Linux Emulation for SCO

Ronald Joe Record (rr@sco.com), **SCO**
Michael Hopkirk (hops@sco.com), **SCO**
Steve Ginzburg (steven@ugcs.caltech.edu), **California Institute of Technology** May 8, 1998

This paper describes some of the rationale and implementation decisions for lxrun, an Intel Linux emulator, plus how to get, build, configure and run it; what some of the current and future development issues and enhancements are and the current status of the project. There is not much difference between the execution environment required by Linux binaries and binaries for other Intel UNIX platforms, the main one being the way in which system calls are handled. For example, in Linux an "int $0x80" instruction is used, which jumps to the system-call-handling portion of the Linux kernel. On SCO systems, "int $0x80" is an unused vector and therefore causes a general protection trap resulting in a SIGSEGV signal. Lxrun intercepts these signals and calls the SCO equivalent of the system call that the Linux program attempted. It also remaps some ioctls, flags, return values and error codes. Using lxrun, a Linux binary can be run on a non-Linux platform with little performance penalty. Lxrun can also take advantage of the lower overhead in some Linux libraries, occasionally resulting in improved performance over native binaries. No kernel modifications are necessary.

Contents

1 Introduction

1.1 What is lxrun ?

Lxrun is an emulator that allows the execution of Intel Linux binaries on Intel UNIX platforms. The currently supported platforms are SCO OpenServer 5 (UNIX SVR3), SCO UnixWare 2.x (UNIX SVR4) and SCO UnixWare 7 (UNIX SVR5).

Lxrun works by remapping system calls on the fly. Since lxrun does its work at the system call level, it requires copies of the Linux dynamic loader (ld-linux.so.*) and whatever Linux shared libraries are required by the program being run. The current development release of lxrun consists of approximately 6000 lines of code (146 Kilobytes).

Most programs that do not rely on Linux-specific quirks or deal directly with hardware should work under lxrun. Users of lxrun have reported success with raplayer (RealAudio client), xquake, the StarOffice suite, gcc (the GNU C compiler), smbclient, the AC3D modeller and a myriad of smaller applications and utilities.

1.2 Why should I use lxrun ?

The original impetus for writing lxrun was to be able to run Netscape Navigator 1.x on SCO OpenServer. When Netscape ported Navigator to SCO platforms, the need was obviated. Later, work was resumed in an attempt to run Adobe Acrobat Reader on OpenServer. The body of system call mappings in lxrun grew gradually as users modified it to work with more and more applications. By this process, lxrun became quite robust, able to handle sophisticated X11, audio, and networking applications.

Lxrun promotes the interoperability of Linux with other UNIX and UNIX-like platforms. This improves the user's ability to combine the best aspects of available operating systems. For example, it allows a user to combine Linux's large freeware application base with SCO's:

- Large robust filesystem support
- Large installed base
- Renowned and Industry-proven scalability
- Enterprise applications base
- Installed Enterprise server deployment
- Network Computer / "Any Client" deployment of applications
- Ability to distribute and execute applications via *Tarantella* `<http://tarantella.sco.com/>`
- Robust DOS/Windows '95 emulation capabilities
- *Free UNIX* `<http://www.sco.com/offers/>` offer makes for extremely inexpensive robust UNIX workstation
- SVR5 enhancements
  - Up to 64GB of main memory
  - 1 TB File Systems and disk partitions
  - 1 TB file sizes
  - Up to 76,800 TB of total storage supported
  - Up to 32 CPUs supported
  - Support for I2O Peripherals
  - 64-bit technology
  - Advanced Clustering Support for up to 4 nodes
  - Hot Pluggable Disk and Tape Drives
  - Disk Mirroring and Spanning
  - Hot Plug PCI
  - Multi-path I/O
  - Real-time Data Management
  - UPS Monitoring
  - Centralized, on-line backup
  - Journaling File Systems for improved data recovery

Every UNIX vendor could construct such a "Why you should use my platform" list. Lxrun allows the user to leverage the large base of precompiled Linux software without restricting the choice of platform.

