Linux Preload Library Support

Thomas J. Moore

Revision 182

Abstract

This document describes and implements the support required for simple Linux preload libraries, in particular for emulating a filesystem in user space. It also includes a sample application: an extremely simplistic filesystem which returns random data on reads and verifies that data on writes.

This document was generated from the following sources, all of which are attached:

```
$Id: build.nw 183 2012-11-26 07:59:32Z darktjm $
$Id: ldpreload.nw 182 2012-11-26 07:23:23Z darktjm $
```

Contents

1 Introduction

The most efficient and compatible method for creating a filesystem is to write a driver for each supported operating system and kernel. If only one operating system or kernel is to be supported, this is not necessarily the most difficult process, either. However, there are a number of reasons one might wish to do this in user space, instead. One is portability: user-space interfaces tend to change much more slowly than kernel-space interfaces. In fact, completely different operating systems may have similar user-space interfaces, making the porting job much easier. Another is system stability: errors in kernel interfaces tend to affect the system much more profoundly than in user interfaces, even in layered operating systems. This makes it less dangerous to

initially develop and debug the filesystem. Finally, user-space filesystems can be run by ordinary users, rather than just root.

There are a number of reasonable approaches to user-space filesystems. One is the FUSE project[1], which provides a consistent and short API, and is guaranteed to work with any application which accesses it, as the system will provide fallbacks and other necessary changes needed to support new functionality not covered by the API. Another is to use library call overrides: any attempt to access a file using the overridden functions is intercepted and managed by the overrider instead of the system library. This can be done either using source code redefinitions, which require recompiling the code, or using the operating system's mechanism for dynamic library loading to override the symbol bindings. Either way, the advantages over FUSE are the elimination of any system call overhead, and the ability to run even in systems that do not support FUSE due to administrator policy or lack of support. It is also possible to provide more services than just filesytem overrides, such as authentication overrides. The disadvantages are recompilation in the case of redefinition-type overrides, inability to use with setuid binaries in the case of dynamic overrides, inability to override implicit calls within the system libraries, and an unlimited API, leaving many potential holes where system calls will be used instead of an appropriate override.

In spite of the disadvantages, this document only describes tools to assist in creating the latter type: dynamic library overrides, specifically for Linux, to be used with the LD_PRELOAD facility. There are plenty of override libraries (especially memory debuggers) for other operating systems as well, but concentrating on Linux will keep this short. At least I'm not limiting this to ia32 architectures on a limited set of distributions, as most commercial Linux support does. Instead, any systems meeting the minimum prerequisites will work.

This document was built using a literate programming support package. To build, place the NoWeb source files into their own directory (optionally extracting them from the document you are reading first), extract the makefile, and make using GNU make:

```
# if your PDF viewer supports attachments, save the attachment
# otherwise, use pdfdetach:
pdfdetach -saveall ldpreload.pdf
# or pdftk:
pdftk ldpreload.pdf unpack_files output .
# then extract the makefile and build
notangle -t8 build.nw > makefile
make install
```

If NoWeb is not available, but perl is, the following code can be copied and pasted into a text file using a reasonable document viewer and made executable as an imperfect, but adequate `notangle` replacement:

```
#!/bin/sh
#!perl
# This code barely resembles noweb's notangle enough to work for simple builds
eval 'exec perl -x -f "$0" "$@"'
  if 0;
my $chunk = '*';  my $tab = 0;

while($#ARGV > 0 && $ARGV[0] =~ /^-.*/) {
  if($ARGV[0] =~ /^-R/) {
    $chunk = $ARGV[0];
    $chunk =~ s/-R//;
  } elsif($ARGV[0] =~ /^-t(.*)/) {
    $tab = $1;
  }
  shift @ARGV;
}
my ($inchunk, %chunks);
while(<>) {
  if(/^<<(.*)>>=$/) {
```

[1] http://fuse.sourceforge.net/

```
      $inchunk = $1;
    } elsif(/^@( |$)/) {
      $inchunk = "";
    } elsif($inchunk ne "") {
      $chunks{$inchunk} .= $_;
    }
  }
  my $chunkout = $chunks{$chunk};
  while($chunkout =~ /(?:^|\n)(|[^\n]*[^\n@])(?:<<((?:[^>\n]|>[^>\n])*)>>)/) {
    my $cv = $chunks{$2};
    my $ws = $1;
    $cv =~ s/\n$//;
    $ws =~ s/\S/ /g;
    $cv =~ s/\n/$&$ws/g;
    $chunkout =~ s/(^|[^@])(<<(([^>\n]|>[^>\n])*)>>)/$1$cv/;
  }
  $chunkout =~ s/@(<<|>>)/\1/g;
  $chunkout =~ s/((^|\n)\t*) {$tab}/$1\t/g if($tab gt 0);
  print $chunkout;
```

Additional build configuration can be done in `makefile.config` before installing (in particular, the install locations). This file can be generated using either `make makefile.config` or `make -n`. On the other hand, to avoid having to modify this file after cleaning, `makefile.config.local` can be created for this purpose instead.

3a ⟨*(build.nw) Sources* 3a⟩≡

```
$Id: ldpreload.nw 182 2012-11-26 07:23:23Z darktjm $
```

3b ⟨*(build.nw) Version Strings* 3b⟩≡ (8e)

```
"$Id: ldpreload.nw 182 2012-11-26 07:23:23Z darktjm $\n"
```

3c ⟨*(build.nw) Common NoWeb Warning* 3c⟩≡ (5a 8e)

```
# $Id: ldpreload.nw 182 2012-11-26 07:23:23Z darktjm $
```

2 Creating override wrappers

The utilities described here will only support programming in C, and overriding functions in the standard C library. Since the overrides should only cover accesses to the virtual filesystem, and still allow some access to normal filesystems, the original functions well need to be callable as well. This requires the original symbol definition, obtained by calling `dlsym` on a reference to the original library. This definition needs to be obtained once for every overridden symbol, and stored somewhere. It could be done in bulk at library initialization, or once before attempting to call the original function, or at every single function call. For this implementation, the second approach will be used. For example, to call the original definition of `open`, assuming a handle to the standard C library has been stored in the global/static variable `libc_dll`, the following code can be used:

3d ⟨*Example* `open` *call* 3d⟩≡

```
int libc_open(const char *path, int flags, int mode)
{
  static void *sym = NULL;
  pthread_mutex_lock(&lock);
  if(!sym && !(sym = dlsym(libc_dll, "open"))) {
    pthread_mutex_unlock(&lock);
    fprintf(stderr, "Can't find libc's open\n");
    exit(1);
  }
  pthread_mutex_unlock(&lock);
  return ((int (*)(const char *path, int flags, int mode))sym)(path, flags, mode);
}
```

The code also assumes a global lock called `lock` to allow this call to work in threaded applications. The code can still be used without threading support using `#define`. The locking can also be temporarily disabled by adding a guard and parameter, if desired.

