Pip Tutorial
Getting started on Intel x86 architecture

Quentin Bergougnoux

November 16, 2018

Contents

Listings

1 Requirements

1.1 Build environment

In order to create your own partitions on top of Pip, you need an appropriate development environment, relying on several required tools.

GNU C Compiler: GCC is the only C compiler known to compile Pip correctly. CLANG, for example, is not supported. To that end, you need a version of GCC capable of producing 32bits ELF binaries. The last working version is gcc v7.3.0.

Haskell Stack: Pip uses a home-made extractor to convert Coq code into C code. In order to compile this Extractor, which is written in Haskell, we use the Stack tool to download and install automatically the required GHC and libraries.

Coq Proof Assistant: Pip's source code and formal proof of its memory isolation properties are written using the Coq proof assistant. In order to compile Coq files and generate the required intermediate files for the kernel to build, you will need the 8.6.1 version of Coq. A clean way to install Coq is via OPAM and is described in Section. 2.4.

Netwide Assembler: Pip's assembly sources are compiled using the Netwide Assembler (NASM). A known working version is version 2.11.08, although any version since 2.0 should be working.

GNU Toolchain: Althought Pip is known to compile on FreeBSD and OSX hosts, these need some GNU software in order to perform the compilation, which are GNU Sed and GNU Make.

Doxygen: Pip's documentation is generated through CoqDoc (included with Coq) for the Coq part, and Doxygen for the C part. The documentation is not mandatory to compile Pip, but it is highly required that you compile it and keep it somewhere safe so you always have some reference to read if you need some information about Pip's internals.

QEMU: Although it is not required to build Pip, it is highly recommended to run Pip on emulated hardware rather than physical hardware during development. As such, QEMU is a known, multi-platform emulator, and is fully integrated into Pip's toolchain.

1.2 Get the source

Pip source: Once your development environment is ready, open a terminal emulator and clone the Pip repository at `https://github.com/2xs/pipcore`. This repository contains Pip's kernel, proof and documentation. Still, it is not ready to compile yet, as it provides no partition to run. We will cover this in the next section, "*Hello World tutorial*".

LibPip source: In order to make partition development easier, we also provide a userland library available at `https://github.com/2xs/libpip`. This library, LibPip, provides method useful for calling Pip's API, or managing Pip's data structures. It should also provide, in a near future, a way for partitions to communicate.

Digger source: As said previously, we use an extractor to convert Coq code into C code. The source is included in Pip's main repository as a submodule, run "`git submodule init`" and "`git submodule update`" to fetch it.

2 Preparing for the build

2.1 Building Digger

First of all, you need to compile the Extractor. This step is **mandatory**. The Digger extractor is compiled through the stack tool. The compilation is straightforward.

- Open a terminal emulator
- Go to Pip's repository folder
- Navigate to the `tools/digger` folder
- Import the submodule via the commands "`git submodule init`" then "`git submodule update`"
- Type in "`make`"

The compilation might complain about a missing GHC version : type in the asked command to install the required GHC version (usually "`stack setup`"), and type in `make` again to grab the required libraries and build the extractor.

2.2 Building LibPip

In order to compile a partition, you will probably need LibPip. Open a terminal emulator and go into LibPip's directory. Typing "`make ARCH=x86`" should be enough to compile the library. Remember the path to the repository : we will call it `LIBPATH` further in this document.

2.3 Configuring Pip

Before doing anything else, you need to configure the partition building toolchain.

- Open a terminal emulator
- Go to Pip's repository
- Navigate to the `src/partitions/x86` directory
- Copy the `toolchain.mk.template` file into `toolchain.mk`
- Open `toolchain.mk` with a text editor

Here, you need to specify which compiler and LibPip to use. Basically, we will use GCC as C compiler and assembler, and LD as linker. Set LIBPIP to `LIBPATH`, which we defined in the previous chapter as the path to LibPip's repository. Feel free to replace those with your favourite toolchain (e.g. i386-elf-gcc on Mac OSX).