2 Implementation

The emulator needed to handle the following major issues (detailed below): system call handling along with remapping of arguments, flags, return values and error codes; library pathname lookup and remapping; ioctl mappings for the various major devices and differences in the tty subsystems.

2.1 Design goals

- Allow execution of Intel Linux binaries on other Intel UNIX platforms
- Exist entirely in user-space (no kernel modifications)
- No modification of the Linux binaries or libraries required

2.2 Difference in syscall handling

System call handling is implemented differently between Linux and Intel UNIX System V platforms; that is each uses a different instruction to implement the switch to kernel mode.

Intel Unix System V uses a "lcall $0x07". In Linux an "int $0x80" software interrupt instruction is used, which jumps to the system-call-handling portion of the Linux kernel. On SCO systems, "int $0x80" is an unused vector and therefore causes a general protection trap resulting in a SIGSEGV signal. Lxrun intercepts these signals and calls the SCO equivalent of the system call that the Linux program attempted. In many cases this is a direct map or call of the equivalent native system call, in other cases some mapping or translation of arguments passed in and flags and error codes passed out is required. Where there is no equivalent system call available on the emulated system or the equivalent syscall mapping has not been implemented the system call fails and returns an errorcode of ENOSYS.

2.3 Pathname for loading Linux libraries

Because lxrun works at the system call level, any Linux shared libraries required by the application must be present in the emulation environment. This leads to a possible filename-space conflict between native and Linux binaries. To resolve this problem, lxrun remaps any pathnames beginning with /lib, /usr/lib, /usr/local/lib, or /usr/X11R6/lib by prepending them with a "Linux root" path. This path is specified at compile time and can be overridden by setting the LINUX_ROOT environment variable. This remapping allows the user to install a set of Linux libraries in a separate directory hierarchy from the native system libraries, thus avoiding conflicts.

2.4 Device major number mappings

The arguments to an ioctl call alone do not provide enough information to remap the command number correctly. This is because the same command number can have different meanings to different drivers.

Lxrun works around this problem by maintaining a mapping from open file descriptors to drivers. On the first ioctl call to a new file descriptor, lxrun determines the associated major device number. It compares this with a table of drivers and major device numbers set up at run-time. This mapping is then cached to improve performance on future ioctl calls to that file descriptor. Lxrun can then take the driver into consideration when remapping ioctl command numbers.

2.5 Differences in kernel tty systems

The following is a discourse on lxrun tty handling from Robert Lipe (robertl@dgii.com), the principal author of lxrun's tty handling code:

"The kernel tty systems are very different. The user level tty systems are actually quite similar, fortunately for lxrun. Most of the members of things like termio and struct termios are the same size, alignment, and offset. Even most of the bitfields fall into place. While it would have been much safer to do

```
if ( (lx_tio->c_cflag & LX_CBAUD) == LX_B50)
        tio->c_cflag |= B50;
if (lx_tio->c_iflag & LX_ICANON)
        tio->c_iflag |= ICANON;
```

and repeat this for each of a couple hundred flags, the reality was that it generated horrible code that would have performed poorly and been a nightmare to maintain.

The only sticky spot was that Linux has four distinctly separate members in c_cc (the control character array) for VMIN, VTIME, EOF, and EOL. Most System V's including OpenServer (I can't recall what SVR[45] does here) use the same offset and multiplex this into two bytes. In reality, since they can never both be active at the same time (if ICANON is set, EOF and EOL are used. If ICANON is clear, VMIN, VTIME are used) this doesn't turn out to be much of a issue.

[Since] TCGETA and TCSETA are by far the most frequently used, I implemented them first (and their derivatives that wait and flush and the derivatives of each of those for POSIX struct termios). With those 8 in hand, I started firing up applications that were known to do wierd things to the tty. When elvis (the Linux vi binary I had at the time) worked well enough, I submitted the changes. Then, I just used more and more applications and picked up a few stragglers like FIONREAD that mapped very simply.

