Creating a C Library

Difficulty level
Advanced

This tutorial will discuss implementing your own Standard C Library (libc). While implementing a minimal subset for kernel use is easy, it is considerably more work to implement sufficient functionality to port third party programs. You may wish to save yourself the effort and port an existing C library. Before going ahead, you may wish to create a OS Specific Toolchain, which will allow us more control of the compiler. Note that this tutorial is somewhat ELF-specific, although the relevant details can be adapted to other platforms. This tutorial also contains information on how to adapt your GCC cross-toolchain to your operating system.

Contents

Building

Now that you have an OS-specific freestanding toolchain that can build your kernel, there is a problem of bootstrapping. To build an OS-specific toolchain that supports a hosted environment, you need the headers of your standard C library installed. To build your libc you need a compiler that supports a hosted environment (building a user-space libc as in a freestanding environment is a logical mistake). The solution is to create a make target that installs your C library and kernel headers into your sysroot's include directory without needing a compiler. Then you can simply build your cross-compiler with --with-sysroot="$SYSROOT" (and not giving the --without-headers option), and you should get a cross-compiler that offers a freestanding environmment for your kernel, and a hosted environment for your user-space. Note that libgcc may depend on a few things from your C library: if you get errors during building libgcc, then simply add the declarations to your header files. You will get undeclared symbols if GCC deems it appropriate to call a libgcc function that needs a libc symbol, so it is a good idea to implement what libgcc needs early on. Note that libgcc is *not* optional, and gcc *will* emit calls to it if it thinks it is a good idea.

Then you can simply build your C library source files using:

```
x86_64-myos-gcc -c strfoo.c -o strfoo.o
x86_64-myos-as x86_64/crt0.s -o x86_64/crt0.o
x86_64-myos-ar rcs libc.a strfoo.o x86_64/crt0.o
```

Note that unlike the kernel, you don't need to add extra special options that disable standard include directories and libraries. After all, this is the C library meant for user-space. You don't need to add -I options to the compiler if you already have installed your libc and kernel headers into your sysroot. Otherwise, simply add the appropriate -I options to the above commands. It may be useful to use the same libc for both user-space and the kernel, in that case don't make the mistake of thinking that user-space-libc and kernel-libcs are the same thing. First of all many user-space things don't make sense (or even work) in the kernel. Secondly, a kernel-usable libc must be built differently than the user-space libc because it is *freestanding*. A kernel libc must have all the special options that a kernel binary is passed. For instance, don't forget to add -mno-red-zone on x86_64 to both your kernel and libc, or interrupts may corrupt the stack.

Program Initialization

See also: Calling Global Constructors

The first and most important thing to implement in a C library is the _start function, to which control is passed from your program loader. Its task is to initialize and run the process. Normally this is done by initializing the C library (if needed), then calling the global constructors, and finally calling exit(main(argc, argv)). You can change the name of the default program entry point by adding ENTRY=_my_start_name in your OS-specific binutils emulparams script (binutils/ld/emulparams). You can change which start files are used by modifying gcc/gcc/config/myos.h in your OS-specific GCC. The macros STARTFILE_SPEC and ENDFILE_SPEC define the object files to use in the GCC spec language. See gcc/gcc/config/gnu-user.h for examples on how to use this. If you decide to use the conventional (GNU-like) names and semantics for these initialization files, the following information applies:

crt0.o

The crt0.o object file will contain the _start function that initializes the process and calls exit(main(argc, argv)). This file is normally written in assembly, though depending on how your program loader invokes _start, you may be able to write it in C. Note that the SysV ABIs have some opinions on how _start is invoked and the stack contents. Adhering to this standard may help porting third party software to your OS.

Below is a simple implementation of crt0.s for x86_64. It assumes that the program loader has put *argv and *envp on the stack, and that %rdi contains argc, %rsi contains argv, %rdx contains envc, and %rcx contains envp

```
.section .text

.global _start
_start:
        # Set up end of the stack frame linked list.
        movq $0, %rbp
        pushq %rbp # rip=0
        pushq %rbp # rbp=0
        movq %rsp, %rbp

        # We need those in a moment when we call main.
        pushq %rsi
```

```
        pushq %rdi

        # Prepare signals, memory allocation, stdio and such.
        call initialize_standard_library

        # Run the global constructors.
        call _init

        # Restore argc and argv.
        popq %rdi
        popq %rsi

        # Run main
        call main

        # Terminate the process with the exit code.
        movl %eax, %edi
        call exit
.size _start, . - _start
```

This implementation is careful to set up the end of the stack frame linked list. If you compile your files without optimization or you use -fno-omit-frame-pointer, then each function adds itself to this linked list. This is very useful if you wish to add calltracing support. In that case, you'll need to know when you have reached the end, which is why we add an explicit zero in the above code..

