The xv6 Survival Guide
Introduction to Operating Systems

Revision: March 28, 2021

Contents

1 Introduction **1**
- 1.1 Administrivia . . . 1
- 1.2 Regrade Requests . . . 1
- 1.3 D2L Usage and Deadlines . . . 2
- 1.4 Access and Inclusion for Students with Disabilities . . . 2
- 1.5 Extraordinary Circumstances . . . 2
- 1.6 Programming Languages . . . 2
  - 1.6.1 Coding Style . . . 3
  - 1.6.2 The C Preprocessor . . . 3
  - 1.6.3 Makefile Considerations . . . 3
- 1.7 Prerequisites . . . 4
  - 1.7.1 Processes . . . 4
  - 1.7.2 System Calls and Low-Level I/O . . . 4
  - 1.7.3 Concurrency . . . 4
  - 1.7.4 Algorithms and Data Structures . . . 4
- 1.8 Resources . . . 5
- 1.9 The Project Test Report . . . 5
  - 1.9.1 LaTeX . . . 5
  - 1.9.2 Project Test Report Structure . . . 6

2 Preparing Your Code Submission **7**
- 2.1 `runoff.list` . . . 7
- 2.2 Creating and Testing Your Submission . . . 7

3 About xv6 **9**
- 3.1 Getting Started . . . 9
  - 3.1.1 Download and Unpack xv6 . . . 9
  - 3.1.2 Building the Kernel . . . 10
  - 3.1.3 Running the Kernel . . . 10
- 3.2 QEMU . . . 13
- 3.3 Running xv6 under QEMU . . . 13
- 3.4 Debugging . . . 14
  - 3.4.1 Alternate Symbol Table . . . 14
  - 3.4.2 Debugging a Crashed Kernel . . . 14
  - 3.4.3 Tracing Instruction Pointers . . . 15
  - 3.4.4 Inspecting Key Data Structures . . . 16
- 3.5 The `argint()` and `argptr()` functions . . . 17

3.6 XV6 C Library 19

Chapter 1

Introduction

Operating systems is a very large topic. In this course, you will learn about core concepts and implement some of those concepts. The purpose of this document is to give you guidance and advice that course staff feel will be beneficial. This document is **required reading**.

This document is *not* a replacement for attending lectures nor is it a crutch that will allow you to avoid assigned readings. This document assumes that you are reading assigned materials before the subject matter lecture, attend and participate in lectures, and keep current on material discussed on the class slack channel.

1.1 Administrivia

This section of CS333 will, in addition to lectures, include a series of projects that will add interesting features to xv6. The lectures will explain concepts and strategies used in operating systems while the projects will reinforce these concepts and give you hands-on experience with implementing some strategies in a small operating system called xv6[4].

The main course text, OSTEP[2], is a free e-textbook; the main reference for xv6 is similarly free[4]. Other resources may be helpful as we will use the C programming language[13][14]. Familiarity with Intel assembly, as taught in CS 201, is assumed. You may have a copy of "K&R"[14], but this is not required.

1.2 Regrade Requests

Any regrade request must be received by the instructor via email within two weeks of the assignment grade and feedback being made available. Requests must be specific as to why the score or feedback is incorrect or unclear. A response will be provided in email. The email chain will be archived.

A regrade request for exams require that the student schedule a meeting with the instructor. The instructor and student will review the exam together and assess the exam for possible additional points.

Note: Any request for regrade may result in the reevaluation of the entire work. This reevaluation may result in additional points being awarded, no change in the points awarded, or in additional points being deducted from the work. In rare circumstances, additional student work may be reevaluated.

1.3 D2L Usage and Deadlines

You will use D2L to submit your projects. We will also provide project feedback and grading on D2L. One of the reasons that we are using D2L is that the assignment submission folders on D2L can be set to reject any submissions beyond a certain date and time. Late assignments will be penalized 10% of the total points per hour late. No assignments will be accepted that are over two hours late. Check the D2L calendar for specific due dates.

1.4 Access and Inclusion for Students with Disabilities

Accommodations are collaborative efforts between students, faculty, and the Disability Resource Center. Students with accommodations approved through the DRC are responsible for contacting the faculty member in charge of the course prior to or during the first week of the term to discuss accommodations. Students who believe they are eligible for accommodations but who have not yet obtained approval through the DRC should contact the DRC immediately. The DRC may be reached at 503-725-4150 or visit the DRC at http://www.pdx.edu/drc.

