CS 1217 Spring 2023

Assignment 3

Due - Monday, 20th February, 11:59 PM

(60 Points)

General Instructions

1. **Only one** person of a team (of 2) should submit the assignment. This should be the same person who had submitted assignments 1 and 2.
2. This assignment constitutes **3% of the total course grade**.
3. Please refer to the class webpage for notes on late assignment submission policy. ***No extensions, other than the ones allowed by class policy, will be granted***.
4. **Important**: The assignment submission should also contain a pdf file listing the contributions of individual team members.
5. Boot xv6 is worth 10 points. Exercises 1 and 2 are worth 25 points each.

Assignment Specific Instructions

Use the following GitHub Classroom link to gain access to your team's GitHub repository:
https://classroom.github.com/a/y5OIDHTI

Once your repository has been created, clone the repository using the following command:

```
git clone git@github.com:CS1217-Spring2023/CS1217-Assignment-3-[TEAMNAME]
```

(replace [TEAMNAME] with your team's name)

You can also find the link under the dropdown labelled **Code** on the top of the page. Make sure to use the link under the **SSH** tab.

Important note: Make sure that this is cloned in a new folder.

Assignment Submission Instructions

There are **2 places** where you'll have to submit. The **code** needs to be **submitted via Github Classroom**. A **PDF file** needs to be **submitted via Google Classroom**.

1) **Github Classroom**: For submitting code, make sure that your final submission code is on the **master** branch of the repository. We will only evaluate the *most recent commit at the time of the deadline* on the **master** branch.

2) **Google Classroom**: Submit a PDF file providing an explanation for your solutions. If you have solved any challenge problems, please enumerate the ones that you have solved, as well as brief explanations for your solutions. Please submit the PDF file on Google Classroom, named **<Team_Member_1_Name>_CS1217_Assignment3.pdf.** Any instructions that might be required for us to compile and run your code should be included in the PDF.

Boot xv6

QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC).q

(source: https://wiki.qemu.org/Main_Page)

In this part, we are building the xv6 OS with a QEMU emulator and attaching the kernel with the gdb. (Instructions on how to quit `qemu` and `gdb` are towards the end of this document.)

`cd` into the directory that has the xv6 and build it:

```
$ cd CS1217-Assignment-3-[TEAMNAME]
$ make
```

```
...
gcc -O -nostdinc -I. -c bootmain.c
gcc -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o
bootmain.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
...
$
```

Finding and breaking at an address

Find the address of `_start` by executing below **command on terminal**, the entry point of the kernel:

```
$ nm kernel | grep _start
8010a48c D _binary_entryother_start
8010a460 D _binary_initcode_start
0010000c T _start
```

In this case, the address is **`0010000c`**.

1. To run the kernel inside QEMU GDB,
2. setting a breakpoint at `_start` (i.e., the address you found using the previous command),

follow the instructions:

Run the following two commands in **two separate terminal windows**:

```
$ make qemu-nox-gdb
$ gdb
```

Note: Leave `make qemu-nox-gdb` running in one terminal, and in a new terminal (tab/window), navigate to the same directory and run the following. Make sure that you are running both the commands on the same machine, which in this case is your VM

```
$ gdb
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) br * 0x0010000c     // set the breakpoint here
Breakpoint 1 at 0x10000c
(gdb) c                  // Continue and stop at the breakpoint
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:      mov    %cr4,%eax

Breakpoint 1, 0x0010000c in ?? ()
(gdb)
```

The details of what you see are likely to differ from the above output, depending on the version of gdb you are using, but gdb should stop at the breakpoint, and it should be the above mov instruction.

Your gdb may also complain that auto-loading isn't enabled. In that case, it will print instructions on how to enable auto-loading, and you should follow those instructions.

If gdb still gives the `warning` **“The program is not being run”** execute the following inside gdb:

```
(gdb) file kernel
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:      mov    %cr4,%eax

Breakpoint 1, 0x0010000c in ?? ()
(gdb)
```

If you see a similar output as above inside `gdb`, then you have successfully attached the xv6 kernel executable with gdb, and you can debug this running program.

