Porting Musl to the M3 microkernel
TU Dresden

Sherif Abdalazim, Nils Asmussen

May 8, 2018

Contents

Chapter 1

Abstract

Today's processing workloads require the usage of heterogeneous multiprocessors to utilize the benefits of specialized processors and accelerators. This has, in turn, motivated new Operating System (OS) designs to manage these heterogeneous processors and accelerators systematically.

M3 [9] is an OS following the microkernel approach. M3 uses a hardware/-software co-design to exploit the heterogeneous systems in a seamless and efficient form. It achieves that by abstracting the heterogeneity of the cores via a Data Transfer Unit (DTU). The DTU abstracts the heterogeneity of the cores and accelerators so that they can communicate systematically.

I have been working to enhance the programming environment in M3 by porting a C library to M3. I have evaluated different C library implementations like the GNU C Library (glibc), Musl, and uClibc. I decided to port Musl as it has a relatively small code base with fewer configurations. It is simpler to port, and it started to gain more ground in embedded systems which are also a perfect match for M3 applications.

Porting a C library to a non-POSIX compliant OS is a challenging task full of trade-offs. There is a long list of system calls that are mostly not supported in a microkernel, and caution is needed to translate the supported system calls to what is available. Also, M3 uses Scons [13] as it's build environment, while Musl uses Autotools [4]. Bridging the gap between the two build systems while separating the changes outside of Musl repository was a priority. I wanted to make sure it is easy to pull upstream changes.

To validate my work I have ported part of Unix coreutils. I have ignored the utilities that are dependent on those system calls that M3 does not support. In this report, I will discuss the steps I have taken to port Musl and Busybox coreutils. I will talk about some of the challenges I had, and I will end up with an evaluation.

Chapter 2

Introduction

2.1 Background

The number of processor cores per system has been increasing rapidly in recent years. The main reason is the power wall. Processors manufacturers have given up increasing single processor clock speeds due to the increase of power consumption and the generated heat to unmanageable levels. Hence, the computer industry has turned to multiprocessor systems. Symmetric Multiprocessor (SMP) has been there for years. Now with the current trend of Internet of Things (IoT) and mobile computing, the interest in heterogeneous multiprocessor system-on-chip (MPSoC) has increased rapidly. It is quite standard now to have a main general purpose processor and several accelerators and specialized processors on the same chip. Heterogeneous MPSoCs serve the need for graphics, audio, and real-time processing. That, in turn, introduced a significant challenge to OS developers. Designing an OS for SMPs has been standardized for years. Designing for heterogeneous MPSoCs is quite complicated.

There are several attempts done to tackle this problem. As an example, Linux is built to execute a kernel on SMP. Popcorn Linux [11] and K2 [14] have been trying to execute Linux on heterogeneous cores. They still depend on processor features like privileged mode, exceptions, and memory management unit (MMU). Thus they still handle accelerators as second-class citizens (Devices).

Multi-kernel designs like Barrelfish [12] supports heterogeneous cores. But yet it still requires message passing in a traditional form and needs memory isolation and management.

NIX [10] introduced the notion of application cores. Application cores are time-sharing cores which do not support running a kernel, but it still depends on message passing in a traditional form, so an MMU is required.

2.2 M3

M3, on the other hand, attempts a hardware/OS redesign. M3 introduces a new hardware component next to each compute unit. The hardware component is called data transfer unit (DTU) and creates a unified interface for all compute units. The DTU supports message passing and memory access. Since compute units can only communicate via the DTU, compute units can be isolated by controlling their DTUs. This allows to treat all kinds of compute units, ranging from general purpose cores to accelerators as first-class citizens, because the compute units do not need to provide architectural support for OSes (user/kernel mode, memory management unit, etc.).

M3 manages cores as processing elements (PEs). Each PE has its own DTU. DTU consists of several End Points (EPs). Each endpoint can be configured to be a sending endpoint, a receiving endpoint, or a memory endpoint. Endpoint configuration registers are only writable by Kernel PEs, while command registers are writable by the application PEs as well. In the beginning, all registers are writable by All PEs, and the kernels downgrade the permissions at the applications PEs during boot time.

M3 provides a library, which delivers abstractions for communicating with the kernel or OS services, accessing files, using the DTU, etc. M3 also offers a small subset of a C library. This added limitations on the applications which can run on M3. I have been working to replace this library with a full-fledged C library. Choosing the right C library to port is vital for M3's application domain.

I will discuss different design choices to choose a C library in section 3.1, then I will go briefly over alternative C library implementations in section 3.2. I will give an overview of M3 and Musl build systems in section 4.1. In section 4.2 I will discuss how I laid down the repository configurations. I will end my discussion with an Evaluation in section 5.

Chapter 3

Picking a C library

3.1 C libraries design factors

There are several decisions when it comes to picking a C library for an OS. Some OSes require flexibility via different configurations so that they can be deployed on different systems with varied requirements. For example, a C library used on a server with 32GB of RAM that mostly runs a single application needs to allocate memory in huge chunks so that it can efficiently handle application memory. In contrast, a library running on an embedded system with 512MB of RAM needs to allocate memory in smaller chunks and release them at first opportunity.