Of course the above code makes some assumptions about error handling, but they are pretty safe in most cases. The main problem would be if a user were to override `exit`, to which the answer is, "Don't do that!" Writing such a stub function for every overridden function can be tedious, so instead a `sed` script is provided to generate these functions. The function's prototype is all that needs to be declared; the script will insert the function right after it, or replace the function when run twice. A companion script strips those functions out. The prototypes are declared using a specially formatted comment at the beginning of a line: the first character is a plus sign, and a full prototype with parameter names is declared. Care must be taken to insert a blank line after it if and only if the generated function is not in place yet.

4a ⟨*Example* `open` *prototype* 4a⟩≡

```
/*+ int open(const char *path, int flags, int mode) */
```

4b ⟨*(build.nw) makefile.vars* 4b⟩≡ 7d▷

```
C_POSTPROCESS += | sed -f addldovr.sed
```

4c ⟨*(build.nw) makefile.rules* 4c⟩≡ 7e▷

```
$(CFILES) : addldovr.sed
```

4d ⟨*(build.nw) Plain Files* 4d⟩≡

```
addldovr.sed \
delldovr.sed \
```

4e ⟨*addldovr.sed* 4e⟩≡

```
⟨For each prototype comment 5a⟩ {
  ⟨Convert prototype comments to stub 5b⟩
}
```

4f ⟨*delldovr.sed* 4f⟩≡

```
⟨For each prototype comment 5a⟩ {
  ⟨Remove stubs after prototype comments 5c⟩
}
```

For each prototype, the prototype comment itself is printed first. The prototype is then saved for processing after stripping off the comment characters. No nested or following comments are permitted. Then, the existing stub, if any, is skipped by searching for the next blank line. All text within the existing stub is deleted. For the deletion script, this is all that needs to be done.

5a ⟨*For each prototype comment* 5a⟩≡ (4)

```
# Process C files for LD_PRELOAD wrappers
# e.g. /*+ int open(const char *path, int flags, int mode) */
# will have function libc_open() added/replaced using libc_dll to call libc
# function is locked using global mutex "lock"

⟨(build.nw) Common NoWeb Warning 3c⟩

/^\/\*+.*(/
```

5b ⟨*Convert prototype comments to stub* 5b⟩≡ (4e) 5e▷

```
⟨Find and strip next prototype comment 5d⟩
```

5c ⟨*Remove stubs after prototype comments* 5c⟩≡ (4f)

```
⟨Find and strip next prototype comment 5d⟩
```

5d ⟨*Find and strip next prototype comment* 5d⟩≡ (5)

```
p
s/... *//;s/ *\*\/.*//
h
# Find next blank line
:b
  N
  s/.*\n//
  /^..*$/bb
```

To start the new stub, the saved prototype is printed with the `libc_` prefix. In addition, a new first parameter to support disabling the locks is provided.

5e ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁5b 5f▷

```
# spit out prototype
g
s/^/static /
s/\([a-zA-Z0-9_]* *(\)/libc_\1int with_locking, /
s/, *)/)/
p
```

The first bit of the stub is the same for all functions. The only special bit is to use the actual name of the function when printing a message or loading the symbol.

5f ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁5e 6a▷

```
i{
i \  static void *sym = NULL;
i \  if(with_locking) pthread_mutex_lock(&lock);
g
s/[^(]* \([^( ]*\) *(.*/  if(!sym \&\& !(sym = dlsym(libc_dll, "\1"))) {/
p
i \    if(with_locking) pthread_mutex_unlock(&lock);
g
```

5f ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁5e 6a▷

```
s/[^(]* \([^( ]*\) *(.*/    fprintf(stderr, "Can't find libc's \1\\n");/
p
i \    exit(1);
i \  }
i \  if(with_locking) pthread_mutex_unlock(&lock);
```

The next part is to call the dynamic symbol. That means casting the symbol to a function pointer, and then calling that function using the same parameters passed into the stub. The return value from the call is then returned from the stub.

To perform the cast, the function name is stripped out, and the type (all text appearing before the function name) is printed, followed by the function pointer dereference ([(*)]), followed by the verbatim contents of the prototype. Some compilers may warn about the use of variable names in prototypes, but that issue is too complex to deal with for this simple script.

6a ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁5f 6b▷

```
g
s/\([^(]* [^A-Za-z_]*\)[^( ]* *\((.*)\)/  return ((\1(*)\2)sym)(/
```

If the function has no return value, the return statement is replaced by a standalone statement. There is no more code in this function, so there is no need for an explicit return.

6b ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁6a 6c▷

```
s/return ((void (/((void (/
```

To pass the parameters on, an assumption is made about the parameters: none of the parameters are function pointers, arrays, or variable-length argument lists. Since this covers the vast majority of functions, and handling variable-length argument lists in a generic way is nearly impossible, no attempts will be made to improve these restrictions. What the restrictions mean is that the argument name is always the last word before a comma or parenthesis, meaning that all text other than that word and its following comma or parenthesis can be used verbatim. The only special case is a single parameter named `void`, which is actually an indication of the lack of parameters.

6c ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁6b 6d▷

```
x
s/[^(]*(//;s/).*/)/
s/[^,]* [^A-Za-z0-9_, ]*\([a-zA-Z0-9_]* *[,)]\)/\1 /g
s/^void)/)/
# paste it all together and tack on a semicolon
x
G
s/\n//
s/ *$/;/
```

That's all there is to the function. The only remaining thing to add is a blank line, as the sed code above will strip out the line between prototypes.

6d ⟨*Convert prototype comments to stub* 5b⟩+≡ (4e) ◁6c

```
a}
a
```

In order to use these stubs, the standard C library must be opened. To do this, a few more static variables are used; it is up to the caller of the initialization function to ensure thread safety. This is not done on every stub call mainly to allow other initialization to be intermingled.

The library can either be opened explicitly, which requires an absolute file name, or implicitly, which requires support for the `RTLD_NEXT` special symbol. This is very likely supported, although GNU libc requires `_GNU_SOURCE` to be defined in order to obtain that definition. The `RTLD_NEXT` also supports multiple, stacked library overrides, which the explicit form does not.

