PORTING OF C LIBRARY

Ludek Dolihal

Doctoral Degree Programme (2), FIT BUT

E-mail: xdolih00@stud.fit.vutbr.cz

Supervised by: Tomas Hruska

E-mail: hruska@fit.vutbr.cz

Abstract: In this article will be discussed a porting of a library of the C language into a simulator. This is needed for testing of an automatically generated C compiler.

Keywords: Porting of a library, C library, compiler testing, simulation

1. INTRODUCTION

One of the goals in out research group Lissom [1] is an automatic generation of C compilers for various architectures. To ensure that the generated compilers are compliant with the norm, it is necessary to put the generated compilers under test. Because the the whole process of compiler generation is highly automatic and we do not have all the platforms available for testing, we use simulators for the compiler testing instead of the chips or development kits. If one wants to test the compiler within the simulator, it is necessary to add the support for the C library functions into the simulator.

2. CHOOSING THE LIBRARY

As we are focused mainly on embedded systems and we design the whole process of compiler development for them we dedicated quite a lot a time to choosing the correct library. It was clear right from the beginning that glibc is needlessly large and therefore not suitable for use in embedded systems. We needed library, that satisfies following criteria:

- minimalism,
- support for porting on different architectures,
- well-documented,
- new release at least once a year,
- compatibility with glibc,
- modularity.

All this conditions were satisfied just by several libraries. Amongst those, we chose uClibc[3]. This library is largely minimalistic. It doesn't contain certain modules, because, according to the authors, it would be against the minimalism. As far as the new releases are concerned, it can be said that the library is alive. New version is released at least once a year. This is very important because we need to keep pace with the up to date versions of glibc.

Majority of programs, that use glibc, should be usable with uClibc without problems. Hence the libraries are compatible. On the other hand authors don't even try to ensure the binary compatibility with glibc, neither do they ensure the binary compatibility amongst the different versions of uClibc. The whole library is build on modular base. The modules can be without many efforts disabled during the process of compilation.

Last but not least the uClibc has already been ported to several platforms. There is a documentation dedicated to porting. Unfortunately this library is dependent on kernel header files. But during the process of porting we will get rid of these dependencies.

3. THEORY OF PORTING

The main reason for porting the library into a simulator is the fact that we need to add the support for C functions into the simulator itself. To be precise, we want to use the libc functions such as printf, malloc, free etc. in the programs, that will be used for testing of the compiler. And because we do not posses the development kits for all the platforms we use simulators instead.

If the C library support is not granted in the simulated environment, the number of constructions we can use and test is significantly limited. On the contrary the main aim of testing is to cover as wide area as possible and also try as many different combinations of functions as we can. On the other hand we focus especially on embedded systems, so we do not even try to cover all the functions provided by glibc or in our case uClibc. In fact we will use and hence test only functions that can run under the simulated environment and are useful for the programs that will be executed on the given platform. Moreover embedded systems are not designed for use of vast number of constructions that programming languages offer. Typically there is just one task, usually quite complicated, that is launched repeatedly. As we will see the functions that we will use forms just small part of uClibc. The functions that are not important to us can be easily removed via configuration interface or manually. Following categories are examples of unimportant functions:

- threads, we assume that in simple programs for embedded systems one will not use threads,
- math, functions for computing sin, cos etc.
- inet module, even though networking plays important part in modern embedded systems the module was removed,
- files and operations with files, our application do not need interface for working with files.

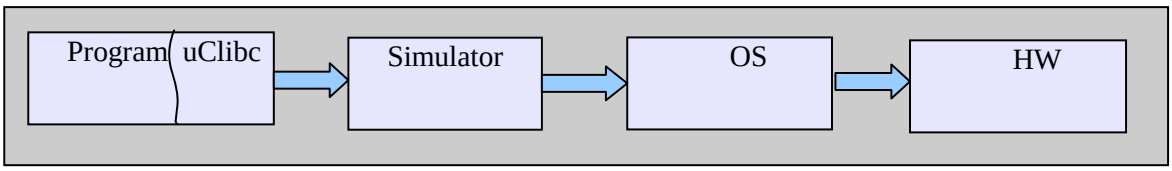

Fig. 1 Position of the library in testing system.

Now we come to the important parts of the library. Simply spoken all that really has to remain from the library are the sysdeps, (the the core of the whole system; how to allocate more memory etc.), then important modules such as stdio (for outputs, inputs) and other modules we wish to preserve. In our case we wished to preserve following parts of the uClibc library:

- stdio, this was the main reason of porting the library, to get in human readable form output from the simulator,
- module for working with strings and memory, in our applications we would like to use functions such as memcpy, strcpy, strcat etc.,
- memory functions, for example malloc, free, realloc,
- abort and exit.