Another choice is the number of architectures that the C library supports. An OS that needs to support several architectures needs a C library that supports at least those architectures. But again increasing the system support comes with the increase in codebase and maintainability cost.

In all cases increasing the codebase hinders the porting process. It will require a careful testing strategy to make sure nothing is broken. It will also discourage fetching upstream updates to avoid breaking the system. In the next section, we will look at different open source C library implementations.

3.2 Alternative C libraries

There are different open source C library implementations available. They serve different purposes and have different goals in mind. For example the de facto standard in Linux distribution is glibc [3]. This choice is mainly because of its wider system support and configuration choices that serve different purposes. But that comes at the cost of a huge codebase.

On the other hand, we have uClibc [7]. uClibc was considered the choice of embedded systems for years. It has a very small codebase and light implementation that matches the requirements of embedded platforms. Unfortunately, uClibc has not received any updates for years. The development has been con-

tinued in a fork called uClibc-ng [8] in 2014, but I avoided that choice as the project is still new and I haven't seen it in extensive use yet.

My choice came to Musl [6]. It is very suitable to M3 applications as it has a very lightweight implementation which can scale well from biggest to smallest PE. It has a small codebase and very few configurations. That means it would require less effort to fetch upstream updates.

Chapter 4

Porting Musl

4.1 M3 and Musl build systems

While M3 uses Scons [13] as its build system, Musl uses GNU Autotools [4] which is a dominant choice in open source projects. This difference in build systems created another challenge when integrating the two build systems. While Scons uses Python scripts as means to describe the build system, Autotools uses make and config files to configure and build the system correctly. I will briefly discuss both build systems and how I tackled the differences between them.

4.1.1 Scons

Scons is a relatively new build system. Version 1.0.0 was released in August 2008. Scons is written in Python, and it provides an elegant build system. It leverages the power of Python to create a very flexible build environment. Scons searches for a file named `SConstruct` which serves as the entry point for Scons. The `SConstruct` script can reference subsidiary files named `SConscript`. This tree structure serves as a basis for hierarchical builds.

Scons supports different types of signature systems to give a signature of a file. The MD5 sum is the default signature subsystem but other signatures can be used like timestamps, or a user-provided signature can be used. Using MD5 signatures instead of timestamps solves an issue in build systems in a distributed environment when a slight difference in clocks in two systems can cause dependency to be missed.

In general, Scons is very flexible and powerful. It can produce a very efficient and highly maintainable build system.

4.1.2 GNU Autotools

GNU Autotools is the dominant build system in open source environments. It has been around for many years and provides a very stable and widely accepted

build environment. Autotools are based on Autoconfig and Automake. Autoconfig can generate different make files based on the libraries and tools available in the system. Autotools is based on timestamps which can causes issues with distributed environments. Musl uses Autotools as its build system which needed to be integrated with M3's Scons build system.

4.1.3 Integrating Autotools with Scons

To be able to trigger Autotools from Scons, there are two options. Either execute `configure` and `make` directly from Scons as a build step or emulate the actions that are done by Autoconfig like generating needed files and building the files directly via Scons. I picked up the second approach because I needed the flexibility to control what to build. For example, I have ignored malloc implementation from being built as malloc family of functions are provided directly by M3.

4.2 Repository configuration

I started the porting effort by picking the repository configuration. I wanted to separate all custom changes I make to Musl from its source code. I opted for having Musl as a git submodule that is created inside `/src/libs/c`. Below is the directory structure followed by an explanation for each item.

```
src/libs/c/
├── musl/
├── obj/
├── SConscript
└── src/
```

musl Musl source code

obj generated files as part of configuration step

SConscript build script

src Arch specific code lives here

As you can see I have separated all Musl source files from any files, I modified or added. I added the list of source and header files to compile in the `SConscript`. I would not have got the same flexibility if I had opted to use Scons `MakeBuilder` [5].

4.3 Compilation

We can divide porting a C library into three parts. Free-standing code which is independent of architecture or OS-specific details. An OS dependent code that

uses OS system calls. Architecture-dependent code, which usually contains some architecture specific implementations or optimizations.

Porting the free-standing code is quite simple. Compiling the code in the supported architecture is enough for it to work. And hence I started by compiling the string library as it mostly uses free-standing code. I followed that by the math library. In Musl libc contains both `libc` and `libmath` in the same executable.

M3 had already a small subset of a C library. I gradually added parts of Musl to it. Once I had a good portion of Musl compiling, it was time to replace the old c library by Musl. Unfortunately, M3 Kernel is dependent on some functions that are provided by this subset of C library, so I had to keep it around. I have moved it into its separate library.