1.5 Extraordinary Circumstances

If an extraordinary situation (for example severe illness) prevents you from working for a period of time, contact us as soon as possible to discuss your situation and arrange a special schedule. Scheduled work commitments do not constitute an extraordinary circumstance. Makeup exams will not be given except in cases of severe and documented medical or family emergencies. Please note that travel is not considered an emergency. If an emergency arises and you miss an exam, contact the instructor before the exam to arrange for a special circumstance. Medical or similar documented emergencies that fall within University guidelines for accommodation will receive accommodation. Know the University rules before you ask as the burden of meeting the guidelines falls upon you, the person requesting accommodation.

Any student may take an exam in the testing center. You are required to make your own appointment. Unless otherwise given written aproval from the instructor, the start time for an exam taken at the testing center must overlap with the time when the exam is given in class. Students are required to follow all rules and regulations of the testing center; see the web site.

1.6 Programming Languages

Students will be exposed to assembly language code written in GNU assembly for the Intel IA-32 architecture (x86) and code in the C programming language for both kernel and user level programs.

All programming projects will be in ANSI C and we will use the GNU C compiler[8]. The xv6 kernel does contain 32-bit assembly code for the Intel x86 architecture, which you are expected to understand. We will not be using advanced features of the Intel instruction set. The 1987 version of the manual is relatively small, easy to understand, and will suffice for our purposes[12].

Students should be aware that the C library provided with xv6 (ulib.c, printf.c, and umalloc.c) is a highly restricted subset of the standard C library. There is no written documentation for this library but students can easily learn the details by reading the source, which is quite small. The course instructor must approve any student modification to the xv6 C library.

1.6.1 Coding Style

In any large project, matching the existing coding style is important. Having different coding styles intermixed leads to confusion and bugs. Students are **required** to follow the existing coding style in xv6 – this includes maintaining the indentation style in .c and .h files using spaces, not tabs. In particular, pay close attention to function declarations and how the *function name* begins the *line after* the function return type. For *helper functions* which are limited in scope to a specific file, you **must** declare the function as "static" in the same file in which it is used. If this is unclear, see §4.6 of [14].

On using goto: Course staff cannot stress enough how important it is to *avoid using the C goto instruction*. There are times when it is appropriate, especially for code clarity and handling awkward error states, but this is rare. The original xv6 code uses the `goto` instruction in certain places for code clarity in order to assist your learning. Do not add any additional goto statements. Using a *goto* is considered a style violation and will be penalized with a deduction. **Do not use goto.**

The indentation style for xv6 is 2 spaces. Many students are taught to use tabs for indentation, which can make code very hard to read, especially when there are several levels of indentation. For programmers who use the vi or vim editors, you can place the following lines into your `.vimrc` file to set tab stops to the class indentation style and then automatically convert the tabs to spaces when you exit the editor.

```
:set smartindent
set expandtab autoindent shiftwidth=2 tabstop=2
```

1.6.2 The C Preprocessor

You have already seen the C preprocessor in action if only for the simple case of a `#include` statement. Section 4.11 of [14] concerns the C preprocessor. We will use additional features of the preprocessor, especially the `#define` statement and `#ifdef ... #endif` for conditional compilation (§4.11.3). The GNU documentation for the C preprocessor[7] is excellent. The CS 201 text [3] has excellent information on the compilation process and the C preprocessor in chapter 7.

Conditional compilation will be used in this course for all project code.

1.6.3 Makefile Considerations

The Makefile for this class has the student set the project number as an integer at the top of the file. Unfortunately, if you leave in any trailing spaces, the make system misunderstands your intent. If you are a vim user, you can set up your `.vimrc` file to automatically remove trailing spaces. Here is an email provided by the course TA in a previous section:

```
A number of students have run into problems with trailing whitespace in Makefiles
causing conditional compilation to fail. The easiest way to fix this is to tell your
editor to remove trailing whitespace.

Big red warning: I'm not a regular vim user, so if you run into problems I won't be able to
troubleshoot your vim configuration and I'll send you to either the tutors or Stack Overflow
(where I found this solution)

For brave vim users, here's the fix. You need to add the following to your .vimrc file (it's
located in your home directory on the Linux systems):
```

```
fun! <SID>StripTrailingWhitespaces()
    let l = line(".")
    let c = col(".")
    %s/\s\+$//e
    call cursor(l, c)
endfun
autocmd FileType c,cpp,java,php,ruby,python autocmd
        \ BufWritePre <buffer> :call <SID>StripTrailingWhitespaces()
autocmd BufWritePre Makefile :call <SID>StripTrailingWhitespaces()
```

Make sure you enter this in your .vimrc exactly as is, indentation and all.

