Home

HW5: System Calls

This homework asks you to extend the xv6 kernel with several simple system calls.

You will program the xv6 operating system. To start working on this homework follow the xv6 setup instructions. After you're done with them, you'll be ready to start working on the assignment.

Exercise 1: Running GDB

This first part of the assignment teaches you to debug the xv6 kernel with GDB. First, lets start GDB and set a breakpoint on the `main` function.

From inside your `xv6-public` folder, launch QEMU with a GDB server:

```
CADE$ make qemu-nox-gdb
...
```

Now open another terminal (**you do that on your CADE host machine, i.e., circinus-XX, odin, or tristram, whichever you're using**). In this new terminal change to the folder where you've built xv6, and start GDB:

```
<<<<<<< HEAD
openlab$ cd ~/CS5460/xv6-public
openlab$ gdb
||||||| 38149bf
openlab$ cd ~/cs143a/xv6-public
openlab$ gdb
=======
CADE$ cd ~/cs5460/xv6-public
CADE$ gdb
>>>>>>> 19b7896276667be5bb882db8a655760faa4fb760
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
```

What you see on the screen is the assembly code of the BIOS that QEMU executes as part of the platform initialization. The BIOS starts at address `0xfff0` (you can read more about it in the How Does an Intel Processor Boot? blog post. You can single step through the BIOS machine code with the `si` (single instruction) GDB command if you like, but it's hard to make sense of what is going on so lets skip it for now and get to the point when QEMU starts executing the xv6 kernel.

Note, if you need to exit GDB you can press **Ctrl-C** and then **Ctrl-D**. To exit xv6 running under QEMU you can terminate it with **Ctrl-A X**.

Now from inside the GDB session set a breakpoint on `main`, e.g.

```
 (gdb) br main
Breakpoint 1 at 0x80102e00: file main.c, line 19.
```

Now since you set two breakpoints you can continue execution of the system until one of them gets hit. In gdb enter the "c" (continue) command to run xv6 until it hits the first breakpoint (`main`).

```
(gdb) c
```

If you need help with GDB commands, GDB can show you a list of all commands with

```
(gdb) help all
```

Now you've reached the C code, and since we compiled it with the "-g" flag that includes the symbol information into the ELF file we can see the C source code that we're executing. Enter the `l` (list) command.

```
(gdb) l
14      // Bootstrap processor starts running C code here.
15      // Allocate a real stack and switch to it, first
16      // doing some setup required for memory allocator to work.
17      int
18      main(void)
19      {
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
21        kvmalloc();      // kernel page table
22        mpinit();        // detect other processors
23        lapicinit();     // interrupt controller
```

Remember that when you hit the `main` breakpoint the GDB showed you that you're at line 19 in the main.c file (`main.c:19`). This is where you are. You can either step into the functions with the `s` (step) command (note, in contrast to the `si` step instruction command, this one will execute **one C line at a time**), or step over the functions with the `n` (next) command which will not enter the function, but instead will execute it till completion.

Try stepping into the `kinit1` function.

```
(gdb) s
```

Note, that on my machine when I enter `s` for the first time the GDB believes that I'm executing the `startothers()` function. It's a glitch---the compiler generated an incorrect debug symbol information and GDB is confused. If I hit `s` a couple of times I eventually get to `kinit1()`.

The whole listing of the source code seems a bit inconvenient (entering `l` every time you want to see the source line is a bit annoying). GDB provides a more conventional way of following the program execution with the TUI mechanism. Enable it with the following GDB command

```
(gdb) tui enable
```

```
(gdb) layout asm
```

Now you see the source code window and the machine instructions at the bottom. You can use the same commands to walk through your program. You can scroll the source with arrow keys, PgUp, and PgDown.

TUI can show you the state of the registers and how they are changing as you execute your code

```
(gdb) tui reg general
```

TUI is a very cute part of GDB and hence it makes sense to read more about various capabilities http://sourceware.org/gdb/onlinedocs/gdb/TUI-Commands.html. For example, you can specify the assembly layout to single step through machine instructions similar to source code:

```
(gdb) layout asm
```

For example, you can switch to the asm layout right after hitting the `_start` breakpoint. Similar, you can read the source code of the program with

```
(gdb) layout src
```

You can also have the GDB to show both the assembly and the source code at the same time with

```
        (gdb) layout split
```

Make yourself familiar with GDB. Try stepping through the code, setting breakpoints, printing out variables. Here is your GDB cheat sheet.