7a ⟨*Dynamic Override C Includes* 7a⟩≡ (8e) 9c▷

```
/* for RTLD_NEXT */
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
⟨(build.nw) Common C Includes (imported)⟩
#include <dlfcn.h>

/* Note: the libc_* functions are auto-(re)generated from the + comments */
```

7b ⟨*Dynamic Override Globals* 7b⟩≡ (8f)

```
static int did_dl_init = 0;
#ifndef RTLD_NEXT
static void *libc_dll = NULL;
#else
#define libc_dll RTLD_NEXT
#endif
```

7c ⟨*Dynamic Override Init* 7c⟩≡ (9f)

```
if(did_dl_init)
  return;
did_dl_init = 1;
#ifndef RTLD_NEXT
if(!(libc_dll = dlopen("/usr/lib/libc.so", RTLD_LAZY))) {
  perror("Can't open libc");
  exit(1);
}
#endif
```

Finally, in order to use this, the library must be built appropriately. Like with the executables, it is assumed that the primary object code resides in an object file with the same name as the target library. No support is provided for making versioned libraries.

7d ⟨*(build.nw) makefile.vars* 4b⟩+≡ ◁4b 7f▷

```
EXTRA_CFLAGS += -fPIC -Wall -Wshadow -Wwrite-strings -g
EXTRA_LDFLAGS += -ldl
```

7e ⟨*(build.nw) makefile.rules* 4c⟩+≡ ◁4c

```
lib: $(SHLIB_FILES)
bin: lib
$(SHLIB_FILES) : $(LOCAL_LIB_FILES)
%.so: %.o
	$(CC) -shared -o $@ $< $(LDFLAGS) -L. $(LOCAL_LIBS) $(EXTRA_LDFLAGS)
```

7f ⟨*(build.nw) makefile.vars* 4b⟩+≡ ◁7d

```
SHLIB_FILES = ⟨Shared Libraries 8d⟩
```

8a ⟨*(build.nw) makefile.config* 8a⟩≡

```
# Installation directory for libraries
LIB_DIR=/usr/local/lib
```

8b ⟨*(build.nw) Install other files* 8b⟩≡

```
mkdir -p $(LIB_DIR)
for x in $(SHLIB_FILES); do \
  rm -f $(LIB_DIR)/$$x; \
  cp -p $$x $(LIB_DIR); \
done
# ldconfig
```

8c ⟨*(build.nw) Clean built files* 8c⟩≡

```
rm -f $(SHLIB_FILES)
```

3 Example Random Number Filesystem

As an example, a simple filesystem that generates random numbers on reads and generates only errors on writes, and then only if the written data does not match what the random numbers should be. The file name is used as a seed for the random number generator. Listings, attributes, and other such things are not supported. Any code which appears to be reusable for similar projects will be placed in Dynamic Override code chunks.

8d ⟨*Shared Libraries* 8d⟩≡ (7f)

```
randfs.so \
```

8e ⟨*Dynamic Override Header* 8e⟩≡ (8f)

```
/*
  ⟨(build.nw) Common NoWeb Warning 3c⟩
*/
⟨Dynamic Override C Includes 7a⟩
static const char version_id[] = ⟨(build.nw) Version Strings 3b⟩;
```

8f ⟨*randfs.c* 8f⟩≡

```
⟨Dynamic Override Header 8e⟩
⟨randfs Includes 8g⟩

⟨randfs Globals 9a⟩
⟨Dynamic Override Globals 7b⟩

#include "cproto.h"
⟨randfs Functions 9f⟩
```

Since this library may be used for threaded code, the locking will not be disabled. If it were, a few `#defines` would suffice.

8g ⟨randfs *Includes* 8g⟩≡ (8f) 17d▷

```
#include <pthread.h>
#ifdef THRDBG
#define pthread_mutex_lock(x) do { \
  fprintf(stderr, "lock @%d\n", __LINE__); \
```

8g ⟨randfs *Includes* 8g⟩≡ (8f) 17d▷

```
  pthread_mutex_lock(x); \
  fprintf(stderr, "got lock @%d\n", __LINE__); \
} while(0)
#define pthread_mutex_unlock(x) do { \
  fprintf(stderr, "unlock @%d\n", __LINE__); \
  pthread_mutex_unlock(x); \
} while(0)
#endif
```

9a ⟨randfs *Globals* 9a⟩≡ (8f) 9d▷

```
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

9b ⟨*Disable Dymaic Override Threading* 9b⟩≡

```
#define pthread_mutex_lock(x)
#define pthread_mutex_unlock(x)
```

The first thing that needs to be supported is opening of files. Since the `open` function takes a variable number of arguments, the arguments must be extracted for the real function; as stated above, variable arguments are not supported on the libc call.

9c ⟨*Dynamic Override C Includes* 7a⟩+≡ (8e) ◁7a 10a▷

```
#include <sys/file.h>
#include <stdarg.h>
```

9d ⟨randfs *Globals* 9a⟩+≡ (8f) ◁9a 14a▷

```
⟨Dynamic Override FS Globals 10b⟩
```

9e ⟨*Dynamic Override FS* `open` 9e⟩≡ (16b)

```
/*+ int open(const char *path, int flags, int mode) */

int open(const char *path, int flags, ...)
{
  int mode = 0;
  va_list arg;

  if(flags & O_CREAT) {
    va_start(arg, flags);
    mode = va_arg(arg, int);
    va_end(arg);
  }
  ⟨Override open 10d⟩
}
```

9f ⟨randfs *Functions* 9f⟩≡ (8f) 10f▷

```
static void libc_init(void)
{
  ⟨Dynamic Override Init 7c⟩
  ⟨randfs Init 10c⟩
}

⟨Dynamic Override FS Functions 16b⟩
```

The file name is used to distinguish virtual files from real ones, using a regular expression (specified using an environment variable) match on the fully expanded file name. This means that an expensive canonicalization of the file name must be performed before every open.

10a ⟨*Dynamic Override C Includes* 7a⟩+≡ (8e) ◁9c

```
#include <regex.h>
```

10b ⟨*Dynamic Override FS Globals* 10b⟩≡ (9d) 14c▷

```
static regex_t path_regex;
static regmatch_t *regex_matches = NULL;
static int num_regex_matches = 0;
```

10c ⟨`randfs` *Init* 10c⟩≡ (9f) 22d▷

```
const char *pat = getenv("RANDFS_PATH_PATTERN");

if(!pat)
  pat = "^/rand/";
/* regex_matches is unused */
int err = regcomp(&path_regex, pat, REG_EXTENDED | REG_NOSUB);
if(err) {
  size_t buflen = regerror(err, &path_regex, NULL, 0);

  {
    char buf[buflen + 1];

    regerror(err, &path_regex, buf, buflen + 1);
    fprintf(stderr, "Invalid RANDFS_PATH_PATTERN: %s\n", buf);
    exit(1);
  }
}
```

10d ⟨*Override* `open` 10d⟩≡ (9e) 15a▷

```
⟨If not in randfs path 10e⟩
  return libc_open(1, path, flags, mode);
```

10e ⟨*If not in* `randfs` *path* 10e⟩≡ (10d 21c 23 25a)

