Getting Started with PintOS

This is self-study and not graded but you need to do this for doing the next assignment

Introduction

For the next couple of laboratory assignments in this course we will use the PintOS operating system. PintOS was developed for the CS140 course at Stanford University. While it is a real operating system, capable to run on a x86 computer, PintOS was designed with teaching in mind and has a simple structure. This makes the code easy to understand and modify.

In the next couple of laboratory assignments, you will extend this operating system by implementing system calls, user-space processes and virtual memory.

Assignment

The purpose of this assignment is to set up PintOS and to familiarize yourself with the tools and the source code structure. This is ungraded and is only to familiarize yourself with the tools. The next coupel of assignments will enable you to run user programs, use virtual memory and possible file systems.

PintOS setup

Task Download the PintOS source Code from Moodle and then execute the following command:

```
tar -zxvf pintos.tgz
```

You will use your clone of the repository to manage the changes you are making to the PintOS code. We strongly recommend that you use a version control system to keep track of your changes.

In case you don't have qemu, you would need to install that using apt-get but you probably have it courtesy one of the labs.

The official PintOS documentation is available in `pintos/doc/pintos.pdf`.

Task Read the following sections from the PintOS documentation: 1.1, 2.1, 2.2, Appendix A, Appendix E, Appendix F.

To be able to build and run PintOS on your machine, you need to have `make, ~qemu`, `gcc` and `gdb` installed.

To set up the environment variables needed by PintOS tools, run:

```
cd pintos

source setup.sh
```

Compile PintOS by running:

```
cd pintos/src/threads
```

```
make
```

To run PintOS, move to the build directory and run

```
cd build

pintos --qemu -- run alarm-single
```

This should open a new window displaying a terminal and some text output. If your programs do not return to the command line, you can stop their execution at any time by typing Ctrl-C. If you want to have the output printed in the same terminal from which you are running PintOS, you can run:

```
pintos -v --qemu -- run alarm-single
```

For other available options, run pintos --help.

Running a program on PintOS

PintOS comes with a set of ready-made programs, that resemble the programs with the same name on an UNIX system. You can find these programs in pintos/src/examples. To build them, just move to that directory and run make.

To be able to boot, PintOS needs a disk image. To create that disk image, do the following:

```
cd pintos/src/userprog

make

cd build

pintos-mkdisk fs.dsk --filesys-size=2
```

This creates a disk image, fs.dsk, having a size of 2 MB. To format this image, run:

```
pintos --qemu --disk=fs.dsk  -- -f -q
```

To run a program, copy it to the disk and then pass the run command followed by the program name to the kernel, as follows:

```
pintos --qemu --disk=fs.dsk -p ../../examples/ls -a ls  -- run ls
```

Note that you will have to do a make inside ../../examples/ in order to compile the examples, including the ls. This command copies the file from ../examples/ls to ls and then provides the run ls command to the kernel, which launches the program into execution after boot. Since our OS is currently lacking support for system calls, this will print: System calls not implemented.. Don't worry, we will get to implement system calls during the next assignments.

You may see the following output

```
Kernel PANIC at ../../filesys/fsutil.c:122 in fsutil_extract(): halt: create failed
```

which means that the file you are trying to copy already exist on the disk. Format the disk image, as explained above, and try again.

Debugging PintOS

The easiest way to debug a program is to sprinkle calls to `printf` and check the output. You can use this in PintOS, but it requires that the program is recompiled every time you want to know something new. A faster way to debug is by using a debugger, such as `gdb`. We have seen that the program we are trying to run prints an error message, `System calls not implemented`. Let's try to identify how the program ends up printing that message.

First, copy the file you want to debug onto the disk, if it's not already there:

```
pintos -v --qemu --disk=fs.dsk -p ../../examples/halt -a halt  -- -q
```

If this file is already there you will get a 'Kernel PANIC...:create failed' message. For more tips on handling odd situations please see the **More Tips** at the end of the document.

Then start `PintOS` waiting for a debugger to connect to it (`--gdb`):

```
pintos -v --gdb --qemu --disk=fs.dsk -- run halt
```

In a separate terminal, run the following commands to start `gdb` and load the symbols from the `kernel.o` file.

```
cd pintos/src/userprog/build

pintos-gdb kernel.o
```

Locate in the source code where the error message code is printed. And then add a breakpoint on that location (the line number may not be correct - see `list syscall_handler`):

```
(gdb) break src/userprog/syscall.c:93
```

and then tell `gdb` to connect to the emulator

```
(gdb) target remote localhost:1234
```

Since you will run this command many times, there exists a macro for it, `debugpintos`. The debugger is now stopped at the entry point of the kernel code. We want to move on from here, there issue the `continue` command.

```
(gdb) continue
```

The execution proceeds until the breakpoint we have set earlier is hit:

```
(gdb) continue

Continuing.
```

```
Breakpoint 1, syscall_handler (f=0xc010afb0) at ../../userprog/syscall.c:93
93        printf("System calls not implemented.\n");
```

Use backtrace (shorthand bt) to view the current call stack

```
(gdb) bt
#0  syscall_handler (f=0xc010afb0) at ../../userprog/syscall.c:93
#1  0xc0021d74 in intr_handler (frame=0xc010afb0) at ../../threads/interrupt.c:367
#2  0xc0022061 in intr_entry () at ../../threads/intr-stubs.S:37
#3  0xc010afb0 in ?? ()
#4  0x080480f5 in ?? ()
```

and frame N to select a given stack frame

```
(gdb) frame 1
#1  0xc0021d74 in intr_handler (frame=0xc010afb0) at ../../threads/interrupt.c:367
367      handler (frame);
```

To print the contents of a variable, use print (shorthand p) or p/x for to print hexadecimal values:

```
(gdb) p frame
$1 = (struct intr_frame *) 0xc010afb0
(gdb) p *frame
$2 = {edi = 0, esi = 0, ebp = 3221225408, ...}
(gdb) p/x *frame
$3 = {edi = 0x0, esi = 0x0, ebp = 0xbfffffc0, ...}
```

To execute the next statement, use the next (shorthand n) command:

```
(gdb) next
94        thread_exit();
```

To resume execution, run the continue (shorthand c) command. To stop the execution at any time, type Ctrl-C and type quit to exit gdb.

Implementing an alarm clock

The timer_sleep(int64_t ticks) function suspends the execution of the calling thread for at least int64$_t$ ticks. The current implementation of the function is a busy-wait loop which is not efficient. We want instead to block the current thread from being scheduled until the desired amount of ticks has passed. This can be achieved by calling the thread_block() function. Once the thread is blocked, it will not be scheduled for execution. Modify the `timer_interrupt()` function to schedule sleeping thread if their sleep time has expired. You can add fields `struct thread` to keep track of the time a thread has to sleep. You can build the code by

```
cd src/threads

make
```

The following tests should pass when you run `make check`

```
alarm-single

alarm-multiple

alarm-simultaneous

alarm-zero

alarm-negative
```

Hand-in

This assignment is not graded. All you have to do is to submit a document containing answers to the following questions:

- After you compile the code what is the size of the directory.
- How many new files are created.

Hints

- This assignment does not require writing any assembly code.
- The approximate size of the code changes is:

```
src/devices/timer.c    | 53 +++++++++++++++++++++++++++++++++++++++++++++++
++++++

src/threads/Make.vars |  4 ++--

src/threads/thread.h  |  5 +++++
```