Are there hazards in any of this? Certainly. Each system has a few bits in each of the available ioctls that don't exist in the other system. There is some overlap. We haven't seen any real world failures becuase of this. The reality seems to be that the programs that drive serial ports to the crazy edge don't make sense to emulate anyway. For example, someone once asked me why Linux ecu (a kermit-like program) didn't work well. Since source for it is available and it supports OpenServer just fine, use the native binary instead. Someone once asked me about running Linux ppp, but that's similarly nonsensical, though doomed to failure for different reasons."

2.6 Miscellaneous issues

- For local displays, SCO OpenServer uses ":0.0" and UnixWare uses "unix:0.0". The former causes Linux X11 binaries to try to use shared memory, which won't work with the native X server. The latter causes a name lookup for the machine "unix" which probably fails. These two cases are detected and the DISPLAY is set to contain a valid machine name instead.
- Documentation is supplied in the `doc` subdirectory of the lxrun source. This currently consists of this document, a FAQ, a document describing how to get the Linux StarOffice suite to run, and the system call mapping table.
- Various packaging conveniences have been supplied. For instance, the necessary Linux shared libraries have been re-packaged in the native installation format. A shell script, lxfront, useful in the execution of Linux a.out and statically-linked ELF binaries, has been supplied. To use lxfront:

- Put lxfront and any Linux a.out or statically-linked ELF binaries in /usr/local/linux/bin
- Put lxrun in /usr/local/bin
- Create links in /usr/local/bin to lxfront with the same names as your Linux binaries.

Executing these links should invoke lxrun on your Linux program.

3 Development Issues

3.1 Direct execution of Linux ELF binaries

Rather than running Linux binaries with the lxrun front-end, it is possible to turn lxrun into a program interpreter, ld-linux.so.* - the Linux runtime linker. The path of the interpreter is actually embedded in every dynamic linked ELF executable - for standard UnixWare / OpenServer ELF it is /usr/lib/libc.so.1, and for Linux it is /lib/ld-linux.so.1 (or, ld-linux.so.2 for executables linked with GNU libc v2).

The direct execution of Linux binaries in this manner is in the latest development releases of lxrun. At the time of this writing, support for this model of execution is available on UnixWare platforms only. SCO OpenServer places some rather strict limitations on what a program interpreter can be. Overcoming these restrictions is a current development topic.

The following discourse, from Mike Davidson `(md@sco.com)` the author of lxrun, details some of the issues involved in the direct execution model:

"It's actually quite simple to build a version of lxrun which can be used as the initial program interpreter for Linux binaries. There are, however, a couple of problems:

- runtime linkers are usually ELF shared objects which don't have a fixed load address, and which have to do some rather delicate data relocations when they first start up - I avoided this issue by making the ld-linux.so.* version of lxrun be an actual ELF executable (rather than a shared library) bound to a fixed address which does not conflict with the addresses used by normal Linux binaries - while this is a bit of a kludge it works perfectly well
- the interpreter program loaded by the kernel cannot itself require an interpreter - what this means is that if the program interpreter wants to do any dynamic loading of shared libraries it has to do it for itself - this isn't too much of a problem on OpenServer since you can just do a static link of everything that lxrun needs into a single binary - unfortunately this doesn't work on UnixWare 7 since libsocket is *only* available as a shared library.

