Writing a Kernel in C

Tim Robinson · timothy.robinson@ic.ac.uk · http://www.themoebius.org.uk/

Introduction

So far, your only experience in operating system writing might have been writing a boot loader in assembly. If you wrote it from scratch it might have taken you several weeks (at least), and you might be wishing there was an easier way. Well: there is, particularly if you are already familiar with the C programming language. Even if you're not familiar with C (and you already know some other high-level language), it's well worth learning it, because it's trivial to start coding your kernel in C. It's a matter of getting a few details correct and having the right tools.

The C Programming Language

C was originally designed as a low-level HLL (Unix kernels have traditionally been written in C) and its main advantages are its portability and its closeness to the machine. It's easy to write non-portable programs in C, but given some knowledge of the language, it's possible to code with portability in mind. There's nothing special about compiled C code that makes it different to assembly – in fact, your assembler is nothing more than a low-level compiler, one in which statements translate directly into machine opcodes. Like any other high-level language, C makes the code more abstract and separates you from the machine more; however, C doesn't place too many restrictions on the code you write.

Tools

There are two main software components involved in generating machine-executable code from C source code: a compiler and a linker. The compiler does most of the work, and is responsible for turning C source code into object files; that is, files which contain machine code and data, but don't constitute a full program on their own. Many C compilers have an option where they generate an executable file directly from the source code, but in this case the compiler probably just invokes the linker internally.

The linker bundles all the object files together, patches the references they make to each other and moves them about (relocation), and generates an executable file. Most linkers will work with source files from any compiler, as long as the compiler produces compatible object files. This allows you to, for example, code most of your kernel in C but write some of the machine-specific parts in assembly.

I'd recommend any package that contains the GNU `gcc` compiler and `ld` linker, because:

- They're free and open-source
- `ld` supports virtually any executable format known to man
- Versions of `gcc` are available for virtually any processor known to man

GNU packages are available for various operating systems; the MS-DOS port is called DJGPP, and one good Windows port is Cygwin. All Linux distributions should include the GNU tools. Note that each port generally only supports the generation of code for the platform on which it runs (unless you recompile the compiler and linker), and the bundled run-time libraries are generally of little use to the OS writer: stock DJGPP programs require DOS to be present, and stock Cygwin programs rely on `cygwin1.dll`, which in turn uses the Win32

API. However it is possible to ignore the default RTL and write your own, which is what we'll be doing. If you're writing code on Windows, I'd recommend that you use Cygwin (patch/rebuild it to support ELF if desired) because it's a lot faster than DJGPP and allows use of long command lines and file names natively. Note that, at the time of writing, Cygwin's flat-binary output is broken.

Various other tools come in useful in OS development. The GNU `binutils` package (bundled with `gcc` and `ld`) includes the handy `objdump` program, which allows you to inspect the internal structure of your executables and to disassemble them – vital when you're writing loaders and trying a tricky bit of assembly code.

One possible disadvantage of using a generic linker is that it won't support some custom executable format that you invent yourself. However, there's seldom any need to invent a new executable format: there are already a lot of them out there. Unix ELF and Windows PE tend to be most popular among OS writers, mainly because they're both well supported among existing linkers. ELF seems to be used more in amateur kernels, mainly because of its simplicity, although PE is more capable (and hence more complex). Both are well documented. COFF is also in use in some projects, although it lacks native support for features such as dynamic linking. An alternative for simple kernels is the flat binary format (i.e. no format at all). Here, the linker writes raw code and data to the output file, and the result is similar to MS-DOS's .COM format, although the resulting file can be larger than 64KB in length and can use 32-bit opcodes (assuming the loader enables protected mode first).

A disadvantage of all the mainstream IA-32 compilers is that they assume a flat 32-bit address space, with CS, DS, ES and SS each having the same base address. Because they don't use far (48-bit seg16:ofs32) pointers they make it a lot harder to write code for a segmented OS. Programs on a segmented OS written in C would have to rely on a lot of assembly code and would be a lot less efficient than programs written for a flat OS. However, at the time of writing, Watcom are still promising to release their OpenWatcom compiler, which will supposedly support far pointers in 32-bit protected mode.

