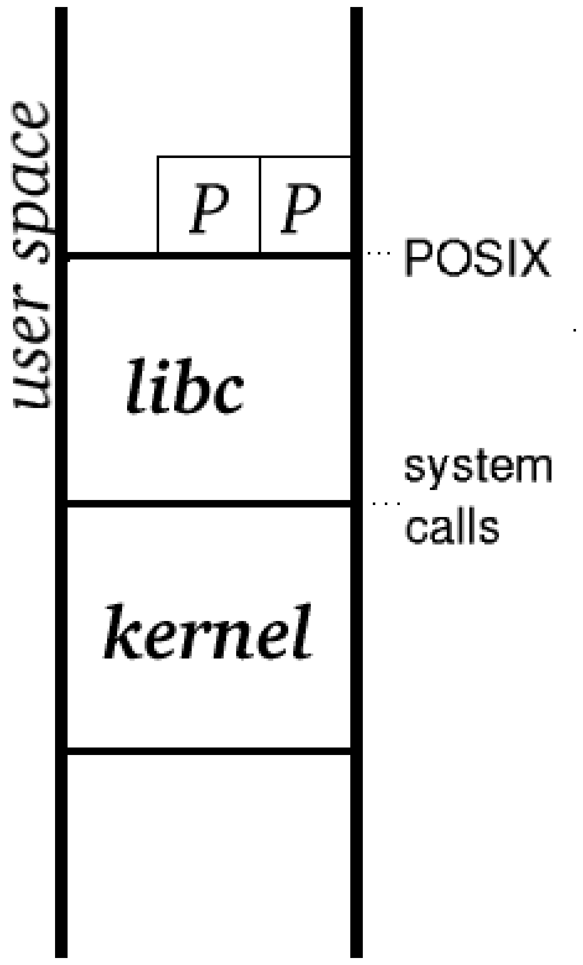
Next Previous Contents

3. User space and the libc interface



Let us look at the two interfaces: that between kernel and user space, and that between application code and system code. We have three layers, with libc between the kernel and the application code.

3.1 Application programs and C library

The programmer who writes the canonical program

```
#include <stdio.h>
int main() {
        printf("Hello world!\n");
        return 0;
}
```

programs for the *libc* interface. This C program calls the routine `printf()` that is part of the C library. A large part of the API (Application Program(ming) Interface) of the C library in a Unix environment is described by POSIX. The latest version of this standard is POSIX 1003.1-2001.

From the viewpoint of the application programmer (and from the viewpoint of the POSIX standard) there is no distinction between library routines and system calls. Kernel and C library together provide the services described.

Many things are handled by the C library itself - those are the things the user could have programmed himself, but need not since the author of the library did this job already. Maybe the presence of the library also saves some memory: many utilities can share common library code.

But for the basic things, starting programs, allocating memory, file I/O etc., the C library invokes the kernel.

3.2 Kernel and user space

The kernel provides certain services, and *user space*, that is, everything outside the kernel, both libraries and application programs, uses these. Programs in user space contain system calls that ask the kernel to do something, and the kernel does so, or returns an error code.

Application programs do not usually contain direct system calls. Instead, they use library calls and the library uses system calls. But an application program can construct a system call "by hand". For example, in order to use the system call `_llseek` (to seek in a file larger than 4 GB when `lseek` does not support that), one can write

```
#include <linux/unistd.h>

_syscall5(int, _llseek, unsigned int, fd,
          unsigned long, offset_high, unsigned long, offset_low,
          long long *, result, unsigned int, origin)

long long
my_llseek(unsigned int fd, unsigned long long offset, unsigned int origin) {
          long long result;
          int retval;

          retval = _llseek (fd, offset >> 32, offset & 0xffffffff,
                            &result, origin);
          return (retval == -1) ? -1 : result;
}
```

This `_syscall5` is a macro that expands to the definition of `_llseek` as system call, with a tiny wrapper to set `errno` if necessary and the routine `my_llseek` invokes this system call. Read the details in `/usr/include/asm/unistd.h`.

Unfortunately, these _syscall macros were removed from the kernel in 2.6.20 so that bypassing libc has become much less convenient. One needs in-line assembly or a private file with copies of the old macros.