I think that there is a way round this, but I haven't had time to try it out yet. Essentially it looks something like this:

- ld-linux.so.* is created as a dynamic linked shared object with appropriate dependencies on other shared libraries (ie at least /usr/lib/libc.so.1 and /usr/lib/libsocket.so.1)
- when ld-linux.so.* is loaded by the kernel and gets control, the *first* thing that it does is to fix up enough of it's own data relocations in order to be able to run
- once that is done, it maps /usr/lib/libc.so.1 (ie the normal system runtime linker and standard library) into memory, fakes up a suitable aux vector and invokes the normal runtime linker, while pretending

to be a normal executable program (this involves a lot of trickery - you have to fake up an appropriate set of program headers to give to the runtime linker, you have to fix up entries in .dynamic, .dynsym and .rel.* to reflect the actual address that ld-linux.so.* is actually loaded at, and you have to provide a fake entry point address so that the system runtime linker will give control back to ld-linux.so.* in the right place ....)

- if all of that works, then ld-linux.so.* proceeds as normal, and now maps in the Linux runtime linker and passes control to it

I realise that this all sounds hideously complicated, but all of the alternatives that I can think of are worse in some way.

At first I thought that we could avoid this by just porting a version of the Linux runtime linker to run native on UNIX, but this looks like it may be more trouble than it is worth - while the Linux runtime linker is quite well written it has to deal with some rather unpleasant limitations in the GNU tools - in particular it looks as if GNU ld doesn't support the equivalent of our -Bsymbolic option which makes writing the startup code for a runtime linker almost impossible (in fact the Linux runtime linker startup routine is just one massive function that uses nothing but local variables and which does all of it's system calls with chunks of inline assembler)."

3.2 SIGSEGV vs. software interrupt kernel module

Consideration was given to implementing a software interrupt kernel module rather than relying on the "int $0x80" segmentation violation. Rather than sacrifice the elegance and portability of a "non-kernel" Linux emulation strategy, Mike Davidson has suggested that:

> This probably isn't really necessary. Assuming that we are really only interested in Linux ELF (and *not* Linux a.out) we can use the dynamic linker to "preload" the Linux runtime compatibility library in such a way that almost all of the system calls will be intercepted and handled directly by the compatibility library without ever going down to an actual "int $0x80" instruction.

3.3 Performance enhancements

Thus far, performance has not been an issue as very little negative impact has been detected. The main cost of running a Linux application under lxrun is the overhead of catching the segmentation violation (int $0x80), fixing up the structures/errors/returns, and mapping the system call. Since much of what lxrun does is done in the normal course of executing a system call natively, the SIGSEGV intercept is the main overhead.

In order to avoid catching SIGSEGV for every system call, current lxrun development plans on implementing a "pre-load" of the "Linux runtime compatibility library" (see the previous subsection). With the direct execution of Linux ELF binaries described above, it is possible to pre-load this library on startup. That is, the binary has already been identified as a Linux ELF binary since it is attempting to load /lib/ld-linux.so.1. This "fake" program interpreter knows it will have to map system calls so, rather than waiting for the SIGSEGV to trigger the mapping, the program interpreter can "pre-map". Thus, system calls made by the Linux ELF binary under the control of such a program interpreter would not cause general protection traps. In this scenario, nearly all of the performance overhead of running Linux binaries with lxrun is eliminated.

4 System Call Mapping

Lxrun emulates or maps most commonly-used system calls for which native equivalents exist. Unimplemented calls return an error indication and set ENOSYS.

The files doc/SysCallTable* list tables of supported, partially supported, and unsupported system calls. The SyscallScript utility run (against the source) will regenerate the text and HTML versions of this file. An on-line copy of the currently supported system calls is available at `<http://www.sco.com/skunkware/emulators/SyscallTable.html>`.