Some parts of the library could not be removed because of the dependencies. According to our estimations nearly 40 percent of the library was disabled or removed.

There are several ways of building the library and also different methods of using it. There is a possibility of building a position independent code. Even though this is an interesting solution we decided against it. Instead of PIC we are going to compile the library into single object and then link it to the program statically. The position of library in the whole process of testing is shown in the figure 1.

Lets return to the functions that remain in the library. They can be divided into two groups. First group consists of functions, that are completely serviced within the simulated environment. For example function strcmp falls into this category. This function together with the declaration remains unchanged within the simulator if it is written in portable C. This functions are not tied with kernel header files so there is no need to change them.

The second group consists of functions, that are translated to the call of a system function. Function printf can be used as an example of this group of functions. The call of printf function can be divided into three phrases that are illustrated at the following picture.

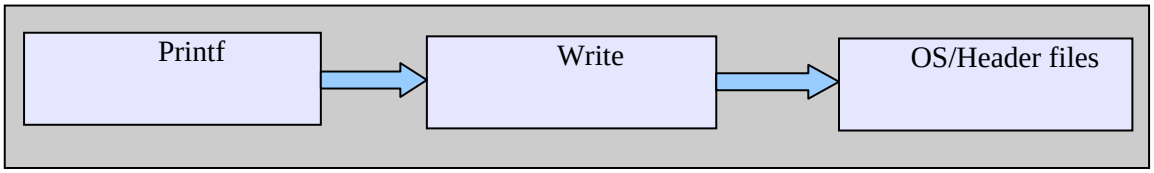

Fig.2 Scheme of calling the printf function

In the beginning, the call of printf function is translated on the call of system function, with the highest probability it is going to be the call of function write. Write, being the POSIX function, is offered by the operation system. But as we want to use the simulator on Unix platform as well as on Windows systems we have to get rid of these dependencies. To do so we will use the special instruction principle.

3.1. SPECIAL INSTRUCTION PRINCIPLE

The special instruction principle means, that we will use instruction with the opcode, that is not used within the instruction set for the special purpose. So far every architecture that was modelled within the Lissom project had several free opcodes. It is typical for the instruction sets that they do not use all provided operation codes. But in case of no free opcode this method can not be used. The special instruction principle will be used for ousting the dependencies on kernel header files.

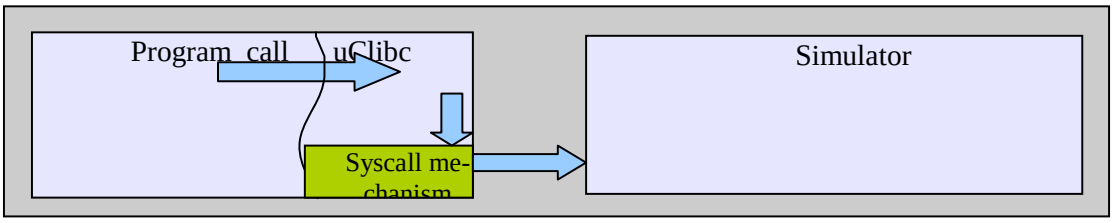

Fig. 3 Scheme of calling the simulator via uClibc layer

Functions provided by operation system are called by the syscall mechanism. Though we want to get rid of operation system dependency, we need to preserve the mechanism. The syscalls will remain in the library, but with different meaning. The file containing syscalls will be changed in the following way: in the beginning the parameters of the syscall will be placed at the given addresses in the memory and we will also define where the syscall return value will be placed. Afterwards the call of the chosen instruction will be performed. It is also possible to put the parameters into registers, but some platforms have limited number of registers, hence this method could cause problems.

3.2. SIMULATORS

All the simulators and profilers [2] are generated automatically. In the beginning all the source files are generated by specialized tools. When the generation phase is finished the simulator is build by a Makefile. It will be necessary to add into this process following information and actions:

- the instruction (opcode) that calls the system function,
- the convention for storing parameters,
- the simulator will have to recognize which system function is going to be called,
- the simulator will have to perform the call of the correct system function.

First three points will be solved within the model of an instruction set. The instruction with the op-code, that is not used will be declared. The instruction behaviour will be defined in the following way: according to the parameters the given system function call will be performed. The simulator will have to recognize the system it runs under and call the correct function. For example on Unix system it will be function write and in Windows WriteFile. This should be solved by the libc library of the given OS. The following figure demonstrates the call of special instruction.