```
pthread_mutex_lock(&lock);
libc_init();
pthread_mutex_unlock(&lock);
const char *rn = expand_path_name(path);
if(!rn)
  return -1;
if(regexec(&path_regex, rn, num_regex_matches, regex_matches, 0))
```

The canonicalization cannot be done using `realpath`, because `realpath` only works with real, existing files. Even if this were a shim over a real filesystem, it wouldn't work for writing new files. For this reason, a simple function which strips out `.` and `..` and expands symbolic links is provided.

10f ⟨`randfs` *Functions* 9f⟩+≡ (8f) ◁9f 15c▷

```
⟨Function to Expand Paths 10g⟩
```

10g ⟨*Function to Expand Paths* 10g⟩≡ (10f)

```
⟨Expand Paths Support 11a⟩

static const char *expand_path_name(const char *orig)
{
  int l = strlen(orig);
  if(!l)
    return orig;
  ⟨Expand Path orig 11d⟩
}
```

Rather than allocate a new buffer every time, a single buffer is maintained and grown to store the expanded file name. This means that locks will have be used outside of the function to make it thread safe. The buffer is expanded in small chunks, which are probably big enough for most names, so there is no need to use geometric buffer length expansion. The initial length needs to be able to at least contain the original path.

11a ⟨*Expand Paths Support* 11a⟩≡ (10g) 11c▷

```
#ifndef EXPBUF_GROW
#define EXPBUF_GROW 80
#endif
static char *expand_buf = NULL;
static int expand_buf_len = 0;
```

11b ⟨*Set Expand Buffer Length to* l 11b⟩≡ (11d)

```
if(expand_buf_len < l + 1) {
  int newlen = l / EXPBUF_GROW + 1;
  if(expand_buf) {
    free(expand_buf);
    expand_buf_len = 0;
  }
  expand_buf = malloc(newlen * EXPBUF_GROW);
  if(!expand_buf)
    return NULL;
  expand_buf_len = newlen * EXPBUF_GROW;
}
```

If the path is relative, the current working directory needs to be added as well. Otherwise, the initial expanded path is empty. It cannot be assumed that the path returned by `getcwd` is already in canonical form, so another buffer is used to create a temporary absolute path name. Retrieving the current working directory for every file may be expensive if the underlying filesystem does not cache it or otherwise retrieve it quickly, but every call may be done in a different directory and there is no quick way to tell.

11c ⟨*Expand Paths Support* 11a⟩+≡ (10g) ◁11a 12b▷

```
static char *expand_extra_buf = NULL;
static int expand_extra_buf_len = 0;
```

11d ⟨*Expand Path* orig 11d⟩≡ (10g) 12c▷

```
if(*orig != '/') {
  if(!expand_extra_buf) {
    expand_extra_buf = malloc(EXPBUF_GROW);
    if(!expand_extra_buf)
      return NULL;
    expand_extra_buf_len = EXPBUF_GROW;
  }
  while(!getcwd(expand_extra_buf, expand_extra_buf_len)) {
    if(errno != ERANGE)
      return NULL;
```

11d ⟨*Expand Path* `orig` 11d⟩≡ (10g) 12c▷

```
    errno = 0;
    ⟨Grow expand_extra_buf 12a⟩
  }
  int el = strlen(expand_extra_buf);
  if(!el || expand_extra_buf[el - 1] != '/')
    expand_extra_buf[el++] = '/';
  if(el + l >= expand_extra_buf_len) {
    int newlen = (l + el) / EXPBUF_GROW + 1;
    char *exbuf_new = realloc(expand_extra_buf, newlen * EXPBUF_GROW);
    if(!exbuf_new)
      return NULL;
    expand_extra_buf = exbuf_new;
    expand_extra_buf_len = newlen * EXPBUF_GROW;
  }
  strcpy(expand_extra_buf + el, orig);
  orig = expand_extra_buf;
  l += el;
}
⟨Set Expand Buffer Length to l 11b⟩
```

12a ⟨*Grow* `expand_extra_buf` 12a⟩≡ (11d 13a)

```
{
  char *exbuf_new = realloc(expand_extra_buf, expand_extra_buf_len + EXPBUF_GROW);
  if(!exbuf_new)
    return NULL;
  expand_extra_buf = exbuf_new;
  expand_extra_buf_len += EXPBUF_GROW;
}
```

Now the function simply loops over each element of the provided path, adding it if necessary and expanding links. In order to prevent link loops, a counter is initialized outside the loop and updated at each link expansion.

12b ⟨*Expand Paths Support* 11a⟩+≡ (10g) ◁11c

```
#ifndef LINK_FOLLOW_MAX
#define LINK_FOLLOW_MAX 100
#endif
```

12c ⟨*Expand Path* `orig` 11d⟩+≡ (10g) ◁11d

```
int maxlinks = LINK_FOLLOW_MAX;
int el = 0;
const char *curp, *nxtp = orig;

while(*nxtp) {
  if(expand_buf_len < el + 2) {
    ⟨Grow expand_buf 13a⟩
  }
  expand_buf[el++] = '/';
  while(*nxtp == '/')
    nxtp++;
  if(!*nxtp)
    break;
  curp = nxtp;
  nxtp = strchr(curp, '/');
  if(!nxtp)
    nxtp = curp + strlen(curp);
  ⟨Process next path element to be expanded 13b⟩
}
/* finally, terminate with 0 */
```

12c ⟨*Expand Path* `orig` 11d⟩+≡ (10g) ◁11d

```
if(el == expand_buf_len) {
  ⟨Grow expand_buf 13a⟩
}
expand_buf[el] = 0;
return expand_buf;
```

13a ⟨*Grow* `expand_buf` 13a⟩≡ (12c 13e)

```
#define expand_extra_buf expand_buf
#define expand_extra_buf_len expand_buf_len
⟨Grow expand_extra_buf 12a⟩
#undef expand_extra_buf
#undef expand_extra_buf_len
```

A single dot is usally the current directory. Standard UNIX ensures that it is a true directory before being ignored, but for speed and simplicity it will always be ignored here.

13b ⟨*Process next path element to be expanded* 13b⟩≡ (12c) 13c▷

```
if((nxtp - curp) == 1 && *curp == '.')
  continue;
```

A similar comment holds true for double dots: the previous element is stripped off of the canonical path without first checking that it really is a directory.

13c ⟨*Process next path element to be expanded* 13b⟩+≡ (12c) ◁13b 13d▷

```
if((nxtp - curp) == 2 && *curp == '.' && curp[1] == '.') {
  while(el > 0 && expand_buf[--el] != '/');
  ++el;
  continue;
}
```

Anything else is tacked on verbatim.

13d ⟨*Process next path element to be expanded* 13b⟩+≡ (12c) ◁13c 13e▷

