Converting a regular kernel into MOOL kernel

Pradeepkumar Gayam

July 24, 2018

Contents

Preface

This document is prepared with an objective to help future researchers in converting a regular kernel into a MOOL kernel. It introduces relevant concepts, libraries and their purpose. It also tries elaborate on functionality of the libraries that are included in the kernel.

This document is not aimed at providing a step by step guide to convert a regular kernel into MOOL kernel. Such guides are not feasible for projects of this scale. Codebases that are as big as Linux kernel change fast. So, any "How To" guides become obsolete very quickly. The best approach one need to follow is to understand the required concepts and tackle the problem logically.

This document has four chapters.

1. Introduction
2. Concepts
3. Libraries
4. Conversion process

First chapter describes the problem and mentions possible solutions to the problem. Second chapter introduces the concepts one has to understand to make sense of the functionality that is added to the kernel. Any modifications done without understanding these concepts will be a guess work and very likely cause unfix-able errors.

The chapter on Libraries explains the purpose of the libraries that need to be included in the kernel. The aspiration is to provide explanations as deep as possible, but that's a never ending task.

The final chapter describes the steps one has to follow to convert a regular kernel into MOOL kernel. No commands are mentioned in this chapter. It merely provides order of steps one has to follow to complete the task.

Chapter 1

Introduction

The objective is to redesign the Linux kernel to reduce coupling and increase maintainability by means of OO (Object Oriented) abstractions. First step in achieving this goal is to add the capability of running C++ code in the kernel.

Different type of solutions are available to use both C & C++ together in a single project. These solutions are relatively easy because both C and C++ follow (almost) same ABI. But, all these solutions are aimed at userspace projects and not aimed at kernel. To solve this problem for Linux kernel, much more nuanced approach is required.

Among all the available solutions, we take the approach of compiling C and C++ code with `gcc` and `g++` respectively and combine them at the link time. For this approach to work correctly, g++ should not emit erroneous object code, i.e., not insert calls to that functions that are not available in the kernel.

We use Freestanding compiler to compile the kernel. The freestanding version of the compilers doesn't provide C/C++ standard libraries and other useful runtime features. If we look at top level `Makefile` of Linux kernel, linker is run with `'ld -nostdlib -nodefaultlibs -nostartfiles'`. These options prevent the linker from using `libgcc, libc` and `libstdc++`. So, we need to port the relevant features from these libraries into the kernel.

To be precise, we need to add runtime support to make the following features of C++ available in kernel.

- Memory allocation operators
- Global constructors/destructors
- Virtual functions
- Dynamic type checking
- Exceptions

Once the C++ runtime support is added, programming in C++ at kernel level become similar to programming in user space.The compiler compiles files ending with .cc as C++ file. However, the Linux kernel is written in vanilla C, so C++ source files do need to include C files. This introduces a problem not commonly encountered in user space, as some of the C++ keywords have been used as identifiers in some of the Linux header files.

After these two problems are taken care, we need to solve the problem of kernel not paying attention to .init of loadable kernel module and incompatibilities between `C` and `C++`.

Chapter 2

Concepts

2.1 Runtime libraries

A runtime library is a set of low-level routines used by a compiler to invoke some of the behaviors of a runtime environment, by inserting calls to the runtime library into compiled executable binary. Some functions that can be performed only during runtime are implemented in the runtime library. This may vary from implementation to implementation.

A runtime library may contain following items.

- Memory management
- Array bound checking
- Dynamic type checking
- Exception handling
- Debugging functionality

Some libraries implement runtime functionality in standard library. The Border between runtime library and standard library is very thin. The concept of a runtime library should not be confused with an ordinary program library like that created by an application programmer or delivered by a third party, nor with a dynamic library, meaning a program library linked at run time. For example, the C programming language requires only a minimal runtime library (commonly called crt0), but defines a large standard library (called C standard library) that has to be provided by each implementation. The runtime library is is always compile and platform specific.

2.1.1 GCC runtime libraries

When we run a program, the entry point appears to be `main()` function we write. But under the hood, main() is not the entry point to the program. `_start` function defined in `crt0.o` is the entry point to a program.

When OS loads an executable, it sets up the process and then it calls the function `_start`. Before `_start()` is called there are list of things that needed to be taken care.

They're :

1. Allocate space for a software stack and initialize the stack pointer
2. Allocate space for a heap
3. Parsing the command line arguments
4. Running global constructors and destructors(ctors, dtors)
5. etc..