An alternative is to use the `syscall()` call. It allows one to invoke system calls by number. See `syscall(2)`. For example, `syscall(__NR_getuid)` is equivalent to `getuid()`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
        printf("%ld\n", syscall(__NR_getuid));
        return 0;
}
```

3.3 Error return conventions

Typically, the kernel returns a negative value to indicate an error:

```
        return -EPERM;  /* Operation not permitted */
```

Typically, libc returns -1 to indicate an error, and sets the global variable `errno` to a value indicating what was wrong. Thus, one expects glueing code somewhat like

```
int chdir(char *dir) {
        int res = sys_chdir(dir);
        if (res < 0 && res > -4096) {
                errno = -res;
                return -1;
        }
        return res;
}
```

Such glueing code is automatically provided if one uses the kernel macros `_syscall`*N*, with *N* in 0..6 (for system calls with *N* parameters), defined in `/usr/include/asm/unistd.h`. These macros all end with a call of the macro `__syscall_return` defined for 2.6.14 as

```
#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-(128 + 1))) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
} while (0)
```

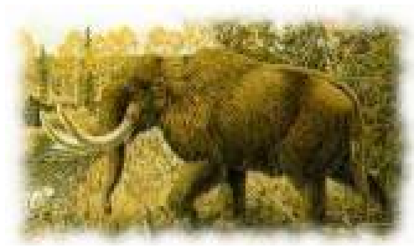
that is supposed to convert the error codes to -1. (The code is buggy: the -(128+1) and the accompanying comment suggest that this is a test for 1..128, but it is a test for 1..129, and in fact error numbers fill the range 1..131 for this kernel.)

Most system calls return positive values, but some can return arbitrary values, and it is impossible to see whether an error occurred.

3.4 Alternative C libraries

The canonical C library under Linux is *glibc*.

```
% /lib/libc.so.6
GNU C Library stable release version 2.2.5, by Roland McGrath et al.
...
```



But several other C libraries exist. In ancient times we had *libc4*, a library still used by the Mastodon distribution. It uses the a.out format for binaries instead of the newer ELF format.

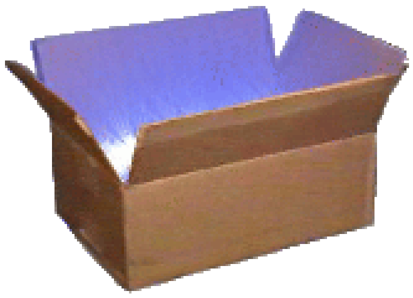
In old times we had *libc5*. It is not much used anymore, but people sometimes like it because it is much smaller than glibc.

But these days we have a handful of other small libc's: *uClibc*, *dietlibc*, *newlib*, *klibc*.

The last one is still in an early stage of development. It is intended for early user space (see also below), when the Linux kernel has been booted but no filesystem has been found on disk yet. Thus, it should be tiny, and needs only a few functions.

All of the libraries uClibc, dietlibc and newlib are meant for embedded use. Especially uClibc is fairly complete. They are much smaller than glibc.

There are also projects to recreate all standard utilities in a minimal form. See, for example busybox.



Exercise *Install and use uClibc. What is the difference in size compared to glibc of statically compiled binaries? Of dynamically compiled binaries? Is there a speed difference?*

3.5 Initial userspace

In 2.5.46 the first version of early userspace was merged into the official kernel tree. One sees the effects mainly in the dmesg output

```
-> /dev
-> /dev/console
-> /root
```

The subdirectory `/usr` of the kernel source tree is for early userspace stuff. In `init/main.c` there is the call `populate_rootfs()` that unpacks the cpio archive `initramfs_data.cpio.gz` into the kernel rootfs. There are only these three device nodes, no actual programs yet, since programs need a library, and klibc has not been merged yet.

For recent news, see initrd-dynamic (an alternative), patch1 (the first code to get merged), klibc.

If one wants to play with this, find kernel 2.5.64 and apply the above klibc patch. It contains a `usr/root/hello.c` program, that however is never invoked. Add an invocation in `init/main.c` before the `prepare_namespace()`, e.g.,

```
        unlock_kernel();
        system_running = 1;

        execve("/sbin/hello", argv_init, envp_init);

        prepare_namespace();
        if (open("/dev/console", O_RDWR, 0) < 0) ...
```

Make sure to give the program the right name, e.g,