Here's another couple of warnings about using `gcc` and `ld` (although these could potentially apply to any compiler linker). Firstly, `gcc` likes to put the literal strings used by functions just before the function's code. Normally this isn't a problem, but it's caught out a few people who try to get their kernels to write "Hello, world" straight off. Consider this example:

```
int main(void)
{
        char *str = "Hello, world", *ch;
        unsigned short *vidmem = (unsigned short*) 0xb8000;
        unsigned i;

        for (ch = str, i = 0; *ch; ch++, i++)
                vidmem[i] = (unsigned char) *ch | 0x0700;

        for (;;)
                ;
}
```

This code is intended to write the string "Hello, world" into video memory, in white-on-black text, at the top-left corner of the screen. However, when it is compiled, `gcc` will put the literal `"Hello, world"` string just before the code for `main`. If this is linked to the flat binary format and run, execution will start where the string is, and the machine is likely to crash. There are a couple of alternative ways around this:

- Write a short function which just calls `main()` and halts. This way, the first function in the program doesn't contain any literal strings.
- Use the `gcc` option `-fwritable-strings`. This will cause `gcc` to put literal strings in the data section

of the executable, away from any code.

Of these, the first option is probably preferable. I like to write my entry point function in assembly, where it can set up the stack and zero the bss before calling `main`. You'll find that normal user-mode programs do this, too: the real entry point is a small routine in the C library which sets up the environment before calling `main()`. This is commonly known as the `crt0` function.

The other main snag concerns object-file formats. There are two variants of the 32-bit COFF format: one used by Microsoft Win32 tools, and one by the rest of the world. They are only subtly different, and linkers which expect one format will happily link the other. The difference comes in the relocations: if you write code in, say, NASM, and link it using the Microsoft linker along with some modules compiled using Visual C++, the addresses in the final output will come out wrongly. There's no real workaround for this, but luckily most tools that work with the PE format will allow you to emit files in the Win32 format: NASM has its `-f win32` option, and Cygwin has the `pei-i386` output format.

The Run-Time Library

A major part of writing code for your OS is rewriting the run-time library, also known as `libc`. This is because the RTL is the most OS-dependent part of the compiler package: the C RTL provides enough functionality to allow you to write portable programs, but the inner workings of the RTL are dependent on the OS in use. In fact, compiler vendors often use different RTLs for the same OS: Microsoft Visual C++ provides different libraries for the various combinations of debug/multi-threaded/DLL, and the older MS-DOS compilers offered run-time libraries for up to 6 different memory models.

For the time being, though, it should be sufficient to write only one run-time library (although writing a makefile which offers a choice of static or dynamic linking is often useful). You should aim to replicate the library defined by the ISO C standard because this will make porting programs to your OS easier. If you write a non-standard library you'll have to re-write any applications you want to port; the more standard library functions you define, the easier it will be to port open-source applications straight across.

It makes it a lot easier to write library code if you can get hold of the source code for an existing implementation, particularly if that library's environment is similar to the one you're developing in. There are a lot of C functions which are OS- and platform-independent: for example, most of `<string.h>` and `<wchar.h>` can be copied straight across. Conversely, there are a lot of functions that your compiler might offer which won't make sense for your OS: a lot of DOS compilers offer `<bios.h>` header file which allows access to the PC BIOS. Unless you write a VM86 monitor in your kernel, you won't be able to call BIOS functions directly. However, such functions will be extensions to the C standard library, and, as such, will have names beginning with an underscore (e.g. `_biosdisk()`). You are under no obligation to implement these extensions, and you're free to define your own extensions if you wish, as long as you give them a name starting with an underscore.