These tasks are shared between `crt0.o`, `crti.o`, `crtbegin.o`, `crtend.o` `crtn.o` in the same order. `crtbegin.o` and `crtend.o` are provided by the compiler. For gcc/g++ theyre reside in `libgcc/crtstuff.c`

- http://www.embecosm.com/appnotes/ean9/html/ch05s02.html
- https://wiki.osdev.org/Calling_Global_Constructors

2.2 System V ABI

An ABI is defined as set of specifications based on which operating system and executables are created. The ABIs are standards to be followed by the code-generator phase of the compiler. ABIs specify the details of calling conventions, object file formats such as ELF, linking details and much more. For example ELF is included in ABI because the ELF format defines the interface between operating system and application. When OS runs a program, it expects the program to be formatted in a certain way(for example expects the first section of the binary to be an ELF header) and contain certain information at specific memory offsets. This is how the application communicates important information about itself to the operating system.

A non-ELF binary format (such as a.out or PE), then an OS that expects ELF-formatted applications will not be able to interpret the binary file or run the application. This is the reason why Windows apps cannot be run directly on a Linux machine (or vice versa) without being either re-compiled or run inside some type of emulation layer that can translate from one binary format to another.

An example ABI for a x86 machine is here

Most of the ABIs tend to not include architecture specific details. These details are supplied by processor manufacturer. These details combined with above details forms a complete ABI for an operating system and processor pair.

- http://www.sco.com/developers/gabi/
- https://gcc.gnu.org/onlinedocs/gccint/Initialization.html

2.3 `C++` ABI

Writing C++ code in kernel is very similar to writing C code, except that there are a few pitfalls that need to taken care, such as runtime support, constructors, destructors etc. A lot of features C++ offers can be used on-the-fly. They require no additional support or code to use them properly (e.g. templates, classes, inheritance, virtual functions).

The support for additional functionality of `C++` is standardized in Itanium `C++` ABI. As an ABI, it gives precise rules for implementing the language, ensuring that separately-compiled parts of a program can successfully inter-operate. Though it was initially defined for Itanium processors, later the standard grew to be platform neutral. Along with C runtime support, to run `C++` programs, certain functionality need to be added into the program for the binary to run without problems. This ABI tries to standardize the following.

1. Name mangling
2. Exception handling
3. Run Time Type Information(RTTI)
4. and more..

Complete ABI specification is available here

Most of this functionality is provided in libsup++. But in the context of kernel, additional libraries are needed for seamless running of `C++` programs.

- https://wiki.osdev.org/Calling_Global_Constructors
- http://wiki.osdev.org/C_PlusPlus
- https://wiki.osdev.org/Libsupcxx

2.4 GCC internals

GCC comprises a number of different compilers for different programming languages. The main GCC executable gcc processes source files written in C, `C++`, Objective-C, Objective-`C++`, Java etc and produces an assembly file for each source file. gcc/g++ is a driver program that invokes the appropriate compilation programs depending on the language of the source file.

For a C source file they are the preprocessor and compiler `cc1`, the assembler `as`, and the linker `collect2`. `cc1` and `collect2` come with a GCC distribution, the assembler is a part of the GNU binutils package.

GCCs design can be broadly divided into three sections.

1. Front end
2. Middle end
3. Back end

2.4.1 Front end

The purpose of the front end is to read the source file, parse it, and convert it into the standard abstract syntax tree (AST) representation. Each front end uses a parser to produce the abstract syntax tree of a given source file. The out of this step is a form called GENERIC. This output is used in the middle end.

2.4.2 Middle end

Most of the optimizations that happen before binary creation happens in this step. The GENERIC tree is converted into another representation known as GIMPLE. In this from each expression contains three or less operands. This GIMPLE output is converted to static single assignment(SSA) tree. This form goes multiple passes of optimizers. After the optimizations, the tree is converted into GIMPLE again, which is converted into RTL form.

2.4.3 Back end

GCC uses this RTL form output to generate the assembly code the for target architecture

- https://gcc.gnu.org/onlinedocs/gccint/

2.5 Freestanding vs Hosted compilers

Whenever we do C or `C++` programming in user-space, we use a so-called Hosted Environment. Hosted means that there is a C/C++ standard library and other useful runtime features. Alternatively, there is the Freestanding version. Freestanding means that there is no C/C++ standard library, only what we provide ourselves. However, some header files are actually not part of the C standard library, but rather the compiler. These remain available even in freestanding C source code. In this case we use `<stdbool.h>` to get the bool datatype, `<stddef.h>` to get size_t and NULL etc. `<float.h>`, `<iso646.h>`, `<limits.h>`, and `<stdarg.h>` are also part of freestanding. GCC ships few more headers along with these.

