

Homework 1

[total 12 marks]

Submission

Homework is to be submitted using Gradescope.

Assignment : Get familiar with xv6

[4 marks]

The first part of the homework is to get you started with an instructional OS, xv6, and to get you set-up and running with it, so you can test it for yourself.

Fetch the xv6 source. See [references](#) for instructions on obtaining the source code. Build xv6:

```
$ cd xv6
$ make
gcc -O -nostdinc -I. -c bootmain.c
gcc -nostdinc -I. -c bootasm.S
ld -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary bootblock.o bootblock
...
$
```

Installing QEMU

Qemu is installed on the department cluster. To install qemu on your machine (Linux)

```
$ sudo apt-get install qemu qemu-system-g++-multilib git-all grub2 grub-pc-bin libboost-all-dev xorriso
```

Running xv6 under QEMU

```
$ make qemu
qemu -parallel mon:stdio -smp 2 -hdb fs.img xv6.img
QEMU 0.10.6 monitor - type 'help' for more information
(qemu) QEMU 0.10.6 monitor - type 'help' for more information
(qemu)
```

A separate window should appear containing the display of the virtual machine. After a few seconds, QEMU's virtual BIOS will load xv6's boot loader from a virtual hard drive image contained in the file xv6.img, and the boot loader will in turn load and run the xv6 kernel. After everything is loaded, you should get a '\$' prompt in the xv6 display window and be able to enter commands into the rudimentary but functional xv6 shell. For example, try:

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 1844
cat        2 3 12129
...
$ echo Hello!
Hello!
$ cat README
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
...
$ grep run README
To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
Then run "make TOOLPREFIX=i386-jos-elf-".
```

```
...
$ cat README | grep run | wc
6 70 376
$ echo My New File >newfile
$ cat newfile
My New File
```

The small file system you're examining and modifying here resides on a second virtual disk, whose initial contents QEMU initializes from the file `fs.img`. Later in the course you will examine how xv6 accesses and modifies this file system.

Now close this QEMU session, destroying the state of the xv6 virtual machine. You can do so either by entering `quit` at the `(qemu)` prompt in the original window from which you started QEMU, or just by pressing CTRL-C in that window.

Remote Debugging xv6 under QEMU

The easiest way to debug xv6 under QEMU is to use GDB's *remote debugging* feature and QEMU's remote GDB debugging stub.

Remote debugging is a very important technique for kernel development in general: the basic idea is that the main debugger (GDB in this case) runs separately from the program being debugged (the xv6 kernel atop QEMU) - they could be on completely separate machines, in fact. The debugger and the target environment communicate over some simple communication medium, such as a network socket or a serial cable, and a small *remote debugging stub* handles the "immediate supervision" of the program being debugged in the target environment. This way, the main debugger can be a large, full-featured program running in a convenient environment for the developer atop a stable existing operating system, even if the kernel to be debugged is running directly on the bare hardware of some other physical machine and may not be capable of running a full-featured debugger itself. In this case, a small remote debugging stub is typically embedded into the kernel being debugged; the remote debugging stub implements a simple command language that the main debugger uses to inspect and modify the target program's memory, set breakpoints, start and stop execution, etc. Compared with the size of the main debugger, the remote debugging stub is typically miniscule, since it doesn't need to understand any details of the program being debugged such as high-level language source files, line numbers, or C types, variables, and expressions: it merely executes very low-level operations on behalf of the much smarter main debugger.

When we are doing kernel development using a virtual machine such as QEMU, remote debugging may not be quite as critical: for example, xv6 can also be run under the [Bochs](#) emulator, which is much slower than QEMU but has a debugger built-in and thus does not require the use of GDB remote debugging. On the other hand, while usable, the Bochs debugger is still not as complete as GDB, so we will primarily use GDB with QEMU's remote debugging stub in this course.

To run xv6 under QEMU and enable remote debugging, type:

```
$ make qemu-gdb
*** Now run 'gdb'.
qemu -parallel mon:stdio -smp 2 -hdb fs.img xv6.img -s -S -p 26000
QEMU 0.10.6 monitor - type 'help' for more information
(qemu)
```

You will notice that while a window appears representing the virtual machine's display, nothing appears on that display: that is because QEMU initialized the virtual machine but stopped it before executing the first instruction, and is now waiting for an instance of GDB to connect to its remote debugging stub and supervise the virtual machine's execution. In particular, QEMU is listening for connections on a TCP network socket, at port 26000 (we have configured the makefile to try and ensure that it listens to a different port for each different user) in this example, because of the `-p 26000` in the `qemu` command line above. You might see a different port number: this is because xv6's Makefile tries to compute a port number that is likely to be unique and available even if multiple users are debugging xv6 under QEMU on the same machine (see the `GDBPORT` variable in xv6's Makefile).

To start the debugger and connect it to QEMU's waiting remote debugging stub, open a new, separate terminal window, change to the same xv6 directory, and type:

