1. Debugging tools:

I will introduce three useful debugging tools in Pintos.

1.1 Printf()

- Printf() is implemented in Pintos, we can call it from practically anywhere in the kernel.

  Pintf() is very useful. It can help us to figure out when and where something gose wrong. We can use it as the normal C language.

1.2 ASSERT()

- When we read the source code, we can see many ASSERT().

  Assertions are useful because they can catch problems early, before they'd otherwise be noticed. Ideally, each function should begin with a set of assertions that check its arguments for validity. (Initializers for functions' local variables are evaluated before assertions are checked, so be careful not to assume that an argument is valid in an initializer.) You can also sprinkle assertions throughout the body of functions in places where you suspect things are likely to go wrong. They are especially useful for checking loop invariants.

  Pintos provides the **ASSERT** macro, defined in , for checking assertions.

Macro:ASSERT(expression)

- Tests the value of expression. If it evaluates to zero (false), the kernel panics. The panic message includes the expression that failed, its file and line number, and a backtrace, which should help you to find the problem.

1.3 GDB

GDB is the most important tool for us. You can run Pintos under the supervision of the GDB debugger.

GDB macro configuration file:

1. Enter $PINTOSDIR /src/utils

2. Open pintos-gdb with vim or other editor. Setting the “GDBMACROS=” as the location of the gdb-macros in your computer.

Eg:

change:

```
GDBMACROS = /usr/...
```

to:

```
GDBMACROS = /home/liang/pintos/src/misc/gdb-macros
```

Close and save it.

We can use the macros provide by pintos.

1.3.1Using GDB

How to start debugging?

eg:

After we complied the program.

Open a terminal. Then we need to enter **/src/threads/build**.

Input: **pintos --gdb –- run alarm-multiple**

please pay attention to the space between the —gdb and run.

And then, open another terminal. Enter **/src/thread/build**

input: **pintos-gdb kernerl.o**

then, input: **debugpintos or target remote localhost:1234**

Now,we can use GDB to do debug.

You can read the GDB manual by typing info gdb at a terminal command prompt. Here's a few commonly useful GDB commands for part 1:

GDB Command:c

Continues execution until Ctrl+C or the next breakpoint.

GDB Command:break function

GDB Command:break file:line

GDB Command:break *address

Sets a breakpoint at function, at line within file, or address. (Use a 0x prefix to specify an address in hex.)

Use `break main` to make GDB stop when Pintos starts running.

GDB Command: p expression

Evaluates the given expression and prints its value. If the expression contains a function call, that function will actually be executed.

GDB Command: l *address

Lists a few lines of code around address. (Use a 0x prefix to specify an address in hex.)

GDB Command: bt

Prints a stack backtrace similar to that output by the backtrace program described above.

GDB Command: p/a address

Prints the name of the function or variable that occupies address. (Use a 0x prefix to specify an address in hex.)

GDB Command: diassemble function

Disassembles function.

We also provide a set of macros specialized for debugging Pintos, written by Godmar Back gback@cs.vt.edu. You can type help user-defined for basic help with the macros. Here is an overview of their functionality, based on Godmar's documentation:

More information:

http://web.stanford.edu/class/cs140/projects/pintos/pintos_10.html#SEC145

1.3.2 Using GDB debug assembly

make breakpoint

When we need make a breakpoint in assemble program. We can't use **break function**, we need to use break *address(e.g b *0x7c00)

Some command for debug assemble program:

GDB Command: ni (next instruction)

Used for single step.

GDB Command: si (step instruction)

Used for single step.

GDB Command: display

Register window. This command coule be used to watch the value in register.

e.g ``` (gdb) display /x $eax

(gdb) display /x $ebx

(gdb) display /x $ecx

(gdb) display /x $edx ``` If you want to watch registers, use **'p expression'**.

e.g `p $eax`

1.4 Assignment

Description

In this assignment, you will use the GDB to trace the loader and kernel's code to understand how Pintos is loading. If you are not already familiar with x86 assembly language, the PC Assembly Language Book is an excellent place to start.

Warning: Unfortunately the examples in the book are written for the NASM assembler, whereas we will be using the GNU assembler. NASM uses the so-called Intel syntax while GNU uses the AT&T syntax. Luckily the conversion between the two is pretty simple, and is covered in Brennan's Guide to Inline Assembly.

After you have learned the basic syntax of x86 assembly language, the next step is to learn how Pintos is loading. Make sure you have read through from **"3.1.1.1 The Loader"** to **"3.1.1.4 Physical Memory Map"** before you do the requirement.

In original Pintos, it does not have GDB supportment for tracing the BIOS and loader's 16bit code. So we have modified the **gdb-macro** to support this. You can optionally download our 'gdb-macro', replace the original **'src/misc/gdbmacro'** or insert original **'gdbmacro'** with following code inside the function named `hook-stop` :

```
# There doesn't seem to be a good way to detect if we're in 16- or
# 32-bit mode, but we always run with CS == 8 in 32-bit mode.
    if $cs != 8
        # Translate the segment:offset into a physical address
        printf "[%4x:%4x] ", $cs, $eip
        x/i $cs*16+$eip
    end
```

Requirement

1. Run `pintos --gdb -- -q` and connect to gdb in a new terminal. Set a breakpoint at address `0x7c00` , which is where the boot sector will be loaded(in file loader.S). Continue execution until that breakpoint. Use GDB's si (Step Instruction) command to trace into the Loader for a few more instructions, and try to guess what it might be doing. No need to figure out all the details - just the general idea of what the Loader is doing first.

2. After that, you can set a breakpoint in main and continue. And then, use gdb to trace through until Pintos booting complete. Try to figure out the order of modules of kernel initialization.

3. Download and fill the DESIGNDOC and archive DESIGNDOC into a zip file named sid*nameInPinYin*vid.zip(e.g 12330441*zhangsan*v0.zip) and upload to the ftp://my.ss.sysu.edu.cn/~wh/homework*upload/13%BC%B6%20%B2%D9%D7%F7%CF%B5%CD%B3/assignment2/*.