```
/* others add a new path element */
if((nxtp - curp) + el >= expand_buf_len) {
  int newlen = ((nxtp - curp) + el) / EXPBUF_GROW + 1;
  char *exbuf_new = realloc(expand_buf, newlen * EXPBUF_GROW);
  if(!exbuf_new)
    return NULL;
  expand_buf = exbuf_new;
  expand_buf_len = newlen * EXPBUF_GROW;
}
memcpy(expand_buf + el, curp, (nxtp - curp));
el += (nxtp - curp);
```

After any partial path has been generated, soft links are expanded. Even with a virtual file system, the appropriate function to call should be `readlink`; all errors returned are assumed to be due to the fact that the path isn't really a symbolic link. This is not always correct, but it is probably close enough for this application. If a link is found, a new original is created just as for the current working directory expansion. It uses the same buffer, so it needs to move the original aside if it is already coming out of that buffer. Admittedly, sometimes my tricks for avoiding additional memory allocations are probably not worth the effort.

13e ⟨*Process next path element to be expanded* 13b⟩+≡ (12c) ◁13d

```
expand_buf[el] = 0;
int llen;
while (1) {
  llen = readlink(expand_buf, expand_buf + el + 1, expand_buf_len - el - 1);
  if (!maxlinks && llen >= 0) {
    errno = ELOOP;
    return NULL;
  }
  if (llen < expand_buf_len - el - 1)
    break;
  ⟨Grow expand_buf 13a⟩
}
if (llen >= 0) {
  --maxlinks;
  /* got a link expansion: restart from top using expanded link */
  int lexplen = llen + strlen(curp) + 1;
  if (lexplen > expand_extra_buf_len) {
    lexplen = lexplen / EXPBUF_GROW + 1;
    char *lbuf = expand_extra_buf ? realloc(expand_extra_buf, lexplen * EXPBUF_GROW) :
                                    malloc(lexplen * EXPBUF_GROW);
    if (!lbuf)
      return NULL;
    if (orig == expand_extra_buf) {
      curp = lbuf + (curp - orig);
      orig = lbuf;
    }
    expand_extra_buf = lbuf;
    expand_extra_buf_len = lexplen * EXPBUF_GROW;
  }
  if (orig == expand_extra_buf)
    memmove(expand_extra_buf + llen, curp, strlen(curp) + 1);
  else
    strcpy(expand_extra_buf + llen, curp);
  memcpy(expand_extra_buf, expand_buf + el + 1, llen);
  curp = expand_extra_buf;
  /* POSIX says !llen undefined; I say cwd (aka ignore) */
  if (llen && *curp == '/')
    el = 0;
  else {
    while (el > 0 && expand_buf[--el] != '/');
    ++el;
  }
  maxlinks--;
  continue;
}
```

Once open, the file needs to be tracked, so a dynamic array of file descriptors is kept. While sorting would make handling a large number of files easier, there is little advantage for very few files.

14a ⟨`randfs` *Globals* 9a⟩+≡ (8f) ◁9d 22c▷

```
struct local_fdesc {
  int fd;
  ⟨Local File Descriptor 15b⟩
};
```

14b ⟨*(build.nw) Known Data Types* 14b⟩≡ 19a▷

```
local_fdesc,%
```

14c ⟨*Dynamic Override FS Globals* 10b⟩+≡ (9d) ◁10b

```
/* local_fdesc *must* have an fd that's not likely returned by real open */
static struct local_fdesc *local_files = NULL;
static int num_local_files = 0, max_local_files = 0;
#ifndef NUM_LOCAL_FILES
#define NUM_LOCAL_FILES 8
#endif
```

15a ⟨*Override* `open` 10d⟩+≡ (9e) ◁10d

```
pthread_mutex_lock(&lock);
if(!max_local_files) {
  local_files = malloc(NUM_LOCAL_FILES * sizeof(*local_files));
  if(!local_files) {
    pthread_mutex_unlock(&lock);
    return -1;
  }
  max_local_files = NUM_LOCAL_FILES;
} else if(num_local_files == max_local_files) {
  struct local_fdesc *tmp_local_files =
           realloc(local_files,
                   NUM_LOCAL_FILES * (max_local_files + NUM_LOCAL_FILES) *
                   sizeof(*local_files));
  if(!tmp_local_files) {
    pthread_mutex_unlock(&lock);
    return -1;
  }
  local_files = tmp_local_files;
  max_local_files += NUM_LOCAL_FILES;
}
struct local_fdesc *fdesc = &local_files[num_local_files];
memset(fdesc, 0, sizeof(*fdesc));
if(open_local_file(path, rn, flags, mode, fdesc) < 0) {
  pthread_mutex_unlock(&lock);
  return -1;
}
int ret = local_files[num_local_files++].fd;
pthread_mutex_unlock(&lock);
return ret;
```

For the random filesystem, the local file descriptor includes the random seed. The physical file descriptor is a real one to `/dev/null` in order to keep other functions happy. The length of the file and the seed come from the file name: The seed is a simple hash of the file name, and the length is given as a numerical prefix. If the number is followed by an upper-case K, M, G, or T, then the size is specified in binary kilo-, mega-, giga-, or terabytes instead. The default size is 100MB. When writing to a file, `fstat` and seeks should use the current length of the file, which is tracked separately.

15b ⟨*Local File Descriptor* 15b⟩≡ (14a) 17c▷

```
unsigned long seed;
unsigned long long len;
unsigned long long maxlen;
```

15c ⟨`randfs` *Functions* 9f⟩+≡ (8f) ◁10f 16a▷

```
static int open_local_file(const char *path, const char *real_path,
                           int flags, int mode, struct local_fdesc *fdesc)
{
```

15c ⟨randfs *Functions* 9f⟩+≡ (8f) ◁10f 16a▷

```
  fdesc->fd = libc_open(0, "/dev/null", flags & ~O_CREAT, 0);
  if(fdesc->fd < 0)
    return -1;
  set_finfo(fdesc, path);
  fdesc->maxlen = fdesc->len;
  if(flags & O_TRUNC)
    fdesc->len = 0;
  return 0;
}
```

16a ⟨randfs *Functions* 9f⟩+≡ (8f) ◁15c 17b▷

```
static void set_finfo(struct local_fdesc *fdesc, const char *path)
{
  unsigned long seed = 0;
  const char *pp = strrchr(path, '/');

  if(!pp || !*++pp)
    return;
  if(isdigit(*pp)) {
    fdesc->len = atoll(pp);
    const char *sp = pp;
    while(isdigit(*++sp));
    if(*sp == 'K')
      fdesc->len *= 1024LL;
    else if(*sp == 'M')
      fdesc->len *= 1024LL*1024LL;
    else if(*sp == 'G')
      fdesc->len *= 1024LL*1024LL*1024LL;
    else if(*sp == 'T')
      fdesc->len *= 1024LL*1024LL*1024LL*1024LL;
  } else
    fdesc->len = 100 * 1024 * 1024;
  while(*pp)
    seed = ((seed << 3) ^ *pp++) & 0xffffffff;
  fdesc->seed = seed;
}
```

Practically, though, there may be an `open64` call instead. There are a number of other aliases, but I haven't observed them in use.

16b ⟨*Dynamic Override FS Functions* 16b⟩≡ (9f) 16c▷