Next the _start function calls initialize_standard_library. Note how the program loader register usage nicely fits the x86_64 SysV calling convention, and how the initialize_standard_library(int argc, char* argv[], int envc, char* envp[]) function accepts the same arguments as _start. We save the argc and argv variables, because we'll need them in a moment when we call main. How initialize_standard_library is implemented is up to you, but normally it sets environ to envp, sets up stdin, stdout, and stderr, the heap, and whatever needs to be done.

Programming languages such as C++ offer a feature called global constructors. This allows code to be executed before the main function is called, but after the core standard library has been initialized. Note how GCC allows C programs to use the same feature using the function attribute ((constructor)). Some third party software, such as binutils, silently relies on global constructor functions to be called and otherwise malfunction, even if the program compiled seemingly correctly. To call these global constructors, we call the traditionally-named _init function, which is a piece of magic we'll set up in a moment.

Finally we restore the %rdi and %rsi registers and call main, and use its return value as argument to exit. The _exit function must then call the _fini function, which calls the global destructor functions (as opposed to _init that calls the constructors).

crtbegin.o, crtend.o, crti.o, and crtn.o

These special object files are provided by your OS-specific compiler. The compiler is told to make these files in gcc/gcc/config.gcc by extra_parts="crtbegin.o crtend.o". GCC internally maintains tables of global constructor/destructor functions. However, these tables are set up in a manner such that you cannot directly link to them yourself. Rather, the files crtbegin.o and crtend.o contains the instructions that uses these tables and calls them as requested. The bad news, however, is that crtbegin.o and crtend.o offer no symbols we can call. Instead, GCC expects us provide these symbols ourselves using some linker tricks. The STARTFILE_SPEC and ENDFILE_SPEC macros tell the compiler which files must be linked in before anything else, and which files must be linked in after anything else. The idea is that files are linked in this order: crt0.o, crti.o, crtbegin.o, your-program-foo.o, your-program-bar.o, crtend.o, crtn.o.

The crtbegin.o and crtend.o files contain the necessary instructions that call global constructors in the .init section, and the instructions that call the global destructors in the .fini section. GCC expects us to put the header of the _init function in crti.o's .init section and the footer of the _init function in crtn.o.'s .init section. The same applies to the _fini function and the .fini section. Using that, we can now implement the _init and _fini functions in crti.o and crtn.o.

Hence an crti.s implementation will simply be (x86_64):

```
.section .init
.global _init
_init:
   push %rbp
   movq %rsp, %rbp
   /* gcc will nicely put the contents of crtbegin.o's .init section here. */

.section .fini
.global _fini
_fini:
   push %rbp
   movq %rsp, %rbp
   /* gcc will nicely put the contents of crtbegin.o's .fini section here. */
```

and a simple implementation of crtn.s will be (x86_64):

```
.section .init
   /* gcc will nicely put the contents of crtend.o's .init section here. */
   popq %rbp
   ret

.section .fini
   /* gcc will nicely put the contents of crtend.o's .fini section here. */
   popq %rbp
   ret
```

Finally, you simply need to assemble your crt0.o, crti.o, and crtn.o files and install them in your system library directory. Your _start function is now able to set up the standard library, call the global constructors, and call exit(main(argc, argv)). Don't forget to call your _fini function in your exit function, or the global destructors won't be run, leading to subtle bugs.

Implementing

Most of this will be up to you. A good place to start are the mem* and str*-functions, as they are mostly independent of syscalls and other library routines and make unit testing a breeze. clang and gcc, being nice compilers, both provide a set of builtin (https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html) functions that can be used to skip implementing some parts. This approach gives you working versions of these functions, with the tradeoff being that this method is compiler-specific to clang and gcc only. While this is probably fine for a kernel, your mileage may vary in userland.