If you would prefer to use a project-specific `.vimrc` file instead, some good information is available here.

1.7 Prerequisites

CS201, Computer Systems Programming is a prerequisites for this course. You are assumed to have taken this course and are prepared to implement certain concepts in an applied setting within xv6. The current text for CS201 is "B&O"[3]. There should be a copy of the CS201 text in the library, if you do not have your own. The course staff encourage you to refer to this text throughout the term.

1.7.1 Processes

Chapter 3 of B&O[3] covers process structure and execution. This information will be quite important as we discuss operating system support for processes. In particular, sections 3.4, 3.6 and 3.7 are seen as very useful for students to review.

1.7.2 System Calls and Low-Level I/O

Chapter 8 of K&R, `The UNIX System Interface`, discusses system calls that provide low-level I/O capabilities to programs and also discusses memory allocation and management. Chapter 10 of B&O covers related topics. If you are not already familiar with this material, please look at it and ask questions as soon as possible. Tracing and implementing system calls will be important in xv6.

1.7.3 Concurrency

The CS201 text contains an excellent chapter on concurrent programming[1] that is not normally covered during CS201. Concurrency is a very important topic in operating systems. You are encouraged to read this chapter to assist you in understanding concurrency, especially threads. Concurrency will be a major recurring topic throughout the course.

1.7.4 Algorithms and Data Structures

Lower division courses such as CS163, Data Structures, CS202, Programming Systems, and CS201, Computer Systems Programming cover fundamental algorithms and data structures. This course will emphasize algorithms, data structures, and their correct implementation in a new setting; namely concurrent programming. Students are assumed to be familiar with simple data structures in the C programming language.

[1] See [3] Chapter 12.

1.8 Resources

The following resources are good to know about.

1. Study Groups. Students are strongly encouraged to form study groups. Be sure to go prepared to your study group so that you can participate fully and get the most out of this valuable resource. Remember to study concepts and ideas but do not share solutions.

2. D2L Discussions. You are **required** to monitor the discussion area of D2L for this course. Course staff monitor and contribute to this area. Project clarifications will be posted here. Students are responsible for all clarifications and additional information posted.

3. CS333 slack channel. Slack has become very useful for student collaboration and asking questions. The PSU slack channels are managed by the CS tutors. There will often be a slack channel for this course, at the discretion of the TAs.

4. CS Linux Lab. You will develop your projects using department Linux systems. The main server for our class is babbage.cs.pdx.edu. Students can also use any Linux system in the CAT-supported CS labs. The Linux lab systems have the required virtual machine and tool chain installed. All assignments will be graded on these machines as well. Your assignments must work correctly on these machines or they will be considered incorrect.

5. CS Tutors. The CS tutors can help you with some questions around the C programming language and the GNU debugger, gdb[10]. They cannot help you with programming projects.

6. Class TAs and Technical Course Support Specialists. Class personnel are selected on their ability to help guide you through the rigors of the course. TAs will hold regular office hours and TCSSs will conduct the labs.

1.9 The Project Test Report

A project report will be submitted with each assignment and is **required** to be in PDF format. Handwritten, simple text, MS Word, and similar reports **will not be accepted**. Systems such as MS Word and LaTeX are ideal for producing properly formatted PDFs. There are free packages that claim to produce documents compatible with MS Word and these include OpenOffice, FreeOffice, LibreOffice, and Google Docs. Your instructor is familiar with MS Word and LaTeX and recommends LaTeX for project reports.

1.9.1 LaTeX

LaTeX is a typesetting system that is very suitable for producing scientific and mathematical documents of high typographical quality. It is also suitable for producing all sorts of other documents, from simple letters to complete books. LaTeX uses TeX as its formatting engine[15].

You are encouraged to try out LaTeX and, to assist your efforts, the LaTeX source for this and other course documents will be provided[6]. The systems in the Linux lab have all the LaTeX tools installed but the instructor prefers to use TeXstudio on his Windows machine as it has a great GUI. The easiest path for the Windows user is to get ProTeXt from TUG (TeX Users Group). A very good reference for LaTeX is [15] and the wikibook is also good.