4.1 Directly mapped system calls

The following system calls are mapped directly:

exit() fork() creat() link() unlink() chdir() time() mknod() chmod() chown() lseek() getpid() setuid() getuid() stime() alarm() pause() utime() access() nice() sync() rename() mkdir() rmdir() dup() times() setgid() getgid() geteuid() getegid() setpgid() umask() chroot() dup2() getppid() getpgrp() setsid() setreuid() setregid() sethostname() gettimeofday() settimeofday() symlink() readlink() truncate() ftruncate() fchmod() fchown() setitimer() getitimer() fsync() setdomainname() getpgid() fchdir() sysfs() getdents() readv() writev() getsid()

4.2 System calls with a non-stub emulation function

For the following system calls, lxrun either provides some remapping of arguments, return values, and error codes, or in cases where an analogous native call does not exist, emulates the call using native library functions.

nosys() read() write() open() close() waitpid() execve() oldstat() ptrace() oldfstat() kill() pipe() brk() signal() fcntl() olduname() sigaction() sgetmask() ssetmask() sigsuspend() sigpending() setrlimit() getrlimit() getgroups() setgroups() old_select() oldlstat() uselib() readdir() mmap() munmap() getpriority() setpriority() socketcall() syslog() stat() lstat() fstat() uname() iopl() wait4() sysinfo() ipc() sigreturn() newuname() mprotect() sigprocmask() personality() _llseek() select()

4.3 Partial implementations

The following system calls are partially emulated:

ioctl() ioperm() fdatasync()

4.4 Unimplemented system calls

The list of unimplemented system calls is as follows:

mount() umount() getrusage() swapon() reboot() statfs() fstatfs() vhangup() idle() vm86() swapoff() clone() modify_ldt() adjtimex() create_module() init_module() delete_module() get_kernel_syms() quotactl() bdflush() setfsuid() setfsgid() flock() msync() sysctl() mlock() munlock() mlockall() munlockall() sched_setparam() sched_getparam() sched_setscheduler() sched_getscheduler() sched_yield() sched_get_priority_max() sched_get_priority_min() sched_rr_get_interval()

5 Getting started

5.1 How do I get lxrun ?

Lxrun is currently distributed as a component of **SCO Skunkware** `<http://www.sco.com/skunkware/>`, a free CD-ROM containing hundreds of megabytes of pre-compiled and pre-packaged software for SCO platforms. A Skunkware CD-ROM can be obtained via `<http://www.sco.com/offers/>` and, beginning in 1998, all operating systems released by SCO will contain a Skunkware CD-ROM in the shrink-wrapped product. Lxrun may also be obtained via the SCO Skunkware web site at `<http://www.sco.com/skunkware/emulators/>`. All SCO Skunkware software is freely redistributable.

5.2 Building lxrun from source

Detailed instructions on building lxrun from source, configuring the build, installing the emulation system and additional run-time components, installing a Linux binary, runtime environment variables, and error messages are contained in the file `INSTALL` in the lxrun source distribution and on-line in *the lxrun FAQ* `<http://www.sco.com/skunkware/emulators/lxrun/FAQ.html>`.

The lxrun source distribution contains a Makefile with support for compilation on SCO OpenServer 5, UnixWare 2.x, and UnixWare 7. The Makefile uses the output of "uname -r" to determine the platform. As additional platform support is added this will need to be augmented or the configuration modified to use autoconfig.

To build lxrun on one of the supported platforms, it is only necessary to issue the command "make". The command "make install" will both build lxrun and copy the resulting binaries and documentation into $(DESTDIR) which is set by default to the ./dist directory.

As a convenience this distribution includes a script called lxfront which can be used with a symbolic link (see below) to provide a wrapper around the invocation of lxrun the Linux binary name allowing them to be run directly.

Starting with lxrun 0.9.0 the build of lxrun will produce an ld-linux.so.1 as well as the lxrun binary. The ld-linux.so.1 is installed in /lib on the target system and provides support for direct execution of Linux binaries, thus deprecating the need for the lxrun binary front-end except for the execution of Linux a.out binaries.