```
#ifdef open
#undef open
#define o64
#endif
⟨Dynamic Override FS open 9e⟩
#define open open64
#define libc_open libc_open64
⟨Dynamic Override FS open 9e⟩
#ifndef o64
#undef open
#endif
#undef libc_open
```

To close the file when complete, the local file descriptor is removed. Since a memory move is done on this, make sure that any local file descriptor elements are resistant to memory location changes.

16c ⟨*Dynamic Override FS Functions* 16b⟩+≡ (9f) ◁16b

```
/*+ int close(int fd) */

int close(int fd)
{
  struct local_fdesc lfd;

  ⟨Find local file fdno 17a⟩
    return libc_close(1, fd);
  }
  lfd = local_files[fdno];
  memmove(&local_files[fdno], &local_files[fdno + 1],
          (num_local_files - fdno - 1) * sizeof(*local_files));
  num_local_files--;
  pthread_mutex_unlock(&lock);
  return close_local_file(fd, &lfd);
}
```

17a ⟨*Find local file* fdno 17a⟩≡ (16c 19–22 24b 25b)

```
pthread_mutex_lock(&lock);
int fdno;
for(fdno = 0; fdno < num_local_files; fdno++)
  if(local_files[fdno].fd == fd)
    break;
if(fdno >= num_local_files) {
  pthread_mutex_unlock(&lock);
```

17b ⟨randfs *Functions* 9f⟩+≡ (8f) ◁16a 18a▷

```
static int close_local_file(int fd, struct local_fdesc *fdesc)
{
  libc_close(1, fdesc->fd);
  ⟨Clean up randfs file descriptor 17e⟩
  return 0;
}
```

While the file is open, reads and writes need to be intercepted. The reads and writes will come out of pre-filled random number buffers. The buffers are large and unevenly sized so that it is not likely that the repetition of buffer contents will affect the need for tests to be mostly random. The advantage of serving out of buffers in this way is that seeks are efficient, and the overall speed is much better than `/dev/urandom`. The disadvantage is that the buffers need to be allocated before transfers can begin.

17c ⟨*Local File Descriptor* 15b⟩+≡ (14a) ◁15b 18d▷

```
unsigned char *buf;
```

Rather than generating the buffers on the fly, the buffers are filled completely on the first read or write. This can cause a significant delay if the buffer size is too large, but it eliminates the need to track how much of the buffer has been filled to date.

17d ⟨randfs *Includes* 8g⟩+≡ (8f) ◁8g 21b▷

```
#define BUFSIZE (17*1024*1024)
```

17e ⟨*Clean up* randfs *file descriptor* 17e⟩≡ (17b)

```
if(fdesc->buf)
  free(fdesc->buf);
```

18a ⟨randfs *Functions* 9f⟩+≡ (8f) ◁17b 18b▷

```
static unsigned char *get_rand_buf(struct local_fdesc *lfd)
{
  unsigned char *rbuf = lfd->buf;
  if(!rbuf) {
    rbuf = lfd->buf = malloc(BUFSIZE);
    if(!rbuf)
      return rbuf;
    struct drand48_data drstate;
    srand48_r(lfd->seed, &drstate);
    int i;
    /* lrand48() is supposed to produce 32-bit random numbers */
    /* use unsigned int to keep it 32 instead of long's possible 64 */
    unsigned int *bp = (unsigned int *)lfd->buf;
    for(i = 0; i < BUFSIZE / 4; i++) {
      long rn;
      lrand48_r(&drstate, &rn);
      *bp++ = (unsigned long)rn;
    }
  }
  return rbuf;
}
```

For any type of reads, the data is simply copied repeatedly out of the random source.

18b ⟨randfs *Functions* 9f⟩+≡ (8f) ◁18a 18c▷

```
static void fill_buf(const unsigned char *rbuf, unsigned char *buf,
                     int soff, size_t n)
{
  while(n > BUFSIZE - soff) {
    memcpy(buf, rbuf + soff, BUFSIZE - soff);
    n -= BUFSIZE - soff;
    buf += BUFSIZE - soff;
    soff = 0;
  }
  memcpy(buf, rbuf + soff, n);
}
```

For any kind of writes, the data is simply compared repeatedly with the random source.

18c ⟨randfs *Functions* 9f⟩+≡ (8f) ◁18b 19b▷

```
static int cmp_buf(const unsigned char *rbuf, const unsigned char *buf,
                   int soff, size_t n)
{
  while(n > BUFSIZE - soff) {
    if(memcmp(buf, rbuf + soff, BUFSIZE - soff))
      return -1;
    n -= BUFSIZE - soff;
    buf += BUFSIZE - soff;
    soff = 0;
  }
  if(!n)
    return 0;
  return memcmp(buf, rbuf + soff, n);
}
```

The standard reads and writes do not receive offsets, so they will need to track offsets manually.

18d ⟨*Local File Descriptor* 15b⟩+≡ (14a) ◁17c

```
off64_t curoff;
```

19a ⟨*(build.nw) Known Data Types* 14b⟩+≡ ◁14b

```
off64_t,ssize_t,%
```

19b ⟨`randfs` *Functions* 9f⟩+≡ (8f) ◁18c 19c▷

```
/*+ ssize_t read(int fd, void *buf, size_t n) */

ssize_t read(int fd, void *buf, size_t n)
{
  ⟨Find local file fdno 17a⟩
    return libc_read(1, fd, buf, n);
  }
  struct local_fdesc *lfd = &local_files[fdno];
  unsigned char *rbuf = get_rand_buf(lfd);
  if(!rbuf) {
    pthread_mutex_unlock(&lock);
    return -1;
  }
  pthread_mutex_unlock(&lock); /* note: this could crash now on close */
  if(lfd->curoff + n > lfd->len)
    n = lfd->len - lfd->curoff;
  fill_buf(lfd->buf, buf, lfd->curoff % BUFSIZE, n);
  lfd->curoff += n;
  return n;
}
```

19c ⟨`randfs` *Functions* 9f⟩+≡ (8f) ◁19b 19d▷