Students have recommended sharelatex.com. Sharelatex.com is a web application that is a resource for people who want to try LaTeX but perhaps don't want to commit or aren't sure how to install an

editor. It has all the basic functionality and several nice features, including vim and emacs bindings. Best of all, they have documentation that includes tutorials and guides on most things one would want to do.

1.9.2 Project Test Report Structure

The report has a formal structure. An example report template is provided for you under "Project 1 Example Report" on the course website. Seek clarification from the course staff when some issue is not clear.

You will document your tests and testing strategies to show that your code meets the project requirements (see each rubric). Do not include source code except in those (rare) cases where a code snippet would greatly improve understanding. Function prototypes and structure declarations are okay to include and not subject to this restriction. Every test must have a corresponding output demonstrating the test. You risk losing points if you fail to properly document your tests; "it is obvious" does not suffice. **Tests are required follow this format or points deduction will occur.**

1. Test Name
2. Test Description
3. Expected Results
4. Test Output / Actual Results
5. Discussion
6. Indication of PASS/FAIL

For "Test Output", you are **required to use screen shots**, not copy-paste. Screen shots with white backgrounds are easier for the graders to read; please set the background of your terminal window to white with black text. The project rubric will list all required tests.

Some projects will provide useful test programs that will let you test some of your code. You are free to use any provided test code to show that you meet the relevant requirements, but you must describe how each of the provided tests work and what requirements are specifically tested in order to get any credit for that test.

Note: It is possible that some feature of your project will not be fully functional. If so, you must include a test that shows that the feature fails. *You must have a specific test that shows the failure.* A project that does not compile cannot show tests for specific features. If this is unclear, ask.

Chapter 2

Preparing Your Code Submission

Project submissions will be compiled and graded using department Linux systems. It is the responsibility of each student to ensure that their submission compiles and runs correctly in that environment.

Students are **required** to follow the directions in this section for generating an archive file for submission. Failure to follow these directions may result in a (potentially major) points deduction.

2.1 `runoff.list`

Add any new files that you want to be submitted as part of your assignment to the file `runoff.list`. Here are example entries:

```
# CS333 additions for Mark Morrissey
date.c
ps.c
ps.h
time.c
```

Add new file names at the end of the file. Note that any time that you create a new file you will need to modify this file. See the next section regarding testing to ensure that you have the correct information in `runoff.list`.

2.2 Creating and Testing Your Submission

You are **required** to test your submission to ensure that it works. You test that your code works with the command `make dist-test`. This command will create the `dist` and `dist-test` subdirectories and then run the xv6 kernel for testing. If you find that files are missing then it is likely that you failed to update the `runoff.list` file. When you are satisfied with the results, create your submission with `make tar`. The tar file is a copy of your "dist" subdirectory.

The tar file created by the `make tar` command constitutes the code submission for each project. Students **may not** submit Apple ".rar" files, ".zip" files, etc. Project reports are to be submitted separately from the source code. This means that each projects submission will consist of two files.

Chapter 3

About xv6

Your project will require you to "evolve" the xv6 operating system[4]. Xv6 is designed to be small, compact, easy to understand and modify, and similar in structure to Linux and Unix systems. By the end of the course, not only will you have a bootable operating system of your own, you will also possess practical skills that will directly transfer to industry.

The xv6 source code provided for this course differs from the version released by MIT. Due to these differences, the line numbers in the xv6 book do not refer to the correct locations in your code. The original xv6 source code listing, matching the book, is available on the course web site.

3.1 Getting Started

3.1.1 Download and Unpack xv6

The source code for the xv6 kernel is packaged in a file named xv6-pdx.tar; click the link to download the file. From the Linux lab you can also copy the source with one of these commands:

```
cp ~markem/public_html/CS333/xv6/xv6-pdx.tar .
wget cs.pdx.edu/~markem/CS333/xv6/xv6-pdx.tar
```

To unpack the source code from the tar file, use this command:

```
tar xf xv6-pdx.tar
```

You will now have a directory named `xv6-pdx`. Change to this directory (`cd xv6-pdx`).

List the contents of this directory:

```
$ ls
asm.h  bio.c      bootasm.S  bootmain.c  buf.h   BUGS
cat.c  console.c  cuth       date.h      defs.h
...
```

As you can see, the kernel is comprised of many parts. There are also several "helper" files[1]. You will become very familiar with a small group of these files and leave many of them alone.

[1] Any file not ending in .c, .h, or .S is a helper file, for our purposes.