Listing 1: `toolchain.mk` configuration

```
CC=gcc
LD=ld:w
AS=gcc
LIBPIP=~/Pip/LibPip
```

2.4 Get ready in 5 minutes

The following commands summarize the installation on a Debian 8.1 or Ubuntu 16.04.

```
> sudo apt install haskell-stack nasm doxygen opam

# Install COQ version 8.6.1
> export OPAMROOT=~/opam-coq.8.6.1
> opam init -n --comp=4.01.0 -j 2
> opam repo add coq-released \
    http://coq.inria.fr/opam/released
> opam install coq.8.6.1
> opam pin add coq 8.6.1

# check if COQ is correctly installed
> coqc -v

# Install and compile LibPip
> git clone https://github.com/2xs/libpip
> cd libpip
> make

# Clone PipCore
> git clone https://github.com/2xs/pipcore
> cd pipcore

# Clone submodule coq2c
> cd tools/coq2c
> git submodule init
> git submodule update

# Compile and install coq2c
> make          # The makefile stops with an error
> stack setup   # Install haskell stack
> make
```

3 Tutorial : Hello World !

Now that Pip is ready to compile, we will create a minimal "Hello world" partition to run on it.

3.1 Folder architecture

First, navigate to Pip's `src/partitions/x86` folder, and create a "`HelloWorld`" subfolder. There, we will need several files. Create these empty files using `touch` :

- `Makefile` : Make configuration file
- `boot.s` : Bootstrap assembly code
- `main.c` : Main C file
- `link.ld` : Linker script

3.2 Greeting the world

Once these files are created, we're finally ready to make some code. Hopefully, LibPip already provides everything we need to get some serial output.

3.2.1 Makefile

The Makefile itself isn't complicated, and is composed of the listings 3, 4 and 5. First, we need to include our toolchain configuration file, `toolchain.mk`. Then, we define some flags to pass to the compiler, assembler and linker :

Listing 2: Makefile GCC flags

```
include ../toolchain.mk

CFLAGS=-m32 -c
CFLAGS+=-nostdlib --freestanding
CFLAGS+=-I$(LIBPIP)/include/
CFLAGS+=-I$(LIBPIP)/arch/x86/include/
```

Here are the meanings of the flags.

`-m32` In case we're compiling using a x86_64/64bits GCC, we explicitly say we want to generate 32 bits code.

`-c` We're generating intermediate object files, not a full binary for each file. The linker will later link all objects files together into a single, flat binary file.

`--freestanding` We're compiling on bare-metal, but GCC provides some platform-free, bare-metal ready includes, such as `stdint.h`, which proves to be quite useful in this kind of development.

`-nostdlib` Quite self-explanatory, this flag disables the linking towards the LibC, thus eliminating the dependancy to the host system.

`-I$(LIBPIP)/include/ ...` We want to link our Hello World partition towards LibPip. Here, we specify LibPip's include directory so that we can use our library's include files, as well as LibPip's architecture-dependant includes. See section 1.2, "*LibPip source*".

Listing 3: Makefile GAS and LD flags

```
ASFLAGS=$(CFLAGS)

LDFLAGS=-L$(LIBPIP)/lib -melf_i386 -Tlink.ld
LDFLAGS+=-lpip
```

`ASFLAGS` We use GCC as assembler (wrapping GNU as). We use the same flags as for GCC.

`-L$(LIBPIP)/lib` During linking phase, we want to link our partition with LibPip. This flag specifies LibPip's directory as additional directory to search in for libraries.

`-lpip` Specifies we link our partition with LibPip (`libpepin.a`).

`-melf_i386` Specifies the output architecture of the binary. *This will be overridden in link.ld, but required during linker invocation.*

`-Tlink.ld` Specifies the linker script file, which is given in section 3.4, "`link.ld`".

We then define some generic rules for our sources, and invoke the required compiler for each one, calling the linker once everything has been done.