Troubleshooting GDB

```
circinus-1:1001-/16:40>gdb
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-110.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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-redhat-linux-gnu".
For bug reporting instructions, please see:
.
warning: File "/home/aburtsev/projects/cs5460/xv6-public/.gdbinit" auto-loading has been
declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load:/usr/bin/mono-gdb.py".
To enable execution of this file add
        add-auto-load-safe-path /home/aburtsev/projects/cs5460/xv6-public/.gdbinit
line to your configuration file "/home/aburtsev/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/aburtsev/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) quit
```

Add the line

```
add-auto-load-safe-path /home/aburtsev/projects/cs5460/xv6-public/.gdbinit
```

to

```
/home/aburtsev/.gdbinit
```

but of course replace "aburtsev" with your user name.

Exercise 2: Breaking inside the bootloader

This exercise asks you to break at the address early in the boot chain of the kernel, i.e., the bootloader and the entry point of the kernel.

Remember that the BIOS loads the kernel bootloader at the address `0x7c00`. The kernel bootloader is implemented in the file `bootasm.S`

```
# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

Lets try to set the breakpoint at this address. First, exit both GDB and QEMU sessions and start them over. Inside GDB set the breakpoint at a specific address `0x7c00`

```

(gdb) br * 0x7c00
Breakpoint 1 at 0x7c00
```

Now enter `c` to continue

```
(gdb) c
Continuing
[   0:7c00] => 0x7c00:  cli
```

When the breakpoint is hit you'll see familiar assembly code of the `bootasm.S`, i.e., the `cli` instruction that disables the interrupts.

Now use the `si` (step instruction) command to single step your execution (execute it one machine instruction at a time). Remember that the `0x7c00` address is defined in the assembly file, `bootasm.S` (the entry point of the boot loader). Enter `si`

```
(gdb) si
```

Every time you enter `si` it executes one machine instruction and shows you the next machine instruction so you know what is coming next

```
(gdb) si
(gdb) si
[   0:7c01] => 0x7c01:  xor    %ax,%ax
```

Note, you don't have to enter `si` every time, if you just press "enter" the GDB will execute the last command.

You can switch between ATT and Intel disassembly syntax with these commands:

```
(gdb) set disassembly-flavor intel
(gdb) set disassembly-flavor att
```

If you hit `si` a couple of times you eventually reach the C code of the `bootmain()` function that implements loading of the kernel from disk (it's implemented in the `bootmain.c` file). Note that GDB doesn't pick up the source file information since it's using the debugging info for the kernel, not for the bootblock which are two different programs. You can however follow what is going on by comparing the instructions that you execute with the `si` GDB command against the `bootblock.asm` file --- a helper file generated by the compiler that interleaves assembly instructions with the source.

Note the first `call` instruction. Right before it, the first stack was set up (this will be useful in the question about the stack below).

Part 1 (40%): Memtop system call

Now you're ready to start on the main part of the homework in which you will add a new system call to the xv6 kernel. 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 print the stats about available and used system memory.

Specifically, your new system call will have the following interface:

```
 int memtop();
```

It takes no arguments and returns the amount of memory available in the system. When you invoke it from your test program `mtop` you should print the number in bytes on the console:

```
$ mtop
available memory: 29712384
```

In order to test your system call you should create a user-level program `mtop` that calls your new system call. In order to make your new `mtop` program available to run from the xv6 shell, look at how other programs are implemented, e.g., `ls` and `wc` and make appropriate modifications to the Makefile such that the new application gets compiled, linked, and added to the xv6 filesystem.

When you're done, you should be able to invoke your `mtop` program from the shell. You can follow the following example template for `bt.c`, but feel free to extend it in any way you like:

```
#include "types.h"
#include "stat.h"
```

```
#include "user.h"

int
main(int argc, char *argv[])
{
  /* Syscall invocation here */

  exit();
}
```

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

Your strategy for making the `memtop` system call should be to clone all of the pieces of code that are specific to some existing system call, for example the "uptime" system call or "read". You should grep for uptime in all the source files, using `grep -n uptime *.[chS]`.

Some hints

To count up the available system memory, you should walk the linked list used by the memory allocator and count how many pages are still available on that list.

Submit your work

Submit your solution through <<<<<<< HEAD Gradescope Gradescope CS5460 Operating Systems as a compressed tar file of your xv6 ||||||| 38149bf Gradescope Gradescope CS143A Operating Systems as a compressed tar file of your xv6 ======= Gradescope Gradescope CS5460 Operating Systems as a compressed tar file of your xv6 >>>>>>> 19b7896276667be5bb882db8a655760faa4fb760 source tree (after running make clean). You can use the following command to create a compressed tar file.

```
CADE$ cd xv6-public
CADE$ make clean
CADE$ cd ..
CADE$ zip -r hw5.zip xv6-public
```

Updated: April, 2023