```
        cpio_mkfile("usr/root/hello", "/sbin/hello", 0777, 0, 0);
```

in `gen_init_cpio.c`, the program used to generate `initramfs_data.cpio.gz`. (Of course one can also generate this cpio archive using cpio, and give it arbitrary contents. If the contents is large, a patch is needed that is first present in 2.5.65.)

If after booting you see `Hi Ma!` then `hello.c` executed successfully, but the boot will stop there, since this process is "init" and must never exit. (So, the "hello" program should do its work and then exec the real init.)

Note that the `/` here is in the initramfs filesystem. But after `prepare_namespace()` one has `/` in the root filesystem.

3.6 Libraries and binary formats

The binary files one meets in daily life are object files, executables and libraries.

Given the standard example `hello.c`, that contains something like

```
main() { printf("Hello!\n"); }
```

one creates the object file `hello.o` by `cc -c hello.c`, or the executable `hello` by `cc -o hello hello.c`. Now this executable does not contain code for the `printf()` function. The command

```
% ldd hello
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/libc.so.6 (0x40036000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

shows that this executable requires `ld-linux.so.2` and `libc.so.6` at run time. The former is a linker that will at startup time insert the address of the `printf()` routine (found in `libc.so.6`) into a table with function pointers. (For `linux-gate.so.1`, see the section on vsyscalls.) See also ld.so(8).

It is possible to produce complete executables, that do not require run-time linking by giving `cc` the `-static` flag: `cc -static -o hello-static hello.c`.

```
% ldd hello-static
        not a dynamic executable
% ls -l hello* | sort -n +4
-rw-r--r--  1 aeb users      31 2004-10-16 13:40 hello.c
-rw-r--r--  1 aeb users     848 2004-10-16 13:40 hello.o
-rwxr-xr-x  1 aeb users    8647 2004-10-16 13:40 hello
-rwxr-xr-x  1 aeb users 2189142 2004-10-16 13:40 hello-static
% strip hello hello-static
% ls -l hello hello-static
-rwxr-xr-x  1 aeb users   2952 2004-10-16 13:41 hello
-rwxr-xr-x  1 aeb users 388108 2004-10-16 13:41 hello-static
```

The `strip` utility removes the symbol table. Static executables are huge and usually needed only in emergency situations. For example, it is common to have a statically linked version `sln` of the `ln` utility, to set up links like `/lib/libc.so.6 -> libc-2.3.2.so` making the library name point at the actual library. (Changing such links should be done with `ln -sf ..`, so that there never is a moment that the libc link points to nowhere. If one tries to go in two steps: remove old link, create new link, then the second step will fail with an `ln` that needs libc, and suddenly no command works anymore.) It is also common to have a statically linked `/sbin/init`.

Binary formats

Various binary formats exist, like a.out, COFF, and ELF. ELF is the modern format. Support for a.out is disappearing.

The linux libc4 (like `libc.so.4.6.27`) libraries use the a.out format. In 1995 the big changeover to ELF happened. The new libraries are called libc5 (like `libc.so.5.0.9`). Around 1997/1998 libc5, maintained by HJLu, was replaced by libc6, also known as glibc2, maintained by Ulrich Drepper.

The a.out format comes in several flavours, such as OMAGIC, NMAGIC, ZMAGIC, QMAGIC. The OMAGIC format is very compact, but program in this format cannot be swapped or demand paged because it has a non-page-aligned header. The ZMAGIC format has its .text section aligned to a 1024-byte boundary, allowing `bmap()`, provided the binary lives on a filesystem with 1024-byte blocks. It was superseded by the QMAGIC format, that has its .text section starting at offset 0 (so that it contains the header) but with the first page not mapped. The result is that QMAGIC binaries are 992 bytes smaller than ZMAGIC ones, and moreover allow one to trap dereference of NULL pointers.

The binary format of an executable must be understood by the kernel when it handles an `exec()` call. There are kernel configuration options `CONFIG_BINFMT_AOUT`, `CONFIG_BINFMT_MISC`, `CONFIG_BINFMT_ELF` etc. Support for certain types of binaries can also be compiled as a module.

The kernel has (in `exec.c`) a linked list `formats` and routines `(un)register_binfmt()` called by modules who want to announce that they support some binary format. The routine `search_binary_handler()` tries all of the registered modules, calling their `load_binary()` functions one by one until one returns success. If all fail, the first few bytes of the binary are used as a magic (decimal) number to request a module that was not loaded yet. For example, a ZMAGIC binary starts with the bytes `0b 01`, giving octal 0413, decimal 267 and would cause the module `binfmt-267` to be requested. (Details depend on kernel version.)