The lxrun source has the following capability ifdefs:

- **DEF_LINUX_ROOT** - Default root directory name for lxrun to searching under for any native Linux files. Specifically when a Linux library tries to load a dynamic library, lxrun remaps the pathname to somewhere below this directory. The value specified here becomes the internal default which can be overridden by the LINUX_ROOT environment variable. Set in Makefile with make macro LXROOT. Default value is "/usr/local/linux"
- **TRACE** - Flag for making a version of lxrun that emits system call traces (to file /tmp/lxrun.nnn (where "nnn" is the process pid.)) This can be used for tracing a binaries use of an unimplemented system call or other runtime problems. Its not enabled bu default since it slows down the operation of lxrun and produces large log files. (build should be augmented to make a variantly named lxrun binary with this on regardless). Set in Makefile with make macro TRACE. Default is disabled (off)

- **ELF_DEBUG** - Flag to enable output to stderr of debug traces for the ELF loader capability of lxrun. Outputs ELF Header information, interpreter remap values and open, load, mmap, mprotect values/status (should be modified to be integrated with TRACE logging) Set in Makefile via make macro DBG. Default is disabled (off)
- **UNIDIRECTIONAL_PIPES** - Flag to enable use of unidirectional pipes (pipe()) instead of bidirectional pipes (socketpair()) for the pipe() syscall emulation on platforms where pipes are not bidirectional. Not referenced in Makefile. Default is disabled (emulation will use bidirectional pipes)
- **NO_DISPLAY_HACK** - Flag to disable remapping of DISPLAY variable from a local server specification to a full hostname specification. This remapping is done (by default) to address some problems with local connections on OSr5 (at least) to some X servers. Not referenced in Makefile. Default is disabled (remapping will be done)
- **LXRUN_AUTO_PATH_BEHAVIOR** - if enabled makes lxrun search for the Linux binary to be run in the normal PATH rather than in the (expected) absolute pathname given. Not referenced in Makefile. Default is disabled (off)
- **ELFMARK_HACK** - enables detection of binaries marked (with elfmark) as Linux binaries (mark value "LXRN" as an unsigned long). Not referenced in Makefile. Default is disabled (off) - status is experimental.

Platform defines for OpenServer 5 (OSR5), UnixWare 2.x (UNIXWARE) and UnixWare 7 (GEMINI) are automatically setup in the Makefile.

5.3 Installing the emulation system

Lxrun expects to find all its (Linux) library files in a normal root hierarchy rooted under a single place called the LINUX_ROOT. Unless respecified in the build this defaults internally in lxrun to /usr/local/linux.

You can respecify or change it at runtime with the environment variable LINUX_ROOT (wherever it ends up this must be the place the Linux libraries are placed under).

"make install" will install the built binary (lxrun) into /usr/local/bin, lxfront into $LINUX_ROOT/bin, and the lxrun program interpreter ld-linux.so.1 into /lib. The HTML documents describing lxrun are placed in /usr/local/man/html/lxrun.

5.4 Installing Linux applications

With lxrun 0.9.0 and later, Linux ELF binaries can be installed anywhere in the standard execution path (e.g. /usr/local/bin). See section 3.1 (Direct execution of Linux ELF binaries) for details on how this is done. Further, Linux applications distributed in RPM format can be installed using either a native RPM port or the Linux RPM run under the control of lxrun. Some additional arguments to RPM may be necessary. For instance, a native port of RPM for SCO OpenServer is available at `<http://www.sco.com/skunkware/osr5/sysadmin/rpm/>`. Using the SCO OpenServer RPM it is necessary to invoke RPM as follows:

```
rpm --nodeps --ignorearch --ignoreos --prefix /usr/local ...
```

The Skunkware distribution of RPM for OpenServer includes a shell script front-end `rpm4sco` which inserts these arguments for you.

Linux a.out and statically linked ELF binaries should be installed in /usr/local/linux/bin and symbolic links by the name of the binary created from /usr/local/bin to the lxfront shell script in /usr/local/linux/bin.

5.5 Error messages

`linuxemul: fatal error: program load failed: No such file or directory`

Indicates that the Linux binary couldn't run (either lxrun could't find the Linux binary or the Linux binary couldn't find the dynamic linker) It probably means your LINUX_ROOT environment variable isn't set up correctly or you don't have the required minimum Linux libraries.