3.1.2 Building the Kernel

The project `Makefile` differs substantially from the original MIT version. See Project One for details on using this file. Be careful to follow the rules regarding conditional compilation.

Build the kernel using one of these commands:

```
make
make run
make debug
```

3.1.3 Running the Kernel

Since we are in a development environment using a simulated machine (via QEMU), the way that we run our code is a little bit more complex than just turning on a computer and watching the monitor. Fortunately, our method is also very flexible and does not require dedicated hardware.

Normal Boot

When you "boot" the kernel normally, you will use the command

```
make run
```

This will build the kernel then load and run xv6. You will know that your kernel has properly loaded and is executing the shell when you see output similar to

```
qemu-system-i386 -nographic -hdb fs.img xv6.img -smp 2 -m 512
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

The "$" is the shell prompt. The other lines are output when xv6 is initializing various parts of the kernel. We will not look at initialization in depth in this class.

At this point, you have a running operating system but one that can do very little. We will discuss the limitations and expand the capabilities through the course, but for now just type "ls" and you will see the entire list of shell commands available to you. The list is very small. The xv6 shell does not understand the concept of a search path.

Initializing with GDB

Remote access[2] will be via some SSL-based tool such as putty or slogin. You will need a minimum of two remote logins; one for running the kernel and one for the gdb debugger session[3].

[2] If you are logged in at a console in the Linux lab, just use two separate shell windows.

[3] If you use a tool such as `screen` or `tmux` then you can just use sub-windows inside that tool.

Once connected and in the correct directory, use the command
`make debug`

This will build the kernel, load it into the qemu simulator and wait for gdb to connect. Sample output looks like this[4]:

```
$ make debug
gcc -Werror -Wall -o mkfs mkfs.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -fvar-tracking -fvar-tracking-assignments
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -fvar-tracking -fvar-tracking-assignments
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o usys.o usys.S

  [. . . many lines of the compilation process deleted . . .]

dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.0171023 s, 29.9 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
353+1 records in
353+1 records out
181117 bytes (181 kB) copied, 0.00223519 s, 81.0 MB/s
sed "s/localhost:1234/localhost:27377/" < .gdbinit.tmpl > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -nographic -hdb fs.img xv6.img -smp 2 -m 512  -S -gdb tcp::27377
```

The last line states that the boot of the kernel is waiting for a gdb session to connect. You do this by typing "gdb" from a login session in the same directory, on the same machine, as you ran the make command. The first time, you will see an error message like this:

```
$ 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".
```

[4]Example output text may run off the right edge of the page. The output that is truncated is just an example. You will be able to see the entire lines of output when you run the commands for yourself.

```
warning: File "/u/markem/xv6-pdx/.gdbinit" auto-loading has been declined by your `auto-load s
To enable execution of this file add
        add-auto-load-safe-path /u/markem/xv6-pdx/.gdbinit
line to your configuration file "/u/markem/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/u/markem/.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)
```

It is a bit wordy, but the part to pay careful attention to is

```
  To enable execution of this file add
          add-auto-load-safe-path /u/markem/xv6-pdx/.gdbinit
  line to your configuration file "/u/markem/.gdbinit".
```

What this *really means* is to edit the file ".gdbinit" in your home directory (you may need to create it) and add the line that corresponds to the one in bold above to enable gdb to connect to your kernel. Note that the message will, of course, be slightly different and specific to you. Once you have added this command, you then restart gdb and you should see something like this:

```
$ 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:27377
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb)
```

Running the Kernel

Start QEMU running with the xv6 kernel in debug mode. You must have two sessions to the same machine and the current directory for both sessions must be the same. We will designate the one running xv6 the "command window" and the one running gdb the "debug window".

In the command window, enter the command

```
make debug
```

This will start QEMU and pause the loading of the kernel at the start of the boot code, waiting for gdb to connect.

Attaching gdb

In the debug window, launch gdb; you should see output similar to the following