```
$ gdb kernel
GNU gdb (GDB) 6.8
Copyright (C) 2009 Free Software Foundation, Inc.
...
Reading symbols from /Users/ford/cs422/xv6/kernel...done.
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    jmp    $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb)
```

Several things are going on here. Note that we entered 'gdb kernel' just as if we were going to debug a program named kernel directly under this GDB instance - but actually trying to execute the xv6 kernel under GDB in this way wouldn't work at all, because GDB would provide an execution environment corresponding to a user-mode Linux process (or a process on whatever operating system you are running GDB on), whereas the kernel expects to be running in privileged mode on a "raw" x86 hardware environment. But even though we're not going to run the kernel locally under GDB, we still need to have GDB load the kernel's ELF program image so that it can extract the debugging information it will need, such as the addresses of C functions and other symbols in the kernel, and the correspondence between line numbers in xv6's C source code and the memory locations in the kernel image at which the corresponding compiled assembly language code resides. That is what GDB is doing when it reports "Reading symbols from ...".

Important: When remote debugging, *always* make sure that the program image you give to GDB is exactly the same as the program image running on the debugging target: if they get out of sync for any reason (e.g., because you changed and recompiled the kernel and restarted QEMU without also restarting GDB with the new image), then symbol addresses, line numbers, and other information GDB gives you will not make any sense. Fortunately keeping the target program and the debugger in sync is not too difficult when they are both loaded from the same directory in the same host machine as they are in this case, but synchronization can be a bit more of a challenge with "true" remote debugging, where one machine runs GDB and another machine runs the target kernel loaded from a separate media such as a local hard disk or USB stick.

The GDB command 'target remote' connects to a remote debugging stub, given the waiting stub's TCP host name and port number. In our case, the xv6 directory contains a small GDB script residing in the file .gdbinit, which gets run by GDB automatically when it starts from this directory. This script automatically tries to connect to the remote debugging stub on the same machine (localhost) using the appropriate port number: hence the "+ target remote localhost:26000" line output by GDB. If something goes wrong with the xv6 Makefile's port number selection (e.g., it accidentally picks a port number already in use by some other process on the machine), or if you wish to run GDB on a different machine from QEMU (try it!), you can comment out the 'target remote' command in .gdbinit and enter the appropriate command manually once GDB starts.

Once GDB has connected successfully to QEMU's remote debugging stub, it retrieves and displays information about where the remote program has stopped:

```
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    jmp    $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

As mentioned earlier, QEMU's remote debugging stub stops the virtual machine before it executes the first instruction: i.e., at the very first instruction a real x86 PC would start executing after a power on or reset, even before any BIOS code has started executing. For backward compatibility, PCs today still start executing after reset in exactly the same way the very first 8086 processors did: namely in 16-bit, "real mode", starting at address 0xffff0 - 16 bytes short of the end of the BIOS and the top of the 1MB of total addressable memory in the original PC architecture.

We'll leave further exploration of the boot process for later; for now just type in the GDB window:

```
(gdb) b exec
Breakpoint 1 at 0x100800: file exec.c, line 11.
(gdb) c
```

These commands set a breakpoint at the entrypoint to the exec function in the xv6 kernel, and then continue the virtual machine's execution until it hits that breakpoint. You should now see QEMU's BIOS go through its startup process, after which GDB will stop again with output like this:

```
The target architecture is assumed to be i386
0x100800 :      push    %ebp

Breakpoint 1, exec (path=0x20b01c "/init", argv=0x20cf14) at exec.c:11
11      {
(gdb)
```

At this point, the machine is running in 32-bit mode, the xv6 kernel has initialized itself, and it is just about to load and execute its first user-mode process, the /init program. You will learn more about exec and the init program later; for now, just continue execution:

```
(gdb) c
Continuing.
0x100800 :      push    %ebp

Breakpoint 1, exec (path=0x2056c8 "sh", argv=0x207f14) at exec.c:11
11      {
(gdb)
```

The second time the exec function gets called is when the /init program launches the first interactive shell, sh.

Now if you continue again, you should see GDB appear to "hang": this is because xv6 is waiting for a command (you should see a '\$' prompt in the virtual machine's display), and it won't hit the exec function again until you enter a command and the shell tries to run it. Do so by typing something like:

```
$ cat README
```

You should now see in the GDB window:

```
0x100800 :      push    %ebp

Breakpoint 1, exec (path=0x1f40e0 "cat", argv=0x201f14) at exec.c:11
11      {
(gdb)
```

GDB has now trapped the exec system call the shell invoked to execute the requested command.

Now let's inspect the state of the kernel a bit at the point of this exec command.