<u>Note:</u> If it didn't work still, execute `make clean` in the same directory and follow the instructions again beginning from `make qemu-nox-gdb`

The following exercises ask you to look inside the running program using gdb and answer some questions.

Exercise 0: What is on the stack?

While stopped at the above breakpoint, look at the registers and the stack contents:

```
(gdb) info reg
...
(gdb) x/24x $esp
...
(gdb)
```

Problem: Write a short (3-5 word) comment next to each non-zero value on the stack explaining what it is. Which part of the stack printout is actually the stack? (Hint: not all of it.)

You might find it convenient to consult the files `bootasm.S`, `bootmain.c`, and `bootblock.asm` (which contains the output of the compiler/assembler). The reference page of the MIT course has pointers to x86 assembly documentation, if you are wondering about the semantics of a particular instruction. Your goal is to understand and explain the contents of the stack that you saw above, just after entering the xv6 kernel. One way to achieve this would be to observe how and where the stack gets setup during early boot and then track the changes to the stack up until the point you are interested in. Here are some questions to help you along:

- Begin by restarting `qemu` and `gdb`, and set a break-point at `0x7c00`, the start of the boot block (`bootasm.S`). Single step through the instructions (type `si` at the gdb prompt). Where in `bootasm.S` is the stack pointer initialized? (Single step until you see an instruction that moves a value into `%esp`, the register for the stack pointer.)
- Single step through the call to `bootmain`; what is on the stack now?
- What do the first assembly instructions of bootmain do to the stack? Look for `bootmain` in `bootblock.asm`.
- Continue tracing via `gdb` (using breakpoints if necessary -- see hint below) and look for the call that changes eip to `0x10000c`. What does that call do to the stack? (Hint: Think about what this call is trying to accomplish in the boot sequence and try to identify this point in `bootmain.c`, and the corresponding instruction in the bootmain code in bootblock.asm. This might help you set suitable breakpoints to speed things up.)

Exercise 1 : System call tracing

Problem: Your first task is to modify the xv6 kernel to print out a line for each system call invocation. It is enough to print the name of the system call and the return value; you don't need to print the system call arguments.

When you're done, you should see output like this when booting xv6:

```
...
fork -> 2
exec -> 0
open -> 3
close -> 0
$write -> 1
write -> 1
```

The above example is `init` forking and execing `sh`, `sh` making sure only two file descriptors are open, and `sh` writing the `$` prompt.

Hint: modify the `syscall()` function in `syscall.c`.

Challenge Problem: Print the system call arguments.

Exercise 2 : Date system call

Problem: Your second task is to add a new system call to xv6. The main point of the exercise is for you to see some of the different pieces of the system call machinery. Your new system call will get the current UTC time and return it to the user program. You may want to use the helper function, `cmostime()` (defined in `lapic.c`), to read the real time clock. `date.h` contains the definition of `struct rtcdate`, which you will provide as an argument to `cmostime()` as a pointer. You should also create a user-level program that calls your new date system call; here's some source you should put in `date.c`:

```
#include "types.h"
#include "user.h"
#include "date.h"

int
main(int argc, char *argv[])
{
  struct rtcdate r;

  if (date(&r)) {
    printf(2, "date failed\n");
    exit();
  }

  // your code to print the time in any format you like...

  exit();
}
```

In order to make your new date program available to run from the xv6 shell, add `_date` to the `UPROGS` definition in Makefile.

Your strategy for making a date system call should be to clone all of the pieces of code that are specific to some existing system call like `uptime`. Grep for `uptime` in all the source files, using `grep -n uptime *.[chS]`.

When you're done, typing `date` to an xv6 shell prompt should print the current UTC time.

Write down a few words of explanation for each of the files you had to modify in the process of creating your `date` system call.

Challenge Problem: Add a `dup2()` system call and modify the shell to use it. A description of the `dup()` and `dup2()` system calls can be found here.

FAQs

How do I quit these two terminal windows?

Ans: 1. In the terminal with `gdb` you can type `quit` and press `y` on the follow up.
2. In the terminal with `qemu`, where you executed make `qemu-nox-gdb`, use key combination Ctrl + A and then X, which means:

1. first press Ctrl + A,
2. then release the keys,
3. afterwards press X.