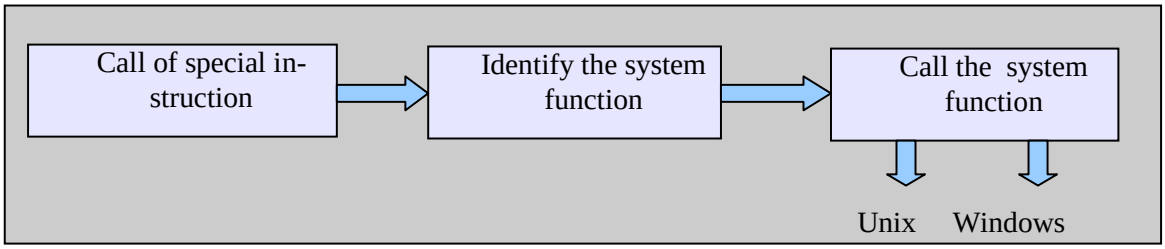

Fig 4. Calling sequence of specialized instruction

One important issue is connected with the simulated memory. As we would like to correctly simulate the operations with memory such as malloc, realloc etc. we need to tell the simulator how many memory it can simulate. This will be done most probably by the special file that will be passed to the linker. This file will contain symbols that will declare how much memory can be used.

4. PROCESS OF PORTING

One of the problems we face is that we need to have the compiler for the architecture we are developing for. In other words if we want to create a library for testing C compiler on a given platform we need another compiler for the same platform. The compiler will be used for building the uClibc because the uClibc will run under simulator of given platform. Moreover the compiler must have exactly the same instruction set. In the future we would like to use the generated compiler for building the library. This requires high quality of backendgen and generated backend.

This may be the first big problem in the whole process of porting. It is not hard to find a compiler for a given platform. Nowadays there are specialized compilers for nearly all architectures used in embedded systems. The buildroot for uClibc contains more then dozen of different architectures such as MIPS, arm, mipsel, SPARC etc. There are even different versions of the micro-architectures in case of MIPS for example.

Problem is that thanks to the aim of the whole Lissom project, we use special formats of the output files, that are no compatible with the formats used by common tools. We could use for building the library the compiler that we want to test but currently it is not stable enough for building large programs. The best solution of this problem is usually building a specialized tool-chain including GNU binutils and GNU compiler collection[4].

Currently we use for the development the set of our tools containing archiver, linker, assembler and compiler. Currently used compiler is called mips-elf-gcc. It is not generated automatically but was created specially for this purpose as our generated compiler is not yet stable enough. Linker and archiver are generated automatically.

The build-system of the library starts by parsing the configuration file and accord to the content of the file are set different macros and variables. When doing manual changes to the build-system we have basically two possibilities:

- change the configuration file or,
- do the changes later in the Makefiles.

The changes are necessary as our tools are not compatible with all the flags,that are used in the build-system. The first possibility is cleaner but the Makefiles often check if the option is present in the configuration file and ends with error in case the option is missing. Hence it is more convenient to do the necessary changes in the Makefiles. Thanks to the hierarchical structure it is in most cases sufficient to do the change in just one place.

In the theoretical part we mentioned the need to link special file containing information how much memory can be used. The file will contain symbols defining the beginning and the end of memory space that can be used. It will have the following syntax:

#file defining memory boundaries

define start 256

define stop 768

Given that the numbers are in kB the simulator can simulate up to 512 kB of memory. Character # denotes comment.

As far as the convention for storing parameters is concerned, we have chosen following approach: first parameter says which system function is going to be called. In the uClibc itself is a list of system functions for Unix systems. The rest of the parameters are parameters (2-7), that are passed to the function call. The parameters remains unchanged. The special instruction itself has no parameters.

5. CONCLUSION

In this paper was sketched the the idea of porting the library into the simulator. The motivation is quite clear: to be able to get any output in the reasonable form out of the simulator as well as to test the compiler behaviour when compiling programs containing different structures and functions. The special instruction principle was proposed. This principle allows us to identify and call given system function. After the implementation of this method we provided all the necessary functions and the comfort of testing was increased rapidly.

ACKNOWLEDGEMENTS

This research was supported by doctoral grant GA CR 102/09/H045, by the grants of MPO Czech Republic FR-TI1/038, by the grant FIT-S-10-2 and by the research plan no. MSM0021630528.

REFERENCES

[1] Lissom Project. http://www.fit.vutbr.cz/research/groups/lissom.

[2] Zdeněk Přikryl, Karel Masařík, Tomáš Hruška, Adam Husár, “Generated cycle-accurate profiler for C language”, 13th EUROMICRO Conference on Digital System Design, DSD'2010, Lille, France, pp. 263—268.

[3] uClibc. http://www.uclibc.org/

[4] GNU Operating System, http://www.gnu.org/software/