Turn in: the output of the following GDB 'print' or 'p' commands, with which we can inspect the arguments that the exec function was called with:

```
(gdb) p argv[0]
(gdb) p argv[1]
(gdb) p argv[2]
```

Turn in: the output of the GDB 'backtrace' or 'bt' command at this point, which traces and lists the chain of function calls that led to the current function: i.e., the function that made this call to exec, the function that called that function, etc.

Now go "up" one call frame so we can inspect the context from which exec was called:

```
(gdb) up
#1  0x00103e96 in sys_exec () at sysfile.c:367
367      return exec(path, argv);
(gdb)
```

Turn in: the output of the GDB 'list' or 'l' command at this point, showing the source code around `sys_exec`'s call to `exec`.

Further exploration: Look through the online [GDB manual](#) to learn about more of GDB's debugging features, and try them out on a running instance of the xv6 kernel.

Note that you don't have to settle for the "vanilla" command-line GDB; you should be able to use one of the many more "user-friendly" variants, such as the GNU Emacs graphical interface to GDB, or the graphical [DDD](#) frontend. But you may have to figure out for yourself how to connect with QEMU's remote debugging stub under your preferred GDB variant or front-end.

Assignment : File descriptors

[2 marks]

See Page 12 (Section Code: File Descriptors) of the xv6 book. The last sentence of the second-last para states that the "xv6 shell doesn't support I/O redirection for the error file descriptor, but now you can implement it."

Implement redirection to the error file descriptor. Reading the chapter should help you (hint: use `dup()` call).

Turn in: Pseudo-code which implements I/O redirection.

Assignment : System calls and processes

[1 mark]

On a linux machine, type the following command

```
$ cat | tee output.file
```

`cat` is a UNIX utility that prints the contents of STDIN to STDOUT. `tee` is a UNIX utility that prints the contents of STDIN to both STDOUT and to the file named by its argument (`output.file`). After you type this command, you can type in some characters followed by the newline character.

Turn in: While this command is running, examine the processes created:

1. Use `ps tree` to see the process hierarchy. Tell us what you find about the process hierarchy.
2. Use `'ps x'` to identify the process IDs of the processes created by `cat` and `tee` commands. Linux provides a `proc` pseudo-filesystem which can be used to examine the state of a process using filesystem namespace.
 - Type the following command for both process-ids:

```
$ ls -l /proc/pid-num/fd/
```

What do you find? What are 0,1,2,...? What do they symlink to?

- Type

```
$ ls -l /proc/self/fd/
```

What do you find? What is 3 pointing to? Why?

3. Use `strace` to see the system calls executed by a given command. For example, type `strace cat foo.txt` (create `foo.txt` before-hand) and see the system calls that were executed, printed to STDOUT. List all the unique system calls that were used by this command.

Assignment : Signal handlers

[1 mark]

- Consider the following C program that executes an infinite loop:

```
#include <signal.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    while (1);
}
```

1. Compile this C program, and run it. Then press Ctrl-C on the shell. What happens?
2. Now modify this C program to add a signal handler for SIGINT (read: `man signal`), such that the string "hello" gets printed each time Ctrl-C is pressed. Run the program, and press Ctrl-C on the shell. What happens? Now try using the kill command on another terminal, by typing `kill -2 pid` (*pid* is the process ID of the process that is running this program; you can find the process-id through the `ps` command). What happens and why? How can you stop this process now? (hint: try using a different signal number).
3. Turn-in the code of the new program with the signal handler for SIGINT, and the kill commands you used to stop the process.

Assignment : Threads

[4 marks]

Threads can be implemented completely at the user level. i.e., we do not require privileged operations to implement a thread abstraction and schedule different threads. In other words, a process can provide multiple threads by implementing a scheduler. Let's see how this can be done.

To implement threads, the process needs to provide the abstraction of multiple control-flow (program counter), multiple register sets and multiple stacks. This can be done if after every periodic time interval, one thread can be interrupted and saved and another thread can be loaded. Saving a thread involves saving its program counter, registers and stack pointer. Similarly, loading a thread involves loading the new thread's program counter, registers and stack pointer. Neither the save operation, nor the load operation requires any privileged operation -- we are just loading and saving registers.

So the only remaining issue is how to periodically interrupt a running thread from within a process. For an OS, this interruption is done by the hardware timer device. A process can do this using the SIGALRM signal.

Such threads implemented inside a process are called user-level threads. The OS cannot distinguish between multiple user-level threads and it can only see one process that is running. The process, internally, includes the thread scheduler and the different threads.

Turn in:

Read the manpage of the `signal`, `alarm`, and `setitimer`. Understand how SIGALRM can help in implementing user-level threads. Briefly describe how you will do this (2-3 sentences and some pseudo-code).