Example of an OMAGIC file

Let us create a small OMAGIC binary by hand. We need a 32-byte header with structure as given in `<asm/a.out.h>`.

```
% cat exb.s
; aout header
.LONG 0407      ; OMAGIC
.LONG 12        ; length of text
.LONG 0         ; length of data
.LONG 0         ; length of bss
.LONG 0         ; length of symbol table
.LONG 0         ; start address
.LONG 0         ; length of relocation info for text
.LONG 0         ; length of relocation info for data

; actual program
main:
        mov eax,#1    ; exit(42)
        mov ebx,#42
        int #0x80
% as86 -3 -b exb exb.s
% od -Ad -tx1 exb
0000 07 01 00 00 0c 00 00 00 00 00 00 00 00 00 00 00
0016 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0032 b8 01 00 00 00 bb 2a 00 00 00 cd 80
0044
% objdump -d exb

exb:     file format a.out-i386-linux

Disassembly of section .text:

0000000000000000 <.text>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   bb 2a 00 00 00          mov    $0x2a,%ebx
   a:   cd 80                   int    $0x80
% chmod +x exb
% ./exb
% echo $?
42
%
```

Thus we have a 44-byte binary that consists of a 32-byte header followed by a 12-byte program. (The statically linked translation of the equivalent C program `main() { return 42; }` takes 388076 bytes.)

That worked. Now avoid creating the header by hand.

```
% cat exa.s
export _main
_main:
        mov eax,#1
        mov ebx,#42
        int #0x80
% as86 -3 -o exa.o exa.s
% ld86 -N -s -o exa exa.o
% ./exa
% echo $?
42
%
```

(Without the export line, `ld86` will complain `ld86: no start symbol`.)

That was (almost) the same 44-byte binary - one header byte differs.

Producing OMAGIC from C source seems to be done using `bcc -3 -N`, but I do not have `bcc` here.

Shared and static libraries

Linking against a static library (with a name like `foo.a`) involves copying the code for the functions needed from that library at compile time. Linking against a dynamic library (with a name like `foo.sa` for a.out, or `foo.so` for ELF) involves finding references to the functions needed at compile time, so that these can be found in the right libraries at run time. (The files `foo.sa` are not the actual libraries, but contain values of global symbols and function addresses needed for run time linking. The actual library is probably called `foo.so.1.2.3`.) The utility `ldd` tells you what libraries a program needs.

Personality

Linux has the concept of *personality* of an executable (since 1.1.20). The purpose is to make the Linux environment more similar to some other environment, like BSD or SCO or Solaris or older Linux, so that foreign or old binaries have better chances of working without modification.

For example, the Linux `select()` system call will update its timeout parameter to reflect the amount of time left. With the STICKY_TIMEOUTS flag set in the personality this is not done, following BSD behaviour.

For example, dereferencing a NULL pointer is sign of a program bug and causes a segfault because 0 is not in mapped memory. But SVr4 maps page 0 read-only (filled with zeroes) and some programs depend on this. With the MMAP_PAGE_ZERO flag set in the personality Linux will also do this.

Or, for example, the Linux `mmap()` system call will nowadays randomize the assigned addresses as a defense against hacking attempts, but with the ADDR_NO_RANDOMIZE flag set in the personality this is not done (since 2.6.12).

The personality value is composed of a 2-byte value identifying the system (Linux, SVR4, SUNOS, HPUX etc), and a number of 1-bit flags. See `<linux/personality.h>`. The personality is inherited from the parent process, and changed using the `personality()` system call. The `setarch` utility is a convenient tool for starting a process with a given personality.

LD_ASSUME_KERNEL

There is a different mechanism to tell libc and the dynamic loader what kernel ABI a program is compatible with. It was introduced at a time when different thread libraries were in use. Setting the environment variable LD_ASSUME_KERNEL to 2.4.20 (or later) would cause the dynamic loader to look in `/lib/tls`, setting it to 2.4.1 (or later) would make it look in `/lib/i686`, and setting it to 2.2.5 (or later) would make it look in `/lib`. See Ulrich Drepper's writeup.

This is an ugly hack and causes all kinds of problems. Avoid this variable if you can.

Next Previous Contents