```
/*+ ssize_t write(int fd, const void *buf, size_t n) */

ssize_t write(int fd, const void *buf, size_t n)
{
  ⟨Find local file fdno 17a⟩
    return libc_write(1, fd, buf, n);
  }
  struct local_fdesc *lfd = &local_files[fdno];
  unsigned char *rbuf = get_rand_buf(lfd);
  if(!rbuf) {
    pthread_mutex_unlock(&lock);
    return -1;
  }
  pthread_mutex_unlock(&lock); /* note: this could crash now on close */
  if(lfd->curoff + n > lfd->maxlen) {
    errno = ENOSPC;
    return -1;
  }
  if(cmp_buf(rbuf, buf, lfd->curoff % BUFSIZE, n)) {
    errno = EIO;
    return -1;
  }
  lfd->curoff += n;
  if(lfd->len < lfd->curoff)
    lfd->len = lfd->curoff;
  return n;
}
```

Speaking of offsets, seeks must be supported as well. This means not only `lseek`, but `pread` and `pwrite` as well.

19d ⟨randfs *Functions* 9f⟩+≡ (8f) ◁19c 20a▷

```
/*+ off_t lseek(int fd, off_t offset, int whence) */

off_t lseek(int fd, off_t offset, int whence)
{
  ⟨Find local file fdno 17a⟩
    return libc_lseek(1, fd, offset, whence);
  }
  ⟨Set rand seek offset 20b⟩
}
```

20a ⟨randfs *Functions* 9f⟩+≡ (8f) ◁19d 20c▷

```
/*+ off64_t lseek64(int fd, off64_t offset, int whence) */

off64_t lseek64(int fd, off64_t offset, int whence)
{
  ⟨Find local file fdno 17a⟩
    return libc_lseek64(1, fd, offset, whence);
  }
  ⟨Set rand seek offset 20b⟩
}
```

20b ⟨*Set rand seek offset* 20b⟩≡ (19d 20a)

```
struct local_fdesc *lfd = &local_files[fdno];
if(whence == SEEK_SET)
  lfd->curoff = offset;
else if(whence == SEEK_CUR)
  lfd->curoff += offset;
else if(whence == SEEK_END)
  lfd->curoff = lfd->len - offset;
else {
  errno = EINVAL;
  return -1;
}
if(lfd->curoff < 0)
  lfd->curoff = 0;
else if(lfd->curoff > lfd->maxlen)
  lfd->curoff = lfd->maxlen;
if(lfd->curoff > lfd->len)
  lfd->len = lfd->curoff;
pthread_mutex_unlock(&lock); /* note: this could crash now on close */
return lfd->curoff;
```

20c ⟨randfs *Functions* 9f⟩+≡ (8f) ◁20a 21a▷

```
/*+ ssize_t pread(int fd, void *buf, size_t n, off_t off) */

ssize_t pread(int fd, void *buf, size_t n, off_t off)
{
  ⟨Find local file fdno 17a⟩
    return libc_pread(1, fd, buf, n, off);
  }
  struct local_fdesc *lfd = &local_files[fdno];
  unsigned char *rbuf = get_rand_buf(lfd);
  if(!rbuf) {
    pthread_mutex_unlock(&lock);
    return -1;
  }
  pthread_mutex_unlock(&lock); /* note: this could crash now on close */
  if(off > lfd->len) {
    errno = EINVAL;
    return -1;
```

20c ⟨randfs *Functions* 9f⟩+≡ (8f) ◁20a 21a▷

```
  }
  if(off + n > lfd->len)
    n = lfd->len - off;
  fill_buf(rbuf, buf, off % BUFSIZE, n);
  return n;
}
```

21a ⟨randfs *Functions* 9f⟩+≡ (8f) ◁20c 21d▷

```
/*+ ssize_t pwrite(int fd, const void *buf, size_t n, off_t off) */

ssize_t pwrite(int fd, const void *buf, size_t n, off_t off)
{
  ⟨Find local file fdno 17a⟩
    return libc_pwrite(1, fd, buf, n, off);
  }
  struct local_fdesc *lfd = &local_files[fdno];
  unsigned char *rbuf = get_rand_buf(lfd);
  if(!rbuf) {
    pthread_mutex_unlock(&lock);
    return -1;
  }
  pthread_mutex_unlock(&lock); /* note: this could crash now on close */
  if(off + n > lfd->maxlen) {
    errno = ENOSPC;
    return -1;
  }
  if(cmp_buf(rbuf, buf, off % BUFSIZE, n)) {
    errno = EIO;
    return -1;
  }
  if(lfd->len < off + n)
    lfd->len = off + n;
  return n;
}
```

Since some utilities might want to know the file and access permissions for the pseudo files, the `stat` and `access` functions need to be supported as well. For these, the modification time is set at library initialization time.

21b ⟨randfs *Includes* 8g⟩+≡ (8f) ◁17d 24d▷

```
#include <sys/stat.h>
```

21c ⟨randfs `lstat` *Function* 21c⟩≡ (21d)

```
/*+ int lstat(const char *path, struct statbuf *buf) */

int lstat(const char *path, struct statbuf *buf)
{
  ⟨If not in randfs path 10e⟩
    return libc_lstat(1, path, buf);
  struct local_fdesc lfd;

  memset(&lfd, 0, sizeof(lfd));
  set_finfo(&lfd, path);
  fill_stat(&lfd, buf);
  return 0;
}
```

21d ⟨randfs *Functions* 9f⟩+≡ (8f) ◁21a 22b▷

```
#define statbuf stat
⟨randfs lstat Function 21c⟩
#define lstat stat
#define libc_lstat libc_stat
⟨randfs lstat Function 21c⟩
#undef statbuf
#undef lstat
#undef libc_lstat
#define statbuf stat64
#define fill_stat fill_stat64
#define lstat lstat64
#define libc_lstat libc_lstat64
⟨randfs lstat Function 21c⟩
#undef lstat
#undef libc_lstat
#define lstat stat64
#define libc_lstat libc_stat64
⟨randfs lstat Function 21c⟩
#undef statbuf
#undef fill_stat
#undef lstat
#undef libc_lstat
```

22a ⟨randfs fstat *Function* 22a⟩≡ (22b)

```
/*+ int fstat(int fd, struct stat *buf) */

int fstat(int fd, struct stat *buf)
{
  ⟨Find local file fdno 17a⟩
    return libc_fstat(1, fd, buf);
  }
  fill_stat(&local_files[fdno], buf);
  pthread_mutex_unlock(&lock);
  return 0;
}
```

22b ⟨randfs *Functions* 9f⟩+≡ (8f) ◁21d 23a▷

```
⟨randfs fstat Function 22a⟩
#define fstat fstat64
#define libc_fstat libc_fstat64
#define stat stat64
#define fill_stat fill_stat64
⟨randfs fstat Function 22a⟩
#undef fstat
#undef libc_fstat
#undef stat
#undef fill_stat
```

22c ⟨randfs *Globals* 9a⟩+≡ (8f) ◁14a

```
static time_t modtime;
static int devno;
```

22d ⟨randfs *Init* 10c⟩+≡ (9f) ◁10c

```
devno = modtime = time(NULL);
```

22e ⟨randfs fill_stat 22e⟩≡ (23a)

