Experiment introduction

The source of this experiment is:https://pdos.csail.mit.edu/6.828/2020/

Tools Used in 6.S081

For this class you'll need the RISC-V versions of a couple different tools: QEMU 5.1, GDB 8.3, GCC, and Binutils.

We highly recommend using a Debathena machine, such as athena.dialup.mit.edu, to work on the labs. If you use the MIT Athena machines that run Linux, then all of these tools are located in the 6.828 locker: just type 'add -f 6.828' to get access to them. If you don't have access to a Debathena machine, you can install the tools directly or use virtual machine with Linux via the instructions below.

If you are having trouble getting things set up, please come by to office hours or post on Piazza. We're happy to help!

Using Athena

ssh into one of the Athena dialup machines and add the tools:

```
$ ssh {your kerberos}@athena.dialup.mit.edu
$ add -f 6.828
```

Installing on macOS

First, install developer tools:

```
$ xcode-select --install
```

Next, install Homebrew, a package manager for macOS:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Next, install the RISC-V compiler toolchain:

```
$ brew tap riscv/riscv
$ brew install riscv-tools
```

The brew formula may not link into `/usr/local`. You will need to update your shell's rc file (e.g. ~/.bashrc) to add the appropriate directory to $PATH.

```
PATH=$PATH:/usr/local/opt/riscv-gnu-toolchain/bin
```

Finally, install QEMU:

```
brew install qemu
```

Installing via APT (Debian/Ubuntu)

Make sure you are running either "bullseye" or "sid" for your debian version (on ubuntu this can be checked by running `cat /etc/debian_version`), then run:

```
sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

(The version of QEMU on "buster" is too old, so you'd have to get that separately.)

qemu-system-misc fix

At this moment in time, it seems that the package `qemu-system-misc` has received an update that breaks its compatibility with our kernel. If you run `make qemu` and the script appears to hang after
`qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0`
you'll need to uninstall that package and install an older version:

```
  $ sudo apt-get remove qemu-system-misc
  $ sudo apt-get install qemu-system-misc=1:4.2-3ubuntu6
```

Installing on Arch

```
sudo pacman -S riscv64-linux-gnu-binutils riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb qemu-arch-extra
```

Other Linux distributions (i.e. compiling your own toolchain)

We assume that you are installing the toolchain into `/usr/local` on a modern Ubuntu installation. You will need a fair amount of disk space to compile the tools (around 9GiB). If you don't have that much space, consider using an MIT Athena machine.

First, clone the repository for the RISC-V GNU Compiler Toolchain:

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

Next, make sure you have the packages needed to compile the toolchain:

```
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

Configure and build the toolchain:

```
$ cd riscv-gnu-toolchain
$ ./configure --prefix=/usr/local
$ sudo make
$ cd ..
```

Next, retrieve and extract the source for QEMU 5.1.0:

```
$ wget https://download.qemu.org/qemu-5.1.0.tar.xz
$ tar xf qemu-5.1.0.tar.xz
```

Build QEMU for riscv64-softmmu:

```
$ cd qemu-5.1.0
$ ./configure --disable-kvm --disable-werror --prefix=/usr/local --target-
list="riscv64-softmmu"
$ make
$ sudo make install
$ cd ..
```

Windows (i.e. running a Linux VM)

We haven't tested it, but it might be possible to get everything you need via the Windows Subsystem for Linux or otherwise compiling the tools yourself.

However, an easier option would probably be to run a virtual machine with one of the other operating systems listed above. With platform virtualization, Linux can cohabitate with your normal computing environment. Installing a Linux virtual machine is a two step process. First, you download the virtualization platform.

- **VirtualBox** (free for Mac, Linux, Windows) — Download page
- VMware Player (free for Linux and Windows, registration required)
- VMware Fusion (Downloadable from IS&T for free).

VirtualBox is a little slower and less flexible, but free!

Once the virtualization platform is installed, download a boot disk image for the Linux distribution of your choice.

- Ubuntu Desktop is one option.

This will download a file named something like `ubuntu-18.04.3-desktop-amd64.iso`. Start up your virtualization platform and create a new (64-bit) virtual machine. Use the downloaded Ubuntu image as a boot disk; the procedure differs among VMs but is pretty simple.

Testing your Installation

To test your installation, you should be able to check the following:

```
$ riscv64-unknown-elf-gcc --version
riscv64-unknown-elf-gcc (GCC) 10.1.0
...

$ qemu-system-riscv64 --version
QEMU emulator version 5.1.0
```

You should also be able to compile and run xv6:

```
# in the xv6 directory
$ make qemu
# ... lots of output ...
init: starting sh
$
```

To quit qemu type: Ctrl-a x.

Lab guidance

Hardness of assignments

Each assignment indicates how difficult it is:

- Easy: less than an hour. These exercise are typically often warm-up exercises for subsequent exercises.
- Moderate: 1-2 hours.
- Hard: More than 2 hours. Often these exercises don't require much code, but the code is tricky to get right.

These times are rough estimates of our expectations. For some of the optional assignments we don't have a solution and the hardness is a wild guess. If you find yourself spending more time on an assignment than we expect, please reach out on piazza or come to office hours.

The exercises in general require not many lines of code (tens to a few hundred lines), but the code is conceptually complicated and often details matter a lot. So, make sure you do the assigned reading for the labs, read the relevant files through, consult the documentation (the RISC-V manuals etc. are on the reference page) before you write any code. Only when you have a firm grasp of the assignment and solution, then start coding. When you start coding, implement your solution in small steps (the assignments often suggest how to break the problem down in smaller steps) and test whether each steps works before proceeding to the next one.

Warning: don't start a lab the night before a lab is due; it much more time efficient to do the labs in several sessions spread over multiple days. The manifestation of a bug in operating system kernel can be bewildering and may require much thought and careful debugging to understand and fix.

Debugging tips

Here are some tips for debugging your solutions:

- Make sure you understand C and pointers. The book "The C programming language (second edition)" by Kernighan and Ritchie is a succinct description of C. Some useful pointer exercises are here. Unless you are already thoroughly versed in C, do not skip or skim the pointer exercises above. If you do not really understand pointers in C, you will suffer untold pain and misery in the labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

  A few pointer common idioms are in particular worth remembering:

  - If `int *p = (int*)100`, then `(int)p + 1` and `(int)(p + 1)` are different numbers: the first is `101` but the second is `104`. When adding an integer to a pointer, as in the

second case, the integer is implicitly multiplied by the size of the object the pointer points to.
    - `p[i]` is defined to be the same as `*(p+i)`, referring to the i'th object in the memory pointed to by p. The above rule for addition helps this definition work when the objects are larger than one byte.
    - `&p[i]` is the same as `(p+i)`, yielding the address of the i'th object in the memory pointed to by p.

  Although most C programs never need to cast between pointers and integers, operating systems frequently do. Whenever you see an addition involving a memory address, ask yourself whether it is an integer addition or pointer addition and make sure the value being added is appropriately multiplied or not.

- If you have an exercise partially working, checkpoint your progress by committing your code. If you break something later, you can then roll back to your checkpoint and go forward in smaller steps. To learn more about Git, take a look at the Git user's manual, or, you may find this CS-oriented overview of Git useful.
- If you fail a test, make sure you understand why your code fails the test. Insert print statements until you understand what is going on.
- You may find that your print statements may produce much output that you would like to search through; one way to do that is to run `make qemu` inside of `script` (run man script on your machine), which logs all console output to a file, which you can then search. Don't forget to exit `script`.
- In many cases, print statements will be sufficient, but sometimes being able to single step through some assembly code or inspecting the variables on the stack is helpful. To use gdb with xv6, run make make qemu-gdb in one window, run gdb (or riscv64-linux-gnu-gdb) in another window, set a break point, followed by followed by 'c' (continue), and xv6 will run until it hits the breakpoint. (See Using the GNU Debugger for helpful GDB tips.)
- If you want to see what the assembly is that the compiler generates for the kernel or to find out what the instruction is at a particular kernel address, see the file `kernel.asm`, which the Makefile produces when it compiles the kernel. (The Makefile also produces `.asm` for all user programs.)
- If the kernel panics, it will print an error message listing the value of the program counter when it crashed; you can search `kernel.asm` to find out in which function the program counter was when it crashed, or you can run addr2line -e kernel/kernel *pc-value* (run man addr2line for details). If you want to get backtrace, restart using gdb: run 'make qemu-gdb' in one window, run gdb (or riscv64-linux-gnu-gdb) in another window, set breakpoint in panic ('b panic'), followed by followed by 'c' (continue). When the kernel hits the break point, type 'bt' to get a backtrace.
- If your kernel hangs (e.g., due to a deadlock) or cannot execute further (e.g., due to a page fault when executing a kernel instruction), you can use gdb to find out where it is hanging. Run run 'make qemu-gdb' in one window, run gdb (riscv64-linux-gnu-gdb) in another window, followed by followed by 'c' (continue). When the kernel appears to hang hit Ctrl-C in the qemu-gdb window and type 'bt' to get a backtrace.
- `qemu` has a "monitor" that lets you query the state of the emulated machine. You can get at it by typing control-a c (the "c" is for console). A particularly useful monitor command is `info mem` to print the page table. You may need to use the `cpu` command to select which core `info mem` looks at, or you could start qemu with `make CPUS=1 qemu` to cause there to be just one core.

It is well worth the time learning the above-mentioned tools.

Lab: Xv6 and Unix utilities

This lab will familiarize you with xv6 and its system calls.

Boot xv6 (easy)

You can do these labs on an Athena machine or on your own computer. If you use your own computer, have a look at the lab tools page for setup tips.

If you use Athena, you must use an x86 machine; that is, `uname -a` should mention `i386 GNU/Linux` or `i686 GNU/Linux` or `x86_64 GNU/Linux`. You can log into a public Athena host with `ssh -X athena.dialup.mit.edu`. We have set up the appropriate compilers and simulators for you on Athena. To use them, run add -f 6.828. You must run this command every time you log in (or add it to your `~/.environment` file). If you get obscure errors while compiling or running `qemu`, check that you added the course locker.

Fetch the xv6 source for the lab and check out the `util` branch:

```
$ git clone git://g.csail.mit.edu/xv6-labs-2020
Cloning into 'xv6-labs-2020'...
...
$ cd xv6-labs-2020
$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

The xv6-labs-2020 repository differs slightly from the book's xv6-riscv; it mostly adds some files. If you are curious look at the git log:

```
$ git log
```

The files you will need for this and subsequent lab assignments are distributed using the Git version control system. Above you switched to a branch (git checkout util) containing a version of xv6 tailored to this lab. To learn more about Git, take a look at the Git user's manual, or, you may find this CS-oriented overview of Git useful. Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can *commit* your changes by running:

```
$ git commit -am 'my solution for util lab exercise 1'
Created commit 60d2135: my solution for util lab exercise 1
 1 files changed, 1 insertions(+), 0 deletions(-)
$
```

You can keep track of your changes by using the git diff command. Running git diff will display the changes to your code since your last commit, and git diff origin/util will display the changes relative to the initial xv6-labs-2020 code. Here, `origin/xv6-labs-2020` is the name of the git branch with the initial code you downloaded for the class.

Build and run xv6:

```
$ make qemu
riscv64-unknown-elf-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL
-MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-
stack-protector -fno-pie -no-pie   -c -o kernel/start.o kernel/start.c
...
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_zombie
user/zombie.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_zombie > user/zombie.asm
riscv64-unknown-elf-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > user/zombie.sym
mkfs/mkfs fs.img README  user/xargstest.sh user/_cat user/_echo user/_forktest
user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh
user/_stressfs user/_usertests user/_grind user/_wc user/_zombie
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954
total 1000
balloc: first 591 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3
-nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-
device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```

If you type `ls` at the prompt, you should see output similar to the following:

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2059
xargstest.sh   2 3 93
cat            2 4 24256
echo           2 5 23080
forktest       2 6 13272
grep           2 7 27560
init           2 8 23816
kill           2 9 23024
ln             2 10 22880
ls             2 11 26448
mkdir          2 12 23176
rm             2 13 23160
sh             2 14 41976
stressfs       2 15 24016
usertests      2 16 148456
grind          2 17 38144
wc             2 18 25344
zombie         2 19 22408
console        3 20 0
```

These are the files that `mkfs` includes in the initial file system; most are programs you can run. You just ran one of them: `ls`.

xv6 has no `ps` command, but, if you type Ctrl-p, the kernel will print information about each process. If you try it now, you'll see two lines: one for `init`, and one for `sh`.

To quit qemu type: Ctrl-a x.

Grading and hand-in procedure

You can run make grade to test your solutions with the grading program. The TAs will use the same grading program to assign your lab submission a grade. Separately, we will also have check-off meetings for labs (see Grading policy).

The lab code comes with GNU Make rules to make submission easier. After committing your final changes to the lab, type make handin to submit your lab. For detailed instructions on how to submit see below.

sleep (easy)

Implement the UNIX program `sleep` for xv6; your `sleep` should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file `user/sleep.c`.

Some hints:

- Before you start coding, read Chapter 1 of the xv6 book.
- Look at some of the other programs in `user/` (e.g., `user/echo.c`, `user/grep.c`, and `user/rm.c`) to see how you can obtain the command-line arguments passed to a program.
- If the user forgets to pass an argument, sleep should print an error message.
- The command-line argument is passed as a string; you can convert it to an integer using `atoi` (see user/ulib.c).
- Use the system call `sleep`.
- See `kernel/sysproc.c` for the xv6 kernel code that implements the `sleep` system call (look for `sys_sleep`), `user/user.h` for the C definition of `sleep` callable from a user program, and `user/usys.S` for the assembler code that jumps from user code into the kernel for `sleep`.
- Make sure `main` calls `exit()` in order to exit your program.
- Add your `sleep` program to `UPROGS` in Makefile; once you've done that, `make qemu` will compile your program and you'll be able to run it from the xv6 shell.
- Look at Kernighan and Ritchie's book *The C programming language (second edition)* (K&R) to learn about C.

Run the program from the xv6 shell:

```
$ make qemu
...
init: starting sh
$ sleep 10
(nothing happens for a little while)
$
```

Your solution is correct if your program pauses when run as shown above. Run make grade to see if you indeed pass the sleep tests.

Note that make grade runs all tests, including the ones for the assignments below. If you want to run the grade tests for one assignment, type:

```
$ ./grade-lab-util sleep
```

This will run the grade tests that match "sleep". Or, you can type:

```
$ make GRADEFLAGS=sleep grade
```

which does the same.

pingpong (easy)

Write a program that uses UNIX system calls to "ping-pong" a byte between two processes over a pair of pipes, one for each direction. The parent should send a byte to the child; the child should print ": received ping", where is its process ID, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print ": received pong", and exit. Your solution should be in the file `user/pingpong.c`.

Some hints:

- Use `pipe` to create a pipe.
- Use `fork` to create a child.
- Use `read` to read from the pipe, and `write` to write to the pipe.
- Use `getpid` to find the process ID of the calling process.
- Add the program to `UPROGS` in Makefile.
- User programs on xv6 have a limited set of library functions available to them. You can see the list in `user/user.h`; the source (other than for system calls) is in `user/ulib.c`, `user/printf.c`, and `user/umalloc.c`.

Run the program from the xv6 shell and it should produce the following output:

```
$ make qemu
...
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

Your solution is correct if your program exchanges a byte between two processes and produces output as shown above.

primes (moderate)/(hard)

Write a concurrent version of prime sieve using pipes. This idea is due to Doug McIlroy, inventor of Unix pipes. The picture halfway down this page and the surrounding text explain how to do it. Your solution should be in the file `user/primes.c`.

Your goal is to use `pipe` and `fork` to set up the pipeline. The first process feeds the numbers 2 through 35 into the pipeline. For each prime number, you will arrange to create one process that reads from its left neighbor over a pipe and writes to its right neighbor over another pipe. Since xv6 has limited number of file descriptors and processes, the first process can stop at 35.

Some hints:

- Be careful to close file descriptors that a process doesn't need, because otherwise your program will run xv6 out of resources before the first process reaches 35.
- Once the first process reaches 35, it should wait until the entire pipeline terminates, including all children, grandchildren, &c. Thus the main primes process should only exit after all the output has been printed, and after all the other primes processes have exited.
- Hint: `read` returns zero when the write-side of a pipe is closed.
- It's simplest to directly write 32-bit (4-byte) `int`s to the pipes, rather than using formatted ASCII I/O.
- You should create the processes in the pipeline only as they are needed.
- Add the program to `UPROGS` in Makefile.

Your solution is correct if it implements a pipe-based sieve and produces the following output:

```
$ make qemu
...
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

find (moderate)

Write a simple version of the UNIX find program: find all the files in a directory tree with a specific name. Your solution should be in the file `user/find.c`.

Some hints:

- Look at user/ls.c to see how to read directories.
- Use recursion to allow find to descend into sub-directories.
- Don't recurse into "." and "..".
- Changes to the file system persist across runs of qemu; to get a clean file system run make clean and then make qemu.
- You'll need to use C strings. Have a look at K&R (the C book), for example Section 5.5.
- Note that == does not compare strings like in Python. Use strcmp() instead.
- Add the program to `UPROGS` in Makefile.

Your solution is correct if produces the following output (when the file system contains the files `b` and `a/b`):

```
$ make qemu
...
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$
```

xargs (moderate)

Write a simple version of the UNIX xargs program: read lines from the standard input and run a command for each line, supplying the line as arguments to the command. Your solution should be in the file `user/xargs.c`.

The following example illustrates xarg's behavior:

```
$ echo hello too | xargs echo bye
bye hello too
$
```

Note that the command here is "echo bye" and the additional arguments are "hello too", making the command "echo bye hello too", which outputs "bye hello too".

Please note that xargs on UNIX makes an optimization where it will feed more than argument to the command at a time. We don't expect you to make this optimization. To make xargs on UNIX behave the way we want it to for this lab, please run it with the -n option set to 1. For instance

```
$ echo "1\n2" | xargs -n 1 echo line
line 1
line 2
$
```

Some hints:

- Use `fork` and `exec` to invoke the command on each line of input. Use `wait` in the parent to wait for the child to complete the command.
- To read individual lines of input, read a character at a time until a newline ('\n') appears.
- kernel/param.h declares MAXARG, which may be useful if you need to declare an argv array.
- Add the program to `UPROGS` in Makefile.
- Changes to the file system persist across runs of qemu; to get a clean file system run make clean and then make qemu.

xargs, find, and grep combine well:

```
$ find . b | xargs grep hello
```

will run "grep hello" on each file named b in the directories below ".".

To test your solution for xargs, run the shell script xargstest.sh. Your solution is correct if it produces the following output:

```
$ make qemu
...
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $
```

You may have to go back and fix bugs in your find program. The output has many `$` because the xv6 shell doesn't realize it is processing commands from a file instead of from the console, and prints a `$` for each command in the file.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Write an uptime program that prints the uptime in terms of ticks using the `uptime` system call. (easy)
- Support regular expressions in name matching for `find`. `grep.c` has some primitive support for regular expressions. (easy)
- The xv6 shell (`user/sh.c`) is just another user program and you can improve it. It is a minimal shell and lacks many features found in real shell. For example, modify the shell to not print a `$` when processing shell commands from a file (moderate), modify the shell to support wait (easy), modify the shell to support lists of commands, separated by ";" (moderate), modify the shell to support sub-shells by implementing "(" and ")" (moderate), modify the shell to support tab completion (easy), modify the shell to keep a history of passed shell commands (moderate), or anything else you would like your shell to do. (If you are very ambitious, you may have to modify the kernel to support the kernel features you need; xv6 doesn't support much.)

Lab: system calls

In the last lab you used systems calls to write a few utilities. In this lab you will add some new system calls to xv6, which will help you understand how they work and will expose you to some of the internals of the xv6 kernel. You will add more system calls in later labs.

Before you start coding, read Chapter 2 of the xv6 book, and Sections 4.3 and 4.4 of Chapter 4, and related source files:

- The user-space code for systems calls is in `user/user.h` and `user/usys.pl`.
- The kernel-space code is `kernel/syscall.h`, kernel/syscall.c.
- The process-related code is `kernel/proc.h` and `kernel/proc.c`.

To start the lab, switch to the syscall branch:

```
$ git fetch
$ git checkout syscall
$ make clean
```

If you run, make grade, you will see that the grading script cannot exec `trace` and `sysinfotest`. Your job is to add the necessary system calls and stubs to make them work.

System call tracing (moderate)

In this assignment you will add a system call tracing feature that may help you when debugging later labs. You'll create a new `trace` system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls `trace(1 << SYS_fork)`, where `SYS_fork` is a syscall number from `kernel/syscall.h`. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The `trace` system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

We provide a `trace` user-level program that runs another program with tracing enabled (see `user/trace.c`). When you're done, you should see output like this:

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: 407: syscall fork -> 408
408: syscall fork -> 409
409: syscall fork -> 410
410: syscall fork -> 411
409: syscall fork -> 412
410: syscall fork -> 413
409: syscall fork -> 414
411: syscall fork -> 415
...
$
```

In the first example above, trace invokes grep tracing just the read system call. The 32 is `1<<SYS_read`. In the second example, trace runs grep while tracing all system calls; the 2147583647 has all 31 low bits set. In the third example, the program isn't traced, so no trace output is printed. In the fourth example, the fork system calls of all the descendants of the `forkforkfork` test in `usertests` are being traced. Your solution is correct if your program behaves as shown above (though the process IDs may be different).

Some hints:

- Add `$U/_trace` to UPROGS in Makefile
- Run make qemu and you will see that the compiler cannot compile `user/trace.c`, because the user-space stubs for the system call don't exist yet: add a prototype for the system call to `user/user.h`, a stub to `user/usys.pl`, and a syscall number to `kernel/syscall.h`. The Makefile invokes the perl script `user/usys.pl`, which produces `user/usys.S`, the actual system call stubs, which use the RISC-V `ecall` instruction to transition to the kernel. Once you fix the compilation issues, run trace 32 grep hello README; it will fail because you haven't implemented the system call in the kernel yet.
- Add a `sys_trace()` function in `kernel/sysproc.c` that implements the new system call by remembering its argument in a new variable in the `proc` structure (see `kernel/proc.h`). The functions to retrieve system call arguments from user space are in `kernel/syscall.c`, and you can see examples of their use in `kernel/sysproc.c`.
- Modify `fork()` (see `kernel/proc.c`) to copy the trace mask from the parent to the child process.
- Modify the `syscall()` function in `kernel/syscall.c` to print the trace output. You will need to add an array of syscall names to index into.

Sysinfo (moderate)

In this assignment you will add a system call, `sysinfo`, that collects information about the running system. The system call takes one argument: a pointer to a `struct sysinfo` (see `kernel/sysinfo.h`). The kernel should fill out the fields of this struct: the `freemem` field should be set to the number of bytes of free memory, and the `nproc` field should be set to the number of processes whose `state` is not `UNUSED`. We provide a test program `sysinfotest`; you pass this assignment if it prints "sysinfotest: OK".

Some hints:

- Add `$U/_sysinfotest` to UPROGS in Makefile
- Run make qemu; `user/sysinfotest.c` will fail to compile. Add the system call sysinfo, following the same steps as in the previous assignment. To declare the prototype for sysinfo() `in user/user.h` you need predeclare the existence of `struct sysinfo`:

  ```
  struct sysinfo;
  int sysinfo(struct sysinfo *);
  ```

  Once you fix the compilation issues, run sysinfotest; it will fail because you haven't implemented the system call in the kernel yet.
- sysinfo needs to copy a `struct sysinfo` back to user space; see `sys_fstat()` (`kernel/sysfile.c`) and `filestat()` (`kernel/file.c`) for examples of how to do that using `copyout()`.

- To collect the amount of free memory, add a function to `kernel/kalloc.c`
- To collect the number of processes, add a function to `kernel/proc.c`

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Print the system call arguments for traced system calls (easy).
- Compute the load average and export it through sysinfo(moderate).

Lab: page tables

In this lab you will explore page tables and modify them to simplify the functions that copy data from user space to kernel space.

Before you start coding, read Chapter 3 of the xv6 book, and related files:

- `kern/memlayout.h`, which captures the layout of memory.
- `kern/vm.c`, which contains most virtual memory (VM) code.
- `kernel/kalloc.c`, which contains code for allocating and freeing physical memory.

To start the lab, switch to the pgtbl branch:

```
$ git fetch
$ git checkout pgtbl
$ make clean
```

Print a page table (easy)

To help you learn about RISC-V page tables, and perhaps to aid future debugging, your first task is to write a function that prints the contents of a page table.

Define a function called `vmprint()`. It should take a `pagetable_t` argument, and print that pagetable in the format described below. Insert `if(p->pid==1) vmprint(p->pagetable)` in exec.c just before the `return argc`, to print the first process's page table. You receive full credit for this assignment if you pass the `pte printout` test of `make grade`.

Now when you start xv6 it should print output like this, describing the page table of the first process at the point when it has just finished `exec()`ing `init`:

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

The first line displays the argument to `vmprint`. After that there is a line for each PTE, including PTEs that refer to page-table pages deeper in the tree. Each PTE line is indented by a number of `".."` that indicates its depth in the tree. Each PTE line shows the PTE index in its page-table page, the pte bits, and the physical address extracted from the PTE. Don't print PTEs that are not valid. In the above example, the top-level page-table page has mappings for entries 0 and 255. The next level down for entry 0 has only index 0 mapped, and the bottom-level for that index 0 has entries 0, 1, and 2 mapped.

Your code might emit different physical addresses than those shown above. The number of entries and the virtual addresses should be the same.

Some hints:

- You can put `vmprint()` in `kernel/vm.c`.
- Use the macros at the end of the file kernel/riscv.h.
- The function `freewalk` may be inspirational.
- Define the prototype for `vmprint` in kernel/defs.h so that you can call it from exec.c.
- Use `%p` in your printf calls to print out full 64-bit hex PTEs and addresses as shown in the example.

Explain the output of `vmprint` in terms of Fig 3-4 from the text. What does page 0 contain? What is in page 2? When running in user mode, could the process read/write the memory mapped by page 1?

A kernel page table per process (hard)

Xv6 has a single kernel page table that's used whenever it executes in the kernel. The kernel page table is a direct mapping to physical addresses, so that kernel virtual address *x* maps to physical address *x*. Xv6 also has a separate page table for each process's user address space, containing only mappings for that process's user memory, starting at virtual address zero. Because the kernel page table doesn't contain these mappings, user addresses are not valid in the kernel. Thus, when the kernel needs to use a user pointer passed in a system call (e.g., the buffer pointer passed to `write()`), the kernel must first translate the pointer to a physical address. The goal of this section and the next is to allow the kernel to directly dereference user pointers.

Your first job is to modify the kernel so that every process uses its own copy of the kernel page table when executing in the kernel. Modify `struct proc` to maintain a kernel page table for each process, and modify the scheduler to switch kernel page tables when switching processes. For this step, each per-process kernel page table should be identical to the existing global kernel page table. You pass this part of the lab if `usertests` runs correctly.

Read the book chapter and code mentioned at the start of this assignment; it will be easier to modify the virtual memory code correctly with an understanding of how it works. Bugs in page table setup can cause traps due to missing mappings, can cause loads and stores to affect unexpected pages of physical memory, and can cause execution of instructions from incorrect pages of memory.

Some hints:

- Add a field to `struct proc` for the process's kernel page table.
- A reasonable way to produce a kernel page table for a new process is to implement a modified version of `kvminit` that makes a new page table instead of modifying `kernel_pagetable`. You'll want to call this function from `allocproc`.

- Make sure that each process's kernel page table has a mapping for that process's kernel stack. In unmodified xv6, all the kernel stacks are set up in `procinit`. You will need to move some or all of this functionality to `allocproc`.
- Modify `scheduler()` to load the process's kernel page table into the core's `satp` register (see `kvminithart` for inspiration). Don't forget to call `sfence_vma()` after calling `w_satp()`.
- `scheduler()` should use `kernel_pagetable` when no process is running.
- Free a process's kernel page table in `freeproc`.
- You'll need a way to free a page table without also freeing the leaf physical memory pages.
- `vmprint` may come in handy to debug page tables.
- It's OK to modify xv6 functions or add new functions; you'll probably need to do this in at least `kernel/vm.c` and `kernel/proc.c`. (But, don't modify `kernel/vmcopyin.c`, `kernel/stats.c`, `user/usertests.c`, and `user/stats.c`.)
- A missing page table mapping will likely cause the kernel to encounter a page fault. It will print an error that includes `sepc=0x00000000XXXXXXXX`. You can find out where the fault occurred by searching for `XXXXXXXX` in `kernel/kernel.asm`.

Simplify `copyin/copyinstr` (hard)

The kernel's `copyin` function reads memory pointed to by user pointers. It does this by translating them to physical addresses, which the kernel can directly dereference. It performs this translation by walking the process page-table in software. Your job in this part of the lab is to add user mappings to each process's kernel page table (created in the previous section) that allow `copyin` (and the related string function `copyinstr`) to directly dereference user pointers.

Replace the body of `copyin` in `kernel/vm.c` with a call to `copyin_new` (defined in `kernel/vmcopyin.c`); do the same for `copyinstr` and `copyinstr_new`. Add mappings for user addresses to each process's kernel page table so that `copyin_new` and `copyinstr_new` work. You pass this assignment if `usertests` runs correctly and all the `make grade` tests pass.

This scheme relies on the user virtual address range not overlapping the range of virtual addresses that the kernel uses for its own instructions and data. Xv6 uses virtual addresses that start at zero for user address spaces, and luckily the kernel's memory starts at higher addresses. However, this scheme does limit the maximum size of a user process to be less than the kernel's lowest virtual address. After the kernel has booted, that address is `0xC000000` in xv6, the address of the PLIC registers; see `kvminit()` in `kernel/vm.c`, `kernel/memlayout.h`, and Figure 3-4 in the text. You'll need to modify xv6 to prevent user processes from growing larger than the PLIC address.

Some hints:

- Replace `copyin()` with a call to `copyin_new` first, and make it work, before moving on to `copyinstr`.
- At each point where the kernel changes a process's user mappings, change the process's kernel page table in the same way. Such points include `fork()`, `exec()`, and `sbrk()`.
- Don't forget that to include the first process's user page table in its kernel page table in `userinit`.
- What permissions do the PTEs for user addresses need in a process's kernel page table? (A page with `PTE_U` set cannot be accessed in kernel mode.)
- Don't forget about the above-mentioned PLIC limit.

Linux uses a technique similar to what you have implemented. Until a few years ago many kernels used the same per-process page table in both user and kernel space, with mappings for both user and kernel addresses, to avoid having to switch page tables when switching between user and kernel space. However, that setup allowed side-channel attacks such as Meltdown and Spectre.

Explain why the third test `srcva + len < srcva` is necessary in `copyin_new()`: give values for `srcva` and `len` for which the first two test fail (i.e., they will not cause to return -1) but for which the third one is true (resulting in returning -1).

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Use super-pages to reduce the number of PTEs in page tables.
- Extend your solution to support user programs that are as large as possible; that is, eliminate the restriction that user programs be smaller than PLIC.
- Unmap the first page of a user process so that dereferencing a null pointer will result in a fault. You will have to start the user text segment at, for example, 4096, instead of 0.

Lab: traps

This lab explores how system calls are implemented using traps. You will first do a warm-up exercises with stacks and then you will implement an example of user-level trap handling.

Before you start coding, read Chapter 4 of the xv6 book, and related source files:

- `kernel/trampoline.S`: the assembly involved in changing from user space to kernel space and back
- `kernel/trap.c`: code handling all interrupts

To start the lab, switch to the trap branch:

```
$ git fetch
$ git checkout traps
$ make clean
```

RISC-V assembly (easy)

It will be important to understand a bit of RISC-V assembly, which you were exposed to in 6.004. There is a file `user/call.c` in your xv6 repo. make fs.img compiles it and also produces a readable assembly version of the program in `user/call.asm`.

Read the code in call.asm for the functions `g`, `f`, and `main`. The instruction manual for RISC-V is on the reference page. Here are some questions that you should answer (store the answers in a file answers-traps.txt):

Which registers contain arguments to functions? For example, which register holds 13 in main's call to `printf`?

Where is the call to function `f` in the assembly code for main? Where is the call to `g`? (Hint: the compiler may inline functions.)

At what address is the function `printf` located?

What value is in the register `ra` just after the `jalr` to `printf` in `main`?

Run the following code.

```
    unsigned int i = 0x00646c72;
    printf("H%x Wo%s", 57616, &i);
```

What is the output? Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

Here's a description of little- and big-endian and a more whimsical description.

In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
    printf("x=%d y=%d", 3);
```

Backtrace (moderate)

For debugging it is often useful to have a backtrace: a list of the function calls on the stack above the point at which the error occurred.

Implement a `backtrace()` function in `kernel/printf.c`. Insert a call to this function in `sys_sleep`, and then run bttest, which calls `sys_sleep`. Your output should be as follows:

```
backtrace:
0x0000000080002cda
0x0000000080002bb6
0x0000000080002898
```

After `bttest` exit qemu. In your terminal: the addresses may be slightly different but if you run `addr2line -e kernel/kernel` (or `riscv64-unknown-elf-addr2line -e kernel/kernel`) and cut-and-paste the above addresses as follows:

```
    $ addr2line -e kernel/kernel
    0x0000000080002de2
    0x0000000080002f4a
    0x0000000080002bfc
    Ctrl-D
```

You should see something like this:

```
kernel/sysproc.c:74
kernel/syscall.c:224
kernel/trap.c:85
```

The compiler puts in each stack frame a frame pointer that holds the address of the caller's frame pointer. Your `backtrace` should use these frame pointers to walk up the stack and print the saved return address in each stack frame.

Some hints:

- Add the prototype for backtrace to `kernel/defs.h` so that you can invoke `backtrace` in `sys_sleep`.
- The GCC compiler stores the frame pointer of the currently executing function in the register

  s0

  . Add the following function to

  kernel/riscv.h

  :

  ```
  static inline uint64
  r_fp()
  {
    uint64 x;
    asm volatile("mv %0, s0" : "=r" (x) );
    return x;
  }
  ```

  and call this function in

  backtrace

  to read the current frame pointer. This function uses

  in-line assembly

  to read

  s0

  .

- These lecture notes have a picture of the layout of stack frames. Note that the return address lives at a fixed offset (-8) from the frame pointer of a stackframe, and that the saved frame pointer lives at fixed offset (-16) from the frame pointer.
- Xv6 allocates one page for each stack in the xv6 kernel at PAGE-aligned address. You can compute the top and bottom address of the stack page by using `PGROUNDDOWN(fp)` and `PGROUNDUP(fp)` (see `kernel/riscv.h`. These number are helpful for `backtrace` to terminate its loop.

Once your backtrace is working, call it from `panic` in `kernel/printf.c` so that you see the kernel's backtrace when it panics.

Alarm (hard)

In this exercise you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers; you could use something similar to handle page faults in the application, for example. Your solution is correct if it passes alarmtest and usertests.

You should add a new `sigalarm(interval, handler)` system call. If an application calls `sigalarm(n, fn)`, then after every `n` "ticks" of CPU time that the program consumes, the kernel should cause application function `fn` to be called. When `fn` returns, the application should resume where it left off. A tick is a fairly arbitrary unit of time in xv6, determined by how often a hardware timer generates interrupts. If an application calls `sigalarm(0, 0)`, the kernel should stop generating periodic alarm calls.

You'll find a file `user/alarmtest.c` in your xv6 repository. Add it to the Makefile. It won't compile correctly until you've added `sigalarm` and `sigreturn` system calls (see below).

`alarmtest` calls `sigalarm(2, periodic)` in `test0` to ask the kernel to force a call to `periodic()` every 2 ticks, and then spins for a while. You can see the assembly code for alarmtest in user/alarmtest.asm, which may be handy for debugging. Your solution is correct when `alarmtest` produces output like this and usertests also runs correctly:

```
$ alarmtest
test0 start
........alarm!
test0 passed
test1 start
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
test1 passed
test2 start
................alarm!
```

```
test2 passed
$ usertests
...
ALL TESTS PASSED
$
```

When you're done, your solution will be only a few lines of code, but it may be tricky to get it right. We'll test your code with the version of alarmtest.c in the original repository. You can modify alarmtest.c to help you debug, but make sure the original alarmtest says that all the tests pass.

test0: invoke handler

Get started by modifying the kernel to jump to the alarm handler in user space, which will cause test0 to print "alarm!". Don't worry yet what happens after the "alarm!" output; it's OK for now if your program crashes after printing "alarm!". Here are some hints:

- You'll need to modify the Makefile to cause `alarmtest.c` to be compiled as an xv6 user program.
- The right declarations to put in

  user/user.h

  are:

  ```
  int sigalarm(int ticks, void (*handler)());
  int sigreturn(void);
  ```

- Update user/usys.pl (which generates user/usys.S), kernel/syscall.h, and kernel/syscall.c to allow `alarmtest` to invoke the sigalarm and sigreturn system calls.
- For now, your `sys_sigreturn` should just return zero.
- Your `sys_sigalarm()` should store the alarm interval and the pointer to the handler function in new fields in the `proc` structure (in `kernel/proc.h`).
- You'll need to keep track of how many ticks have passed since the last call (or are left until the next call) to a process's alarm handler; you'll need a new field in `struct proc` for this too. You can initialize `proc` fields in `allocproc()` in `proc.c`.
- Every tick, the hardware clock forces an interrupt, which is handled in `usertrap()` in `kernel/trap.c`.
- You only want to manipulate a process's alarm ticks if there's a timer interrupt; you want something like

  ```
  if(which_dev == 2) ...
  ```

- Only invoke the alarm function if the process has a timer outstanding. Note that the address of the user's alarm function might be 0 (e.g., in user/alarmtest.asm, `periodic` is at address 0).

- You'll need to modify `usertrap()` so that when a process's alarm interval expires, the user process executes the handler function. When a trap on the RISC-V returns to user space, what determines the instruction address at which user-space code resumes execution?
- It will be easier to look at traps with gdb if you tell qemu to use only one CPU, which you can do by running

  ```
  make CPUS=1 qemu-gdb
  ```

- You've succeeded if alarmtest prints "alarm!".

test1/test2(): resume interrupted code

Chances are that alarmtest crashes in test0 or test1 after it prints "alarm!", or that alarmtest (eventually) prints "test1 failed", or that alarmtest exits without printing "test1 passed". To fix this, you must ensure that, when the alarm handler is done, control returns to the instruction at which the user program was originally interrupted by the timer interrupt. You must ensure that the register contents are restored to the values they held at the time of the interrupt, so that the user program can continue undisturbed after the alarm. Finally, you should "re-arm" the alarm counter after each time it goes off, so that the handler is called periodically.

As a starting point, we've made a design decision for you: user alarm handlers are required to call the `sigreturn` system call when they have finished. Have a look at `periodic` in `alarmtest.c` for an example. This means that you can add code to `usertrap` and `sys_sigreturn` that cooperate to cause the user process to resume properly after it has handled the alarm.

Some hints:

- Your solution will require you to save and restore registers---what registers do you need to save and restore to resume the interrupted code correctly? (Hint: it will be many).
- Have `usertrap` save enough state in `struct proc` when the timer goes off that `sigreturn` can correctly return to the interrupted user code.
- Prevent re-entrant calls to the handler----if a handler hasn't returned yet, the kernel shouldn't call it again. `test2` tests this.

Once you pass `test0`, `test1`, and `test2` run `usertests` to make sure you didn't break any other parts of the kernel.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Print the names of the functions and line numbers in `backtrace()` instead of numerical addresses (hard).

Lab: xv6 lazy page allocation

One of the many neat tricks an O/S can play with page table hardware is lazy allocation of user-space heap memory. Xv6 applications ask the kernel for heap memory using the sbrk() system call. In the kernel we've given you, sbrk() allocates physical memory and maps it into the process's virtual address space. It can take a long time for a kernel to allocate and map memory for a large request. Consider, for example, that a gigabyte consists of 262,144 4096-byte pages; that's a huge number of allocations even if each is individually cheap. In addition, some programs allocate more memory than they actually use (e.g., to implement sparse arrays), or allocate memory well in advance of use. To allow sbrk() to complete more quickly in these cases, sophisticated kernels allocate user memory lazily. That is, sbrk() doesn't allocate physical memory, but just remembers which user addresses are allocated and marks those addresses as invalid in the user page table. When the process first tries to use any given page of lazily-allocated memory, the CPU generates a page fault, which the kernel handles by allocating physical memory, zeroing it, and mapping it. You'll add this lazy allocation feature to xv6 in this lab.

Before you start coding, read Chapter 4 (in particular 4.6) of the xv6 book, and related files you are likely to modify:

- `kernel/trap.c`
- `kernel/vm.c`
- `kernel/sysproc.c`

To start the lab, switch to the lazy branch:

```
$ git fetch
$ git checkout lazy
$ make clean
```

Eliminate allocation from sbrk() (easy)

Your first task is to delete page allocation from the sbrk(n) system call implementation, which is the function sys_sbrk() in sysproc.c. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new sbrk(n) should just increment the process's size (myproc()->sz) by n and return the old size. It should not allocate memory -- so you should delete the call to growproc() (but you still need to increase the process's size!).

Try to guess what the result of this modification will be: what will break?

Make this modification, boot xv6, and type `echo hi` to the shell. You should see something like this:

```
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000001258 stval=0x0000000000004008
va=0x0000000000004000 pte=0x0000000000000000
panic: uvmunmap: not mapped
```

The "usertrap(): ..." message is from the user trap handler in trap.c; it has caught an exception that it does not know how to handle. Make sure you understand why this page fault occurs. The "stval=0x0..04008" indicates that the virtual address that caused the page fault is 0x4008.

Lazy allocation (moderate)

Modify the code in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. You should add your code just before the `printf` call that produced the "usertrap(): ..." message. Modify whatever other xv6 kernel code you need to in order to get `echo hi` to work.

Here are some hints:

- You can check whether a fault is a page fault by seeing if r_scause() is 13 or 15 in usertrap().
- `r_stval()` returns the RISC-V `stval` register, which contains the virtual address that caused the page fault.
- Steal code from uvmalloc() in vm.c, which is what sbrk() calls (via growproc()). You'll need to call kalloc() and mappages().
- Use PGROUNDDOWN(va) to round the faulting virtual address down to a page boundary.
- uvmunmap() will panic; modify it to not panic if some pages aren't mapped.
- If the kernel crashes, look up sepc in kernel/kernel.asm
- Use your `vmprint` function from pgtbl lab to print the content of a page table.
- If you see the error "incomplete type proc", include "spinlock.h" then "proc.h".

If all goes well, your lazy allocation code should result in `echo hi` working. You should get at least one page fault (and thus lazy allocation), and perhaps two.

Lazytests and Usertests (moderate)

We've supplied you with `lazytests`, an xv6 user program that tests some specific situations that may stress your lazy memory allocator. Modify your kernel code so that all of both `lazytests` and `usertests` pass.

- Handle negative sbrk() arguments.
- Kill a process if it page-faults on a virtual memory address higher than any allocated with sbrk().
- Handle the parent-to-child memory copy in fork() correctly.
- Handle the case in which a process passes a valid address from sbrk() to a system call such as read or write, but the memory for that address has not yet been allocated.
- Handle out-of-memory correctly: if kalloc() fails in the page fault handler, kill the current process.
- Handle faults on the invalid page below the user stack.

Your solution is acceptable if your kernel passes lazytests and usertests:

```
$  lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap...
usertrap(): ...
test lazy unmap: OK
running test out of memory
usertrap(): ...
test out of memory: OK
ALL TESTS PASSED
```

```
$ usertests
...
ALL TESTS PASSED
$
```

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Make lazy page allocation work with your simple `copyin` from the previous lab.

Lab: Copy-on-Write Fork for xv6

Virtual memory provides a level of indirection: the kernel can intercept memory references by marking PTEs invalid or read-only, leading to page faults, and can change what addresses mean by modifying PTEs. There is a saying in computer systems that any systems problem can be solved with a level of indirection. The lazy allocation lab provided one example. This lab explores another example: copy-on write fork.

To start the lab, switch to the cow branch:

```
$ git fetch
$ git checkout cow
$ make clean
```

The problem

The fork() system call in xv6 copies all of the parent process's user-space memory into the child. If the parent is large, copying can take a long time. Worse, the work is often largely wasted; for example, a fork() followed by exec() in the child will cause the child to discard the copied memory, probably without ever using most of it. On the other hand, if both parent and child use a page, and one or both writes it, a copy is truly needed.

The solution

The goal of copy-on-write (COW) fork() is to defer allocating and copying physical memory pages for the child until the copies are actually needed, if ever.

COW fork() creates just a pagetable for the child, with PTEs for user memory pointing to the parent's physical pages. COW fork() marks all the user PTEs in both parent and child as not writable. When either process tries to write one of these COW pages, the CPU will force a page fault. The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable. When the page fault handler returns, the user process will be able to write its copy of the page.

COW fork() makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes' page tables, and should be freed only when the last reference disappears.

Implement copy-on write(hard)

Your task is to implement copy-on-write fork in the xv6 kernel. You are done if your modified kernel executes both the cowtest and usertests programs successfully.

To help you test your implementation, we've provided an xv6 program called cowtest (source in user/cowtest.c). cowtest runs various tests, but even the first will fail on unmodified xv6. Thus, initially, you will see:

```
$ cowtest
simple: fork() failed
$
```

The "simple" test allocates more than half of available physical memory, and then fork()s. The fork fails because there is not enough free physical memory to give the child a complete copy of the parent's memory.

When you are done, your kernel should pass all the tests in both cowtest and usertests. That is:

```
$ cowtest
simple: ok
simple: ok
three: zombie!
ok
three: zombie!
ok
three: zombie!
ok
file: ok
ALL COW TESTS PASSED
$ usertests
...
ALL TESTS PASSED
$
```

Here's a reasonable plan of attack.

1. Modify uvmcopy() to map the parent's physical pages into the child, instead of allocating new pages. Clear `PTE_W` in the PTEs of both child and parent.
2. Modify usertrap() to recognize page faults. When a page-fault occurs on a COW page, allocate a new page with kalloc(), copy the old page to the new page, and install the new page in the PTE with `PTE_W` set.
3. Ensure that each physical page is freed when the last PTE reference to it goes away -- but not before. A good way to do this is to keep, for each physical page, a "reference count" of the number of user page tables that refer to that page. Set a page's reference count to one when `kalloc()` allocates it. Increment a page's reference count when fork causes a child to share the page, and decrement a page's count each time any process drops the page from its page table. `kfree()` should only place a page back on the free list if its reference count is zero. It's OK to to keep these counts in a fixed-size array of integers. You'll have to work out a scheme for how to index the array and how to choose its size. For example, you could index the array

with the page's physical address divided by 4096, and give the array a number of elements equal to highest physical address of any page placed on the free list by `kinit()` in kalloc.c.
4. Modify copyout() to use the same scheme as page faults when it encounters a COW page.

Some hints:

- The lazy page allocation lab has likely made you familiar with much of the xv6 kernel code that's relevant for copy-on-write. However, you should not base this lab on your lazy allocation solution; instead, please start with a fresh copy of xv6 as directed above.
- It may be useful to have a way to record, for each PTE, whether it is a COW mapping. You can use the RSW (reserved for software) bits in the RISC-V PTE for this.
- `usertests` explores scenarios that `cowtest` does not test, so don't forget to check that all tests pass for both.
- Some helpful macros and definitions for page table flags are at the end of `kernel/riscv.h`.
- If a COW page fault occurs and there's no free memory, the process should be killed.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the [submission website](). You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Modify xv6 to support both lazy page allocation and COW.
- Measure how much your COW implementation reduces the number of bytes xv6 copies and the number of physical pages it allocates. Find and exploit opportunities to further reduce those numbers.

Lab: Multithreading

This lab will familiarize you with multithreading. You will implement switching between threads in a user-level threads package, use multiple threads to speed up a program, and implement a barrier.

Before writing code, you should make sure you have read "Chapter 7: Scheduling" from the xv6 book and studied the corresponding code.

To start the lab, switch to the thread branch:

```
$ git fetch
$ git checkout thread
$ make clean
```

Uthread: switching between threads (moderate)

In this exercise you will design the context switch mechanism for a user-level threading system, and then implement it. To get you started, your xv6 has two files user/uthread.c and user/uthread_switch.S, and a rule in the Makefile to build a uthread program. uthread.c contains most of a user-level threading package, and code for three simple test threads. The threading package is missing some of the code to create a thread and to switch between threads.

Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, `make grade` should say that your solution passes the `uthread` test.

Once you've finished, you should see the following output when you run `uthread` on xv6 (the three threads might start in a different order):

```
$ make qemu
...
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
...
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

This output comes from the three test threads, each of which has a loop that prints a line and then yields the CPU to the other threads.

At this point, however, with no context switch code, you'll see no output.

You will need to add code to `thread_create()` and `thread_schedule()` in `user/uthread.c`, and `thread_switch` in `user/uthread_switch.S`. One goal is ensure that when `thread_schedule()` runs a given thread for the first time, the thread executes the function passed to `thread_create()`, on its own stack. Another goal is to ensure that `thread_switch` saves the registers of the thread being switched away from, restores the registers of the thread being switched to, and returns to the point in the latter thread's instructions where it last left off. You will have to decide where to save/restore registers; modifying `struct thread` to hold registers is a good plan. You'll need to add a call to `thread_switch` in `thread_schedule`; you can pass whatever arguments you need to `thread_switch`, but the intent is to switch from thread `t` to `next_thread`.

Some hints:

- `thread_switch` needs to save/restore only the callee-save registers. Why?
- You can see the assembly code for uthread in user/uthread.asm, which may be handy for debugging.

- To test your code it might be helpful to single step through your

  thread_switch

  using

  riscv64-linux-gnu-gdb

  . You can get started in this way:

  ```
  (gdb) file user/_uthread
  Reading symbols from user/_uthread...
  (gdb) b uthread.c:60
  ```

  This sets a breakpoint at line 60 of `uthread.c`. The breakpoint may (or may not) be triggered before you even run `uthread`. How could that happen?

  Once your xv6 shell runs, type "uthread", and gdb will break at line 60. Now you can type commands like the following to inspect the state of `uthread`:

  ```
  (gdb) p/x *next_thread
  ```

  With "x", you can examine the content of a memory location:

  ```
  (gdb) x/x next_thread->stack
  ```

  You can skip to the start of `thread_switch` thus:

  ```
  (gdb) b thread_switch
  (gdb) c
  ```

  You can single step assembly instructions using:

  ```
  (gdb) si
  ```

  On-line documentation for gdb is here.

Using threads (moderate)

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this assignment on a real Linux or MacOS computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors.

This assignment uses the UNIX `pthread` threading library. You can find information about it from the manual page, with `man pthreads`, and you can look on the web, for example here, here, and here.

The file `notxv6/ph.c` contains a simple hash table that is correct if used from a single thread, but incorrect when used from multiple threads. In your main xv6 directory (perhaps `~/xv6-labs-2020`), type this:

```
$ make ph
$ ./ph 1
```

Note that to build `ph` the Makefile uses your OS's gcc, not the 6.S081 tools. The argument to `ph` specifies the number of threads that execute put and get operations on the the hash table. After running for a little while, `ph 1` will produce output similar to this:

```
100000 puts, 3.991 seconds, 25056 puts/second
0: 0 keys missing
100000 gets, 3.981 seconds, 25118 gets/second
```

The numbers you see may differ from this sample output by a factor of two or more, depending on how fast your computer is, whether it has multiple cores, and whether it's busy doing other things.

`ph` runs two benchmarks. First it adds lots of keys to the hash table by calling `put()`, and prints the achieved rate in puts per second. The it fetches keys from the hash table with `get()`. It prints the number keys that should have been in the hash table as a result of the puts but are missing (zero in this case), and it prints the number of gets per second it achieved.

You can tell `ph` to use its hash table from multiple threads at the same time by giving it an argument greater than one. Try `ph 2`:

```
$ ./ph 2
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 4.322 seconds, 46274 gets/second
```

The first line of this `ph 2` output indicates that when two threads concurrently add entries to the hash table, they achieve a total rate of 53,044 inserts per second. That's about twice the rate of the single thread from running `ph 1`. That's an excellent "parallel speedup" of about 2x, as much as one could possibly hope for (i.e. twice as many cores yielding twice as much work per unit time).

However, the two lines saying `16579 keys missing` indicate that a large number of keys that should have been in the hash table are not there. That is, the puts were supposed to add those keys to the hash table, but something went wrong. Have a look at `notxv6/ph.c`, particularly at `put()` and `insert()`.

Why are there missing keys with 2 threads, but not with 1 thread? Identify a sequence of events with 2 threads that can lead to a key being missing. Submit your sequence with a short explanation in answers-thread.txt

To avoid this sequence of events, insert lock and unlock statements in `put` and `get` in `notxv6/ph.c` so that the number of keys missing is always 0 with two threads. The relevant pthread calls are:

```
pthread_mutex_t lock;            // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);       // acquire lock
pthread_mutex_unlock(&lock);     // release lock
```

You're done when `make grade` says that your code passes the `ph_safe` test, which requires zero missing keys with two threads. It's OK at this point to fail the `ph_fast` test.

Don't forget to call `pthread_mutex_init()`. Test your code first with 1 thread, then test it with 2 threads. Is it correct (i.e. have you eliminated missing keys?)? Does the two-threaded version achieve parallel speedup (i.e. more total work per unit time) relative to the single-threaded version?

There are situations where concurrent `put()`s have no overlap in the memory they read or write in the hash table, and thus don't need a lock to protect against each other. Can you change `ph.c` to take advantage of such situations to obtain parallel speedup for some `put()`s? Hint: how about a lock per hash bucket?

Modify your code so that some `put` operations run in parallel while maintaining correctness. You're done when `make grade` says your code passes both the `ph_safe` and `ph_fast` tests. The `ph_fast` test requires that two threads yield at least 1.25 times as many puts/second as one thread.

Barrier(moderate)

In this assignment you'll implement a barrier: a point in an application at which all participating threads must wait until all other participating threads reach that point too. You'll use pthread condition variables, which are a sequence coordination technique similar to xv6's sleep and wakeup.

You should do this assignment on a real computer (not xv6, not qemu).

The file `notxv6/barrier.c` contains a broken barrier.

```
$ make barrier
$ ./barrier 2
barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.
```

The 2 specifies the number of threads that synchronize on the barrier (`nthread` in `barrier.c`). Each thread executes a loop. In each loop iteration a thread calls `barrier()` and then sleeps for a random number of microseconds. The assert triggers, because one thread leaves the barrier before the other thread has reached the barrier. The desired behavior is that each thread blocks in `barrier()` until all `nthreads` of them have called `barrier()`.

Your goal is to achieve the desired barrier behavior. In addition to the lock primitives that you have seen in the `ph` assignment, you will need the following new pthread primitives; look here and here for details.

```
pthread_cond_wait(&cond, &mutex);  // go to sleep on cond, releasing lock mutex,
acquiring upon wake up
pthread_cond_broadcast(&cond);     // wake up every thread sleeping on cond
```

Make sure your solution passes `make grade`'s `barrier` test.

`pthread_cond_wait` releases the `mutex` when called, and re-acquires the `mutex` before returning.

We have given you `barrier_init()`. Your job is to implement `barrier()` so that the panic doesn't occur. We've defined `struct barrier` for you; its fields are for your use.

There are two issues that complicate your task:

- You have to deal with a succession of barrier calls, each of which we'll call a round. `bstate.round` records the current round. You should increment `bstate.round` each time all threads have reached the barrier.
- You have to handle the case in which one thread races around the loop before the others have exited the barrier. In particular, you are re-using the `bstate.nthread` variable from one round to the next. Make sure that a thread that leaves the barrier and races around the loop doesn't increase `bstate.nthread` while a previous round is still using it.

Test your code with one, two, and more than two threads.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenges for uthread

The user-level thread package interacts badly with the operating system in several ways. For example, if one user-level thread blocks in a system call, another user-level thread won't run, because the user-level threads scheduler doesn't know that one of its threads has been descheduled by the xv6 scheduler. As another example, two user-level threads will not run concurrently on different cores, because the xv6 scheduler isn't aware that there are multiple threads that could run in parallel. Note that if two user-level threads were to run truly in parallel, this implementation won't work because of several races (e.g., two threads on different processors could call `thread_schedule` concurrently, select the same runnable thread, and both run it on different processors.)

There are several ways of addressing these problems. One is using scheduler activations and another is to use one kernel thread per user-level thread (as Linux kernels do). Implement one of these ways in xv6. This is not easy to get right; for example, you will need to implement TLB shootdown when updating a page table for a multithreaded user process.

Add locks, condition variables, barriers, etc. to your thread package.

Lab: locks

In this lab you'll gain experience in re-designing code to increase parallelism. A common symptom of poor parallelism on multi-core machines is high lock contention. Improving parallelism often involves changing both data structures and locking strategies in order to reduce contention. You'll do this for the xv6 memory allocator and block cache.

Before writing code, make sure to read the following parts from the xv6 book :

- Chapter 6: "Locking" and the corresponding code.
- Section 3.5: "Code: Physical memory allocator"
- Section 8.1 through 8.3: "Overview", "Buffer cache layer", and "Code: Buffer cache"

```
$ git fetch
$ git checkout lock
$ make clean
```

Memory allocator (moderate)

The program user/kalloctest stresses xv6's memory allocator: three processes grow and shrink their address spaces, resulting in many calls to `kalloc` and `kfree`. `kalloc` and `kfree` obtain `kmem.lock`. kalloctest prints (as "#fetch-and-add") the number of loop iterations in `acquire` due to attempts to acquire a lock that another core already holds, for the `kmem` lock and a few other locks. The number of loop iterations in `acquire` is a rough measure of lock contention. The output of `kalloctest` looks similar to this before you complete the lab:

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: bcache: #fetch-and-add 0 #acquire() 1260
--- top 5 contended locks:
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: proc: #fetch-and-add 23737 #acquire() 130718
lock: virtio_disk: #fetch-and-add 11159 #acquire() 114
lock: proc: #fetch-and-add 5937 #acquire() 130786
lock: proc: #fetch-and-add 4080 #acquire() 130786
tot= 83375
test1 FAIL
```

`acquire` maintains, for each lock, the count of calls to `acquire` for that lock, and the number of times the loop in `acquire` tried but failed to set the lock. kalloctest calls a system call that causes the kernel to print those counts for the kmem and bcache locks (which are the focus of this lab) and for the 5 most contended locks. If there is lock contention the number of `acquire` loop iterations will be large. The system call returns the sum of the number of loop iterations for the kmem and bcache locks.

For this lab, you must use a dedicated unloaded machine with multiple cores. If you use a machine that is doing other things, the counts that kalloctest prints will be nonsense. You can use a dedicated Athena workstation, or your own laptop, but don't use a dialup machine.

The root cause of lock contention in kalloctest is that `kalloc()` has a single free list, protected by a single lock. To remove lock contention, you will have to redesign the memory allocator to avoid a single lock and list. The basic idea is to maintain a free list per CPU, each list with its own lock. Allocations and frees on different CPUs can run in parallel, because each CPU will operate on a different list. The main challenge will be to deal with the case in which one CPU's free list is empty, but another CPU's list has free memory; in that case, the one CPU must "steal" part of the other CPU's free list. Stealing may introduce lock contention, but that will hopefully be infrequent.

Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run kalloctest to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests` pass. `make grade` should say that the kalloctests pass.

```
$ kalloctest
start test1
test1 results:
```

```
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 42843
lock: kmem: #fetch-and-add 0 #acquire() 198674
lock: kmem: #fetch-and-add 0 #acquire() 191534
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #fetch-and-add 43861 #acquire() 117281
lock: virtio_disk: #fetch-and-add 5347 #acquire() 114
lock: proc: #fetch-and-add 4856 #acquire() 117312
lock: proc: #fetch-and-add 4168 #acquire() 117316
lock: proc: #fetch-and-add 2797 #acquire() 117266
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ usertests
...
ALL TESTS PASSED
$
```

Some hints:

- You can use the constant `NCPU` from kernel/param.h
- Let `freerange` give all free memory to the CPU running `freerange`.
- The function `cpuid` returns the current core number, but it's only safe to call it and use its result when interrupts are turned off. You should use `push_off()` and `pop_off()` to turn interrupts off and on.
- Have a look at the `snprintf` function in kernel/sprintf.c for string formatting ideas. It is OK to just name all locks "kmem" though.

Buffer cache (hard)

This half of the assignment is independent from the first half; you can work on this half (and pass the tests) whether or not you have completed the first half.

If multiple processes use the file system intensively, they will likely contend for `bcache.lock`, which protects the disk block cache in kernel/bio.c. `bcachetest` creates several processes that repeatedly read different files in