Listing 4: Makefile rules

```
ASSOURCES=$(wildcard *.s)
CSOURCES=$(wildcard *.c)

ASOBJ=$(ASSOURCES:.S=.o)
COBJ=$(CSOURCES:.c=.o)

EXEC=HelloWorld.bin

all: $(EXEC)
        @echo Done.

clean:
        rm -f $(ASOBJ) $(COBJ) $(EXEC)

$(EXEC): $(ASOBJ) $(COBJ)
        $(LD) $^ -o $@ $(LDFLAGS)

%.o: %.S
        $(AS) $(ASFLAGS) $< -o $@

%.o: %.c
        $(CC) $(CFLAGS) $< -o $@
```

3.2.2 boot.s

The file dumped in listing 6 is the partition's entrypoint. We will basically do the same as a stand-alone kernel, booted via Multiboot :

- Declare that our C main() is outside of the assembly file
- Push EBX onto the stack (except that it will contain Pip's boot info, not Multiboot info)
- Disable interrupts virtually (using LibPip's Pip_VCLI() method)
- Call our C main() method
- If we happen to return from main(), loop forever

In order to use LibPip's Pip_VCLI() call, we need to declare Pip_VCLI() external as well. We also want our `boot` method available through the linking phase. The code is then as straightforward as following.

Listing 5: boot.s

```
.section .text
.global boot
.extern main
.extern Pip_VCLI
boot:
    /* Push EBX onto the stack */
    push %ebx

    /* Disable interrupts */
```

```
    call Pip_VCLI

    /* Call main */
    call main

    /* Fallback */
loop:
    jmp loop
```

3.2.3 main.c

Our C main's main purpose is to display our "Hello World!" message. LibPip provides a `Pip_Debug_Puts(char* msg)` method to that end. We just need to include the corresponding header (`pip/debug.h`), then call this method in our `main()` method. We also include `pip/fpinfo.h`, which declares the structure we pushed on the stack from `EBX` in `boot.s`.

Listing 6: main.c

```
#include <stdint.h>
#include <pip/fpinfo.h>
#include <pip/debug.h>

void main(pip_fpinfo* bootinfo)
{
        Pip_Debug_Puts("Hello␣world!\n");
        for(;;);
}
```

3.2.4 link.ld

The linker script declares the output format, entrypoint and sections of our binary. The thing to remember is that our binary will be loaded at address `0x700000`, and must be a flat binary (`binary` output type). We then declare as many sections as needed, the only required thing being that our code section, `.text`, must be at `0x700000`, with our entrypoint at this very address.

First we define our output format and entrypoint.

Listing 7: link.ld output and entrypoint

```
OUTPUT_FORMAT("binary")
ENTRY(boot)
```

Then, we define our sections. First, we define out .text mandatory section, and as many sections as required after then. A standard, basic linker script might declare the following sections and symbols :

Listing 8: link.ld sections

```
SECTIONS
{
  .text 0x700000 : /* Partition load address */
  {
    text = .; _text = .; __text = .;
    *(.text)
    . = ALIGN(0x1000); /* Page-align sections */
```

```
  }

  .data :
  {
     data = .; _data = .; __data = .;
     *(.data)
     *(.rodata)
     . = ALIGN(0x1000);
  }

  .bss :
  {
     bss = .; _bss = .; __bss = .;
     *(.bss)
  }
  end = .; _end = .; __end = .;
}
```

3.3 Making the world

Once all our files are created and filled, we can generate our partition binary by invoking "`make`", which should produce the following output :

Listing 9: Partition make output

```
gcc -m32 -c -I. --freestanding -nostdlib
        -I/home/user/Pip/LibPip/include/
        -I/home/user/Pip/LibPip/arch/x86/include/
        boot.s -o boot.o
gcc -m32 -c -I. --freestanding -nostdlib
        -I/home/user/Pip/LibPip/include/
        -I/home/user/Pip/LibPip/arch/x86/include/
        main.c -o main.o
ld boot.o main.o -o HelloWorld.bin
        -L/home/user/Pip/LibPip/include/lib
        -melf_i386 -Tlink.ld -lpip
Done.
```

3.4 Ruling the world

Great, if you reached this point, your Hello world partition just compiled successfully! All that remains is to compile Pip with your partition on top of it.

- Go to Pip's repository root folder
- Run `make PARTITION=HelloWorld partition kernel`

Your Pip binary should be ready. You can run it on QEMU by running `make qemu`, which should display "Hello world!" after a few seconds.

Congratulations !