Standards

Creating your own C library has a number of advantages. For instance, you can finally add that useful fopen_http that the GNU libc maintainers rejected. You can also add a bunch of new, useful conversion specifiers to your printf. But before you get too carried away, you should spend a few minutes thinking about compatibility. For instance, if you do add your fopen_http and your user-space programs use it, then you just lost portability back to Linux. While you are still bootstrapping your operating system, it is very valuable to be able to run and test your user-space

programs from your host operating system. It allows more rapid development and makes sure that your libc functions do the same as the ones on your host system. Rather than adding these new utility functions to libc, it may well be worth adding them to a new separate library, which is portable and works under your operating system and your host operating system.

Additionally, you should be very careful about changing semantics of existing libc functions or adding new functions to existing headers (namespace pollution). This may very well confuse third-party programs and cause massive porting problems. Whether you do choose to diverge from the conventional path, you should think twice: Is it really worth it? Can you make a new, nicer function and have the old one call the new one? Could your extension conflict with stuff from BSD, Plan 9, or another libc? Has anyone already decided that a given C interface is bad and made a sane replacement? Choosing to copy the interfaces of implementations can certainly help you port software.

Fortunately, you can find plenty of the relevant standards online. For instance, you can look at:

- The C Standard (2011), latest draft is available here (PDF) (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf).
- POSIX (2008) (http://pubs.opengroup.org/onlinepubs/9699919799/).
- GNU libc documentation.

Include Guards

C library headers conventionally use include guards to prevent multiple declarations. It may be wise to use the same format as GNU's libc as some programs (and GCC fixincludes) may erroneously rely on these macros to detect whether a given header has been included. GNU libc's headers usually follow this simple scheme:

```
#ifndef _STDIO_H
#define _STDIO_H 1

/* ... */

#endif
```

Repeated Declarations

The occasional poor design of the C standard library has lead to the requirement that some declarations must be repeated in multiple header files. Worse, if a header needs the FILE declaration from stdio.h, it is not allowed to include stdio.h because of namespace polution. This leads to a multiple maintaince problem and can be dangerous if you update the declaration in one place and forget to do it in another. If you have a look at GNU libc's stdio.h header, you will discover a maze of #ifdefs and #define __need_FILE preprocessor magic that allows you to #define __need_FILE #include <stdio.h> and then get *only* the declaration of FILE. While this works, it is very ugly and grows horribly in complexity.

Another solution is to put the FILE in its own header <decl_FILE.H> with its own include guard. This is much cleaner and simpler to understand, but causes extra work for the preprocessor which can get comparably expensive if this is done for a lot of declarations. An alternative solution is to

pre-preprocess the header files by writing a simple compiler that automatically finds these <decl_FILE.h> inclusions and inserts the contents into the header file. This trades some build-time-complexity for maintainability and runtime performance.

C++ Compatibility

Traditionally C++ programs can use the C headers, even though such headers are written in C and have C linkage. Unless you specify otherwise, GCC will assume that header files found in your system include directory are written in C and have C linkage. This is done by GCC automatically inserting extern "C" { ... } around all included system headers. However, this may lead to strange linking-failures if you try to use C++ headers from the system include directory. The key solution is to make your C headers explicitly compatible with C++ and tell GCC that you understand C++.

For GCC versions < 9, telling GCC that your headers understand C++ is done by having the following in your OS-specific gcc/gcc/config/myos.h:

```
/* Don't assume anything about the header files.  */
#undef  NO_IMPLICIT_EXTERN_C
#define NO_IMPLICIT_EXTERN_C    1
```

For GCC versions >= 9, this patch (https://patchwork.ozlabs.org/patch/934478/) changed the behaviour; they have made `NO_IMPLICIT_EXTERN_C` a so-called "poisoned identifier", so compiling your OS-specific toolchain will fail. If you read the linked patch above, you will realise that they decided to invert the behaviour, such that system headers are assumed to understand C++ by default. So, in theory, you do not need those two lines above at all, and it should Just Work (tm).

If, instead, your system headers indeed *do not* support C++ (why?), you can `#define SYSTEM_IMPLICIT_EXTERN_C 1`, as they have done for AIX.

Adding support for C++ in your C header files (such as stdio.h) is very simple. Previously GCC automatically added extern "C" around all headers if compiling C++, but now we'll need to do it ourselves. The key feature is that we don't need to do this on C++-only headers, meaning that C++ headers will now actually work, instead of GCC assuming they have C-linkage. For instance, you can change your stdio.h to be of this form:

```
#ifndef _STDIO_H
#define _STDIO_H 1

#if defined(__cplusplus)
extern "C" {
#endif

int printf(const char* format, ...);

#if defined(__cplusplus)
} /* extern "C" */
#endif

#endif
```