Proceedings of the Summer 1989 Usenix Conference*, 1989.

[Golu90] "Unix as an application program," D. Golub, R. Dean, A. Forin, R. Rashid, in *Proceedings of the Summer 1990 Usenix Conference*, June 1990.

[IEEE96] IEEE Std 1003.1, 1996 Edition, *Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C language]*, Institute for Electrical and Electronics Engineers, 1996.

[John75] ''Yacc - Yet Another Compiler Compiler,'' S. Johnson, *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill NJ, 1975.

[John79] ''A Tour through the Portable C Compiler,'' S. Johnson, *Unix Programmer's Manual*, 7th Edition, volume 2b, AT&T Bell Laboratories, Murray Hill NJ, 1979.

[Lesk75] ''Lex - a lexical analyzer generator,'' M. Lesk, *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill NJ, 1975.

[Linu00] http://www.linux.org/

[Olso99] ''Berkeley DB,'' M. Olson, K. Bostic, M. Seltzer, in *Proceedings of the Freenix Track*, 1999 Usenix Annual Technical Conference, June 1999.

[Reco98] ''Linux Emulation for SCO,'' R. Record, M. Hopkirk, S. Ginzburg, in Proceedings of the *Usenix 1998 Annual Technical Conference: Invited Talks and Freenix Track*, June 1998.

[Ritc79] ''A Tour through the UNIX C Compiler,'' D. Ritchie, *Unix Programmer's Manual*, 7th Edition, volume 2b, AT&T Bell Laboratories, Murray Hill NJ, 1979.

[Salu98] *Handbook of Programming Languages, Volume III: Little Languages and Tools.*P. Salus, ed. Macmillan Technical Publishing, 1998.

[Unix90] *System V Application Binary Interface*, Unix Software Operation, Prentice-Hall, 1990.