The cross compilers we use to compile the kernel are freestanding compilers.

2.6 C standard library(libc)

The C standard library provides string.h, stdio.h, stdlib.h etc, that are used for string manipulation, input/output, memory management and other purposes. The interface is described in the C standard(ISO/IEC 9899), with further additions described in POSIX.

There are two types of C compilation environments. 1. Hosted, which requires a kernel to work 2. Freestanding, which contains small number of headers which contain defines and types. Hosted environment is used for userspace programming while free freestanding environment is for kernel programming. There is actually no libc in kernel space. Libc is user-space library, and you can't use it from kernel-space. To switch to freestanding environment we need to pass '-ffreestanding' flag to the compiler.

There are several implementations of libc by different vendors. The libc that is used in android is Googles implementation named Bionic

- https://wiki.osdev.org/Creating_a_C_Library

2.7 Purpose of `libgcc`

GCC uses a special library called libgcc during code generation, which contains helper routines and runtime support. Its contents depends on the particular target. All code compiled by GCC must be linked with libgcc. The library is automatically included by default when we link with GCC. However, kernels usually don't link with the standard user-space libc for the reasons mentioned in the above section.

The helper routines that are present in `libgcc` are arithmetic operations that target process cannot perform directly, routines for exception handling and other miscellaneous operations.

2.8 Purpose of `libsupc++`

`libsupc++` is part of `C++` standard library(libstdc++), which contains functionality that provides support for memory management, run-time type information(RTTI) and exception handling. To use exceptions and RTTI in kernel, libsupc++ need to be supplied and complied with the kernel. `libsupc++` requires some parts of `libgcc` such as exception support, to be included in the kernel.

If we want exception, RTTI, new delete together, we need to use `libgcc` and `libsupc++` together. `libgcc` contains unwinder which is used for exceptions. `libsupc++` has a number of dependencies which the kernel need to provide has an alternative to malloc, free and abort etc. These alternative implementations reside in `lib/gccfixdefines.h`

If we dont want to use exception and RTTI we can use `-fno-exception` and `-fno-rtti` with `g++`.

2.9 Incompatibilities between `C` and `C++`

2.9.1 Designated initializers

TODO

Chapter 3

Libraries

3.1 `fixdefines.h`

This file defines the following routines, which provide kernel level implementation of `malloc, free, abort`.

- `void* cxx_malloc(size_t sz)`
- `void cxx_free(void* p)`
- `void cxx_abort(void)`

3.2 `cxa_atexit.c`

The C++ Standard requires that destructors be called for global objects when a program exits in the opposite order of construction. Most implementations have handled this by calling the C library `atexit` routine to register the destructors. But at the kernel level we need to provide our own implementation and the following function provides this implementation.

- `int __cxa_atexit (void (*func) (void *), void *arg, void *d)`

This function is called by code injected by the compiler, when constructors of static objects are run. The last argument is the `__dso_handle` of the module from which the call is made.

- `void __cxa_finalize (void *d)`

This is called from `crtbeginM.o` during module unloading. The argument d is the `__dso_handle` of the module being unloaded. This calls destructors * of static objects defined in the module.

3.3 `lib/gcc`

This library contains selected pieces of code from `libgcc`. Following are the notable files which define necessary functions to support exception and other runtime features.

- `unwind-dw2.c`
- `unwind-dw2-fde.c`
- `crtstuff.c`

3.4 `crtstuff.c`

c-r-t in 'crtstuff' stands for 'C RunTime'.

When C/C++ code is compiled, the compiler includes constructors(also called as initialization functions) - routines to initialize data in the program when the program starts. These routines should be called before main is called. Similarly, destructors(also called as termination routines) need to be called when the program terminates.

GCC currently supports two ways of executing these constructors and destructors. One of the way is to make linker build two lists of these functions. Initialization functions(__CTOR_LIST__) and termination functions(__DTOR_LIST__). The logic for traversing these lists and executing the functions resides in `crtstuff.c`.

We have further modified `crtstuff.c`. This modified version of `crtstuff.c` has customized code that is required to provide support for writing C++ loadable kernel modules. This defines `begin_init` and `begin_fini` which are used in `kernel/module.c`

3.5 `lib/libstdc++`

This library contains code that provides support for `new, new[], delete, delete[]` and exceptions. This code depends on `libgcc`. For this to code function properly, `lib/gcc` need to ported into the kernel.