Other C library functions are dependent on kernel support: most of `<stdio.h>` relies upon there being some sort of file system present, and even `printf()` needs some destination for its output. On the subject of `printf()`: most open-source C library implementations will define a generic `printf()` engine because the same functionality is needed in so many different places (`printf()`, `fprintf()`, `sprintf()`, and the `v`, `w` and `vw` versions of these functions). You should be able to extract this common engine and use it for your own, or at least try to emulate it: write a function which accepts a format string and a list of arguments and which sends its output to some abstract interface (either a function or a buffered stream).

Although a full run-time library is most useful when it comes to writing and porting user applications, it is also convenient to have good RTL support in the kernel. Including commonly-used routines (such as strcpy() and malloc()) will make it quicker and easier to write kernel code, and it will make the resulting kernel smaller, since common routines are only included once. It is also good practice to make such routines available to driver writers, to save them from having to include common routines in their driver binaries.

OS-Specific Support

As a kernel language, C isn't perfect: obviously, there's no standard way of allowing access to a particular machine's features. This means that it's often necessary to either drop into inline assembly when it is needed, or to code portions in assembly and link them with the compiler's output at link time. Inline assembly allows you to write parts of a C function in assembly, and to access the C function's variables as normal: the code you write is inserted among the code the compiler generates. Support for inline assembly varies among compilers; if you can stand the AT&T syntax, gcc's is best among PC compilers. Although Visual C++ and Borland C++ both use the more familiar Intel syntax, their inline assemblers aren't as fully-integrated with the rest of the compiler. gcc forces you to use more esoteric syntax but it allows you to use any C expression as an input to an assembler block and it allows you to place the outputs anywhere. The resulting assembly is also better integrated with the optimizer than in Borland's and Microsoft's compilers: for example, gcc allows you to specify exactly which registers are modified as the result of an assembly block.

Some assembly code will also be required in user mode if you use software interrupts to invoke kernel syscalls. You might place the interrupt calls directly into the C library code (as MS-DOS libraries do), or generate a separate library with a function for each call and link normally to that (as Windows NT does with ntdll.dll). It may also make sense to put the OS-dependent interface in a language-independent library, so that programs written in different languages can use the same OS library. This will allow people to write programs for your OS in languages other than C, or in languages which don't allow C bindings.

C++ in the Kernel

Opinions vary widely over the use of C++ in a kernel: most Linux kernel coders stay well away from it, whereas some people have entire operating systems in C++. I'd personally stick with C, although I see no real reason why you shouldn't use C++, as long as you know what you're doing. One thing you should realize is that to write a C++ kernel you'll need to write a bit more code for the framework, and that some C++ features are off-limits to you (unless you can write the necessary support code).

First off, you'll need to code the new and delete operators. This is easy if you've already coded malloc() and free(): new and delete can be single-line functions which call each of these. If you want to have global instances of classes you'll need to put some code in your startup routine to call each of their constructors. The way this is done will differ between compilers, so the best thing to do is to browse through your compiler's run-time library to see how the vendor did it; try looking in files with names like crt0.c. You'll probably also need to implement atexit(), because the compiler is likely to emit code which uses atexit() to call global object's destructors when the program ends. If you want to use try/catch in your kernel you'll have to implement whatever mechanism your compile uses to implement them. Again, this will depend on the compiler you use, and whatever operating system that compiler targets.

In general, I'd steer clear of both exception handling and huge virtual classes in your kernel. Exception handling usually adds unneeded bloat, and calling masses of virtual functions add unnecessary indirection and defeats the optimizer somewhat. Remember, your kernel should be as efficient as possible, even to the detriment of a beautiful design where necessary.

Overview

Writing an OS in a high-level language such as C can be a lot more productive than coding one entirely in assembly, particularly if you're willing to forego the slight speed increase that assembly offers. There are compilers freely available, such as `gcc`, which make writing kernel code relatively easy, and using C will probably prove beneficial in the long run.

Created 29/12/01; last updated 01/01/02.
Updated by K.J. 02/14/02 & 02/20/02

This tutorial is mirrored on this website with premission from Tim Robinson.