```
$ gdb
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
(gdb)
```

You can now use `gdb` normally.

3.2 QEMU

You will boot and run your operating system using QEMU[16], a powerful emulator offering very good performance. You will not be expected to delve into the QEMU layer of the project. The project is structured so that QEMU "looks" like a native X86 processing environment. The principle advantage of QEMU is that dedicated hardware is not necessary. This environment will greatly speed your learning and also shorten development cycles.

3.3 Running xv6 under QEMU

You will run xv6 under the QEMU emulator by using a `make` command.

- `make run` will load and run xv6 normally.
- `make debug` will load and run xv6 in debug mode. See §3.4 for details.

To exit xv6, use one of these methods. Note that "control-a" means to hold down both the control key and the 'a' key at the same time.

1. The `halt` command from the xv6 command line.

2. control-a x. This will terminate the QEMU console and so terminate xv6.

3. control-a c. This will drop you into the QEMU console[5] where you can use "quit" to exit.

3.4 Debugging

At times, you will be running your operating system from within the GNU debugger, gdb[10]. You should have familiarity with gdb, but if you get stuck and the documentation is not illuminating, you can ask the tutors or ask a TA via email or on the course slack channel.

Note that the first time you run the kernel under gdb, you are likely to get an error message about not being able access the local .gdbinit file. In order to allow gdb to properly connect to your kernel, you will need to modify (or create) the .gdbinit file in your home directory to include a specific auto-load command, such as

```
add-auto-load-safe-path /u/markem/CS333/pdx-xv6/.gdbinit
```

Note that the gdb error message will specify the correct information to put in the .gdbinit file.

3.4.1 Alternate Symbol Table

By default, the xv6 environment will load the symbol table for the executing kernel. However, there are times when you will want to debug user programs inside of xv6. The gdb manual in section §2.1.1 Choosing Files covers selection of different symbol tables from other files.

There are two ways to load an alternate symbol table. One is to use the *file* command and the other is to use the *symbol-file* command. For example, the shell is a user-level program. To debug the shell, sh.c, you will need to load its symbol table. Use one of these gdb commands to load the symbol table for the shell.

```
file _sh
symbol-file sh.o
```

See also §18 Commands to Specify Files and §16 Examining the Symbol Table for additional gdb commands related to symbol tables.

3.4.2 Debugging a Crashed Kernel

The xv6 kernel "crashes" via a called to `panic()`. Unfortunately, this does not cause the kernel to halt, but rather "hang". In your gdb session, you can enter control-c to break to the gdb command prompt and inspect state from the gdb prompt. The `panic()` function will print out instruction pointer[6] information and you can view kernel.asm in an editor to trace the call path that (hopefully) led to the panic.

[5] Further information regarding console commands for QEMU can be found in §3.5 of [16].
[6] Called program counter (PC) in xv6.

3.4.3 Tracing Instruction Pointers

The terms "instruction pointer" and "program counter" are equivalent. As Intel uses "instruction pointer" (e.g., register eip), we will mostly use this term in lectures. Be aware that both terms have the same meaning.

When the kernel "panics"[7], the output will look something like

```
panic: Example panic message

80104ebb 801038f7 80103896 0 0 0 0 0 0 0
```

Normally, there will be a number to the right of the text for the `panic` message, but there is not one here. The line of numbers, however, represents the instruction pointer trace leading to the panic. We can trace the kernel execution by looking at the instruction pointers, most recent first, while viewing the decompiled kernel, `kernel.asm`. We must remember that the instruction pointer will point to the *next* instruction to be executed.

In `kernel.asm`, we search for address `80104ebb`. We find this code in the region of the instruction pointer:

```
  p = ptable.pReadyList[i];
80104e97:       8b 45 f4                mov    -0xc(%ebp),%eax
80104e9a:       05 cc 08 00 00          add    $0x8cc,%eax
80104e9f:       8b 04 85 a4 39 11 80    mov    -0x7feec65c(,%eax,4),%eax
80104ea6:       89 45 f0                mov    %eax,-0x10(%ebp)
  if (p) {
80104ea9:       83 7d f0 00             cmpl   $0x0,-0x10(%ebp)
80104ead:       74 0c                   je     80104ebb <scheduler+0x95>
    panic("Example panic message\n");
80104eaf:       c7 04 24 2f 91 10 80    movl   $0x8010912f,(%esp)
80104eb6:       e8 7f b6 ff ff          call   8010053a <panic>
    // Enable interrupts on this processor.
    sti();

    // Loop over free list looking for process to run.
    acquire(&ptable.lock);
    for (i=0; i<NPRIO; i++) {
80104ebb:       83 45 f4 01             addl   $0x1,-0xc(%ebp)
80104ebf:       83 7d f4 02             cmpl   $0x2,-0xc(%ebp)
80104ec3:       7e 81                   jle    80104e46 <scheduler+0x20>
```

which we would recognize as being part of the scheduler. This shows us the call to `panic()`. Don't worry, with practice you will get the hang of reading `.asm` files.

Now take a look at the address `801038f7`:

```
  xchg(&cpu->started, 1); // tell startothers() we're up
801038d7:       65 a1 00 00 00 00       mov    %gs:0x0,%eax
801038dd:       05 a8 00 00 00          add    $0xa8,%eax
801038e2:       c7 44 24 04 01 00 00    movl   $0x1,0x4(%esp)
```

[7] Makes a call to the `panic()` function.