3.6 begin_include.h/end_include.h

begin_include.h and end_include.h use #define's and #undef's, respectively to redefine these identifiers to names accepted by the C++ compiler.

```
#include <begin_include.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <end_include.h>
```

The begin_include/end_include files take care of renaming C++ keywords temporarily, such as new and virtual, that Linux programmers use as variable names. It is however possible that some of the Linux source contains struct initializations that are incompatible with C++, which causes problems if such structs compiled inside the kernel proper (with the C compiler) need to be referenced from the C++ code. These initializations must currently be changed by hand.

Chapter 4

Conversion process

4.1 Introduce C++ compiler

To compile libsupc++ as well as our `C++` code we need to let the Kbuild know that a different compiler need to be used for compiling these files. So, some changes need to be introduced in the parent Makefile. These change can be see using the following commands.

- `git diff Makefile`
- `git diff scripts/Makefile.build`
- `git diff scripts/Makefile.lib`
- `git diff lib/Makefile`

4.2 Include header files

When we compile the kernel, few header files are generated depending on the configuration (.config) values. These header files usually reside in `asm/mach-generi`, `asm/mach-generic`, `include/generated`, etc. Depending on kernel version and architecture, the files may be generated in other directories too.

All the directories in which the header files are generated need to passed to the compiler. There are many ways to do it. In our case we modify `NOSTDINCXX_FLAGS`.

For example, to include `include/asm/mach-generic` in the path add following line to `Makefile`

```
NOSTDINCXX_FLAGS := -I$(srctree)/include/asm/mach-generic
```

4.3 Port libraries

This is the most important step. By porting these libraries into kernel we are essentially adding all the support that is required. It's not advised to skip porting some files unless you know what you are doing.

Complete adding `C++` runtime support by adding following libraries in the corresponding paths. Also make appropriate changes to relevant Makefiles.

- `cxa_atexit.c`
- `lib/gcc`
- `lib/libstdc++`
- `include/c++`

Many errors occur when we try to compile the above the libraries with the kernel. Most of these errors are trivial. Fixes for the non-trivial errors are available in the existing MOOL kernel code.

4.4 Run `C++` "Hello World" program

Write a simple `C++` program to log "Hello World" to the kernel log and call this functions in `init/main.c` and compile the codebase. If the compilation is successful, it means that the C++ runtime support is added to the kernel. The kernel need to booted to verify that "Hello World" is printed in the kernel logs.

4.5 Add support for `C++` LKM

When creating an ELF executable, GNU g++ secretly links two object files at the front and the back, `crtbegin.o` and `crtend.o`. This is necessary to ensure that global constructors and destructors are run, and that the Dwarf2 frame info is registered to enable exceptions. `g++` adds initialization code into the `.init` ELF section and cleanup into the `.fini` section. To allow exceptions and global constructors to be used in the kernel the Makefile rule for the kernel image and kernel modules is modified to link with those two files. We also need to ensure that the initialization routines are called, since the kernel module loader in Linux pays no attention to the ELF `.init` section. We can accomplish this by using preprocessor macros, that change the definition of the module initialization functions, `module_init` and `module_exit` in `include/linux/init.h`

4.6 Port system call filters

Base class for system call filter framework resides in the following files:

- `include/c++/syscallwrapper.h`
- `kernel/syscallwrapper.cc`

Two filters are implemented using this system call filter framework.

1. SMS counter filter
2. SMS geofence filter

4.6.1 SMS counter filter

The code that's relevant for SMS filter is in the following files:

- `c++/syscallwrapper_sms.h`
- `kernel/syscallwrapper_sms.cc`
- `kernel/smsfilter.cc`

To make SMS filter work on newer kernels, `sendmsg` system call need to modified. This system call is written in `net/socket.c`. The changes that are made to `sendmsg` system call need to be migrated to the new kernel. These migrations need to done in such way that it is compatible with interception code in `kernel/syscallwrapper_sms.cc`

4.6.2 SMS geofence filter

Most of code for geofence filter is common with SMS counter filter. For this filter, the location information is passed to the kernel using an user space application. The kernel uses this information to block or allow SMS.

`COUNTER_FILTER` variable can be used to switch between counter filter and geofence filter.

4.7 Version mismatch

Existing libraries compile with the kernel version 3.18 and below. For newer kernel these libraries may not compile without modifications. This need to be kept in mind while porting these libraries to newer kernels.