`progname: can't load library 'some_library_name.so'`

Indicates you're missing a shared library that is needed to run a particular binary. You can either try to find a compiled version of the library from a Linux ftp site (such as `<ftp://sunsite.unc.edu/pub/Linux/libs/>`) or if you have access to a running Linux system, you can copy the library directly. You should put the library in $LINUX_ROOT/lib on your host system (/usr/localLinuxlib by default on a SCO system).

```
myprog: can't resolve symbol '__iob'
myprog: can't resolve symbol '__iob'
myprog: can't resolve symbol '__ctype'
```

Indicates that the Linux dynamic loader found a native SCO library and is using it instead of the corresponding Linux binary. (You can find out exactly which library is causing the problem by examining the lxrun.log file produced by a debugging version of lxrun.)

This will only occur if you have native libraries installed that have the same names as a dependant Linux binary. If you have XFree86 installed, the /usr/X11R6/lib libraries are common culprits.

The best solution is to make sure no native libraries are available anywhere under the directory pointed to by $LINUX_ROOT.

5.6 If you come across an unsupported binary

1. Go to `<http://www.sco.com/skunkware>` and make sure you have the most recent version of lxrun. If not, download the latest one and try it. We are updating lxrun with new system calls all the time.

2. Recompile lxrun with the TRACE option enabled. (This requires modifying one line in the Makefile.) This will cause lxrun to produce a history of all system calls used by the binary as it was run (similar to the "truss" and "trace" commands). The trace dump will be created in a file called "/tmp/lxrun.nnn" where "nnn" is the process id.

3. Try to narrow down exactly which system call failed. Most likely, the failure will be due to a system call that has not yet been implemented in lxrun.

4. Implement the system call mapping. This is usually pretty easy to do. The vast majority of lxrun's code does mappings of this sort, so you can pick out almost any source file to see how it is done. Chances are, the system call you need to remap is already in one of the lxrun source files, but its code looks something like this:

   int lx_flock() { errno=ENOSYS; return -1; }

   This means that you're the first person who has gotten around to mapping that particular system call.

5. After making your modification, recompile lxrun and see if it works. You may have to remap more than one system call to get your binary working!

6. E-mail your changes to `skunkware@sco.com`. This way, we can put your changes into the next release of lxrun.

7. If steps 1-5 seem beyond your programming ability, contact `skunkware@sco.com`. and maybe one of the Skunkware team will have time to give you a hand with it. Make sure to tell us exactly what program you're having trouble with, and if possible, tell us where you got it.

6 Web Presence

The lxrun web site is at <http://www.sco.com/skunkware/emulators/lxrun/>. Lxrun source is available at <ftp://ftp.sco.com/skunkware/src/emulators>.

Any source changes made (augmentation or bug fixes) doc changes feature requests, questions or problem reports should be mailed to `skunkware@sco.com`

7 Authors and Contributors

The original author of lxrun was Michael Davidson, an engineer at SCO. Major initial followup work was done by Robert Lipe and Steve Ginzburg. Andrew Gallatin ported it to Solaris/x86 and the rest of the cast includes Bela Lubkin, John W. Temples, Mike Hopkirk, Ralf Gelfand, Ronald Joe Record and Udo Monk.

Contributors to this document included Michael Davidson, Michael Hopkirk, Robert Lipe, Steve Ginzburg and the principal author - Ronald Record.

8 About This Document

This document was created using SGML-Tools 1.0.6 in conjunction with TeX, Version 3.14159 (Web2C 7.2) running on an SCO UnixWare 7 platform.

The source to this document is maintained at <http://www.sco.com/skunkware/emulators/lxrun/lxrun.sgml>. A Makefile and formatted varieties of this document are also available at <http://www.sco.com/skunkware/emulators/lxrun/>. For instance, you will find a postscript version at <http://www.sco.com/skunkware/emulators/lxrun/ps/lxrun.ps>.