```
static void fill_stat(struct local_fdesc *fd, struct stat *buf)
{
  memset(buf, 0, sizeof(*buf));
  buf->st_dev = devno;
  buf->st_ino = fd->seed;
  buf->st_mode = 0777 | S_IFREG;
  buf->st_nlink = 1;
  buf->st_uid = geteuid();
  buf->st_gid = getegid();
  buf->st_size = fd->len;
  buf->st_blksize = 1024;
  buf->st_blocks = (fd->len + 1023) / 1024;
  buf->st_atime = buf->st_mtime = buf->st_ctime = modtime - 10;
}
```

23a ⟨randfs *Functions* 9f⟩+≡ (8f) ◁22b 23b▷

```
⟨randfs fill_stat 22e⟩
#define fill_stat fill_stat64
#define stat stat64
⟨randfs fill_stat 22e⟩
#undef stat
#undef fill_stat
```

23b ⟨randfs *Functions* 9f⟩+≡ (8f) ◁23a 24a▷

```
/*+ int access(const char *path, int mode) */

int access(const char *path, int mode)
{
  ⟨If not in randfs path 10e⟩
    return libc_access(1, path, mode);
  if(mode == F_OK)
    return 0;
#if 0 /* should be true on directories, and false elsewhere */
  if(mode & X_OK) {
    errno = EACCES;
    return -1;
  }
#endif
  return 0;
}
```

In the real world, though, the external symbols required by `stat` calls are mangled by the headers to an appropriate environment-specific version. This is very specific to GNU libc.

23c ⟨randfs xstat *Function* 23c⟩≡ (24a)

```
/*+ int __xstat(int ver, const char *path, struct statbuf *buf) */

int __xstat(int ver, const char *path, struct statbuf *buf)
{
  ⟨If not in randfs path 10e⟩
    return libc___xstat(1, ver, path, buf);
  struct local_fdesc lfd;

  memset(&lfd, 0, sizeof(lfd));
  set_finfo(&lfd, path);
  fill_stat(&lfd, buf);
  return 0;
}
```

24a ⟨randfs *Functions* 9f⟩+≡ (8f) ◁23b 24c▷

```
#ifdef __xstat
#undef __xstat
#endif
#ifdef __lxstat
#undef __lxstat
#endif
#define statbuf stat
⟨randfs xstat Function 23c⟩
#define __xstat __lxstat
#define libc___xstat libc___lxstat
⟨randfs xstat Function 23c⟩
#undef statbuf
#undef __xstat
#undef libc___xstat
#define statbuf stat64
#define fill_stat fill_stat64
#define __xstat __xstat64
#define libc___xstat libc___xstat64
⟨randfs xstat Function 23c⟩
#undef __xstat
#undef libc___xstat
#define __xstat __lxstat64
#define libc___xstat libc___lxstat64
⟨randfs xstat Function 23c⟩
#undef statbuf
#undef fill_stat
#undef __xstat
#undef libc___xstat
```

24b ⟨randfs fxstat *Function* 24b⟩≡ (24c)

```
/*+ int __fxstat(int ver, int fd, struct stat *buf) */

int __fxstat(int ver, int fd, struct stat *buf)
{
  ⟨Find local file fdno 17a⟩
    return libc___fxstat(1, ver, fd, buf);
  }
  fill_stat(&local_files[fdno], buf);
  pthread_mutex_unlock(&lock);
  return 0;
}
```

24c ⟨randfs *Functions* 9f⟩+≡ (8f) ◁24a 25b▷

```
#ifdef __fxstat
#undef __fxstat
#endif
⟨randfs fxstat Function 24b⟩
#define __fxstat __fxstat64
#define libc___fxstat libc___fxstat64
#define stat stat64
#define fill_stat fill_stat64
⟨randfs fxstat Function 24b⟩
#undef __fxstat
#undef libc___fxstat
#undef stat
#undef fill_stat
```

Also, `ls` tries to grab the extended attributes.

24d ⟨randfs *Includes* 8g⟩+≡ (8f) ◁21b

```
#include <attr/xattr.h>
```

25a ⟨randfs getxattr *Function* 25a⟩≡ (25b)

```
/*+ ssize_t getxattr(const char *path, const char *name, void *val, size_t size) */

ssize_t getxattr(const char *path, const char *name, void *val, size_t size)
{
  ⟨If not in randfs path 10e⟩
    return libc_getxattr(1, path, name, val, size);
  errno = ENOATTR;
  return -1;
}
```

25b ⟨randfs *Functions* 9f⟩+≡ (8f) ◁24c

```
⟨randfs getxattr Function 25a⟩
#define getxattr lgetxattr
#define libc_getxattr libc_lgetxattr
⟨randfs getxattr Function 25a⟩
#undef getxattr
#undef libc_getxattr

/*+ ssize_t fgetxattr(int fd, const char *name, void *val, size_t size) */

ssize_t fgetxattr(int fd, const char *name, void *val, size_t size)
{
  ⟨Find local file fdno 17a⟩
    return libc_fgetxattr(1, fd, name, val, size);
  }
  errno = ENOATTR;
  return -1;
}
```

4 Usage

To use the dynamic standard C library override support, add a dependency to this document. It is possible to use the scripts outside of the build system, as well. To do this, place `addldovr.sed` and `delldovr.sed` in appropriate places, and apply `addldovr.sed` before building, and `delldovr.sed` before editing. In either case, add the code in ⟨*Dynamic Override C Includes* 7a⟩ to the top of your source; for outside the build system, the ⟨*(build.nw) Common C Includes* (imported)⟩ can probably be replaced by whatever the code depends on.

For each overridden function, add an override comment, followed by a blank line, as shown in ⟨*Example* `open` *prototype* 4a⟩. Then, create the override function with the same prototype. At the top of this function, an initialization function should be called which executes the code in ⟨*Dynamic Override Init* 7c⟩, which in turn depends on static variables defined in the same file, provided by ⟨*Dynamic Override Globals* 7b⟩. The initialization code uses a static guard variable to execute exactly once, and assumes that it is returning from a `void` function.

To use the example random number filesystem, simply add it to `LD_PRELOAD`, and optionally set the `RANDFS_PATH_PATTERN` environment variable to change the regular expression used to match file names in the virtual random filesystem from the default of `^/rand/`. The base filename determines both the random number seed and the size. The size is determined by a numeric prefix, optionally followed by an upper-case K, M, G, or T to multiply by powers of 1024. If no size can be determined from the file name, the file size is 100MB.

5 Code Index

⟨*Set Expand Buffer Length to* `l` 11b⟩ 11b, 11d
⟨*Set rand seek offset* 20b⟩ 19d, 20a, 20b
⟨*Shared Libraries* 8d⟩ 7f, 8d