```
801038e9:       00
801038ea:       89 04 24                mov    %eax,(%esp)
801038ed:       e8 df fe ff ff          call   801037d1 <xchg>
scheduler();     // start running processes
801038f2:       e8 2f 15 00 00          call   80104e26 <scheduler>

801038f7 <startothers>:
```

which we can see corresponds to a call to our `scheduler()` routine.

Finally, we look at the last address `80103896`

```
userinit();      // first user process
8010388c:       e8 4b 0f 00 00          call   801047dc <userinit>
// Finish setting up this processor in mpmain.
mpmain();
80103891:       e8 1a 00 00 00          call   801038b0 <mpmain>
80103896 <mpenter>:
}
```

which shows that the chain started with a call to `mpmain()`.

This instruction pointer trace (a chain of instruction pointers) shows that the kernel called the routine `mpmain` then called the routine `scheduler()` and then called the `panic()` routine. If we look at the file `main.c` at the end of the `main()` function we will see the call to `mpmain` that starts this all off. With practice, you should have a pretty good handle on tracing instruction pointers.

3.4.4 Inspecting Key Data Structures

The most common data structures that we will investigate are associated with processes. The `ptable` and `proc` structures are used to manage processes.

The `ptable` struct defined in proc.c:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

which contains an array of `proc` structures. This struct is defined in proc.h:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t *pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  uint pid;                    // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
```

```
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

If we are running the kernel from inside gdb and we get a panic message such as:

```
cpu1: panic: panic for printing out lock state

80104e6a 801038f7 801038b0 705a 0 0 0 0 0 0
```

Then we can go to the gdb session and enter control-c:

```
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
The target architecture is assumed to be i386
=> 0x8010555c <xchg>:   push   %ebp
xchg (addr=0x801139a0 <ptable>, newval=newval@entry=1) at x86.h:122
122     {
(gdb)
```

Since I happen to know where the `panic()` message is at, I decide that I want to track the instruction pointer values associated with the lock in the `ptable` structure which is defined in proc.c. Here is a listing from the "pcs" (program counters) array in `ptable.lock.pcs`:

```
(gdb) p/x ptable.lock.pcs
$1 = {0x80104e3d, 0x801038f7, 0x801038b0, 0x705a, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
```

which shows the most recent program counter first.

To trace these values, use the same procedure as in §3.4.3.

Other useful gdb commands will be discussed in class and on the slack channel.

3.5 The `argint()` and `argptr()` functions

The `arg*` family of functions available to kernel code in in xv6 provide a mechanism for passing arguments from a process running in user mode to xv6 running in kernel mode. We'll focus on two of them here: `argint()` and `argptr()`[8]. When a process invokes a system call and passes it arguments, the process pushes the arguments onto its stack. The flow of control then changes from the calling process to xv6.

At this point, all of the arguments are stored on the processes' user space stack. To access the arguments, the code for the system call needs to copy them from the user stack to memory that is easy for kernel code to use. One simple way of allocating that memory is to declare a local variable

[8]These functions are defined in `syscall.c`, take a look at the source for more details.

in a function in the kernel, to hold the copy. The declaration allocates memory for that variable on the kernel stack. Now we have somewhere to put the copy!

This is where the `arg*` functions come in: we can use them to copy arguments from the user stack to the easy-to-access variables on the kernel stack.

Both `arg*` functions take an index as the first argument, that specifies which argument on the user stack to copy. For example, if the first argument to `argint()` is 0, then `argint()` copies the first argument the user program passed to the system call. If the first argument is 1, then it copies the second argument, and so on.

The second argument is the address in memory where we want the arg* function to put the copy. Consider this example from `sysproc.c`:

```
int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}
```

This is part of the kernel code for the kill() system call. We know from the declaration of the user space wrapper for kill() in user.h:

```
int kill(int);
```

that `kill()` takes one argument, an int. So, `sys_kill()` just declares a local int on the kernel stack called pid, and then passes the index 0 and address of pid to `argint()`, so that `argint()` will copy the int that the user program passed to kill into pid. Now, after checking that `argint()` did not fail, `sys_kill()` has a local copy in pid of the user program's argument to `kill()`, which it then passes to the actual kernel implementation of `kill()` in proc.c[9].

`argint()` interprets the argument we copy from the user stack as an integer, while `argptr()` interprets it as a pointer to memory in the user program's address space. Consider this example from `sysfile.c`:

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
```

[9]Note that this `kill()` called from `sys_kill()` is not the `kill()` declared in `user.h`! User programs are compiled and linked separately from the kernel, so it's alright to have different functions in user space and in the kernel with the same names. The `kill()` declared in `user.h` is implemented in `usys.S`, and is available to user space programs like `kill.c`, while the `kill()` called from `sys_kill()` is declared in `defs.h` and implemented in `proc.c`, and is only available to kernel code.

```
    return -1;
  return fileread(f, p, n);
}
```

The declaration for the user space wrapper for this system call, `read()`, is also in `user.h`:

```
int read(int, void*, int);
```

The second argument to `read()` is a pointer to a chunk of memory in the user program's address space, and the third argument is an int, the size of that chunk of memory. We can see here that `sys_read()` passes `argptr()` the index 1 as its first argument: this is because the pointer we're copying is the second argument the user program passed to `read()`.

We can also see from this example that unlike `argint()`, `argptr()` takes a third argument: an `int`. This is the expected size in bytes of the chunk of memory that the pointer we're copying points to. `argptr()` uses this third size parameter along with the value of the pointer it's copying to check that the chunk's beginning and end are both inside the range of memory allocated for the process, and fails if this is not the case.

3.6 XV6 C Library

It is likely that students are used to using certain functions from the C library on a system such as Linux. It is important to note that the C library for xv6 is very minimal. Much of the library is implemented in `ulib.c` but some functions are implemented in separate files. For example, the `printf()` function for xv6 is implemented in `printf.c` and provides only a minimal subset of the functionality that students may be used to. The full list of routines in the C library can be found in the file `user.h`.

While it is possible to expand the functionality of xv6 C library routines, and even add new ones, note that this is *not* the purpose of this class. All assignments can be completed without modifying or extending the existing C library beyond what is described in the assignments. An obstacle for many students is be the lack of formatting options in `printf()` but careful thought will allow you to get formatting as necessary although some coding will be required.

As far as **documentation for the xv6 C library**, there isn't any. The xv6 book does describe some facets of some library calls but mostly it is left to the programmer to read the code to determine proper usage. This is normal in systems development and good to practice.

Bibliography

[1] Alex Aiken, *MOSS: A System for Detecting Software Plagiarism*, http://theory.stanford.edu/ aiken/moss/

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, V0.9, May 2015.

[3] Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective, third edition*, Pearson Education, 2016, ISBN 978-0-13-409266-9. Website.

[4] Russ Cox, Frans Kaashoek, and Robert Morris, *xv6: a simple, Unix-like teaching operating system*, Draft as of September 3, 2014, xv6-book@pdos.csail.mit.edu. Local copy.

[5] Russ Cox, Frans Kaashoek, and Robert Morris, *xv6: a simple, Unix-like teaching operating system*, source code listing. Local copy.

[6] Mark Morrissey, *xv6 Survival Guide*, PDF, LaTeX source.

[7] Free Software Foundation, Inc., *The C Preprocessor*.

[8] Free Software Foundation, Inc., *GCC Version 5.3.0*.

[9] Free Software Foundation, Inc., *GCC Make Manual*.

[10] Free Software Foundation, Inc., *Debugging with gdb: the gnu Source-Level Debugger (GDB)*.

[11] Git User Manual.

[12] Intel Corp., *Intel 80386 Programmer's Reference Manual*, 1987, website.

[13] Al Kelley and Ira Pohl, *A Book on C, Fourth Edition*, Addison Wesley, 1998, ISBN 0-201-18399-4.

[14] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall, 1988, ISBN 0-13-110362-8.

[15] Tobias Oetiker, Hubert Partl, Irene Hyna and Elisabeth Schlegl, *The Not So Short Introduction to LaTeX 2ε, Version 5.05*, July 18, 2015.

[16] QEMU: Open Source Processor Emulator. QEMU Documentation.

[17] W. Richard Stevens and Stephen A. Rago, *Advanced Programming in the UNIX Environment, Third Edition*, Pearson Education, 2013, ISBN 978-0-321-63773-4.

[18] MIT course 6.828: Operating System Engineering.