4.4 Testing

At this point most of Musl code compiles, but I have not tested if it's functioning yet. I have started by running a Hello world application to verify basic functionality. I wanted to get `printf` to work first to be able to print to the screen from my test programs. `printf` did not work out of the box. Musl was still using the wrong system call for write. It was required to add some OS specific code now.

4.4.1 Syscalls

M3 uses IPC for communication and syscalls. It also provides a lot of functionality that is equivalent to syscalls in monolithic kernels as a C++ library. As an example `write` *syscall* is supplied from an output stream objects provided by M3. To support this, I created a thin wrapper as a library. I called it `libsyscall`. This library wraps C++ objects as C functions. I created an architecture specific file to hold syscall routing. I routed the syscalls using one switch statement. Having one large switch statement requires optimization, but I left that for future work. I have routed the syscalls by calling the wrapper function in `libsyscall` based on the syscall number. I used the same syscall numbering as in Linux.

`write` *syscall* started to work which in turn got `printf` and the Hello world example to work. Similarly, an input stream object supplies `read` *syscall* functionality. I also created a wrapper for it. Below is a sample wrapper implementation for `read`:

```
extern "C" ssize_t m3_read(int fd, void *buf, size_t count)
{
    m3::File *in_file = m3::VPE::self().fds()->get(fd);
    if (in_file == nullptr)
        return -EBADF;
    return in_file->read(buf, count);
}
```

Chapter 5

Evaluation

5.1 Porting Busybox coreutils

At this point, I realized that I need some programs to test with and help me in my *syscalls* mappings. I have decided to port coreutils to do that. Again, I had to choose between GNU coreutils [2] and Busybox [1]. I chose Busybox as it is more lightweight than GNU coreutils. It has a simple configuration utility that supports in choosing which coreutils to port. It also has a much smaller codebase in comparison to GNU coreutils. It has way more functionality than just coreutils; it provides some network services, some editors and a lot more. That makes it interesting as it opens the possibility to enhance the environment of M3 in the future.

Busybox has an elegant structure. It provides all its services as a single binary. That allows it to reuse most of its code and also yield a much smaller binary footprint. Busybox is created for embedded systems which have limited disk space and memory.

To port Busybox I followed the same steps I did in Musl. I added Busybox's code as a git submodule inside `src/apps/utils/busybox/src/busybox/`. My first attempt was to copy the source file for each coreutils function I wanted to port outside of the tree, modify the file to act as an independent source with its own `main` function and then bring in any dependency it requires during the compilation. Busybox shares a huge chunk of code. This code resides in `libbb` directory. So I have emulated this structure, but I created a separate library and called it `libbb`. I was linking my programs to this library. As a result, the binaries' size increased rapidly as all binaries were statically linked.

My second approach was to compile directly from the source of Busybox. I used the entry point from Busybox. Busybox is deployed as one binary, called `busybox`. It has hard or soft links pointing to it. It's `main` function picks the correct function to call based on the executable name it was called with, i.e. `argv[0]`. A big table is generated as part of Busybox configuration in `busybox/include/applet_tables.h`. This table lists all applet names mapped

to its entry point function in alphabetically sorted order. Busybox do a binary search in that table to find the correct applet function to call and dispatches this function.

Again, I followed the same approach; I configured Busybox with least number of applets to test. I compiled Busybox entry point file along with the applets I needed to port, and I added the dependencies from Busybox as required.

Chapter 6

Conclusion

In this work, I have ported Musl C library to M3 OS and to verify my work I have also ported part of Busybox coreutils. I have ported 66 tools from Busybox coreutils as well as 27 syscalls. Examples of the coreutils I have ported are File System manipulation tools like `cp` and `rm`. String manipulation tools like `uniq`, `sort`, and `wc`. Some compression functions like `bzip2` and `lzma`. And much more.

I have implemented 27 *Syscalls*, as can be seen below:

- `access`
- `close`
- `dup2`
- `exit`
- `fchmod`
- `fcntl`
- `fstat`
- `ftruncate`
- `link`
- `lseek`
- `lstat`
- `mkdir`
- `open`
- `print`
- `read`
- `readlink`
- `readv`
- `rename`
- `rmdir`
- `sendfile`
- `stat`
- `sync`
- `truncate`
- `unlink`
- `utimensat`
- `write`
- `writev`

Bibliography

[1] Busybox.

[2] Coreutils - gnu core utilities.

[3] The gnu c library (glibc).

[4] gnu.org.

[5] Makebuilder.

[6] musl libc.

[7] uclibc.

[8] Welcome to uclibc-ng! - embedded c library.

[9] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. *SIGPLAN Not.*, 51(4):189–203, March 2016.

[10] Francisco J. Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ron Minnich, and Enrique Soriano-Salvador. Nix: A case for a manycore system for cloud computing. *Bell Lab. Tech. J.*, 17(2):41–54, September 2012.

[11] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 29:1–29:16, New York, NY, USA, 2015. ACM.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[13] William Deegan. A software construction tool.

[14] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. *SIGPLAN Not.*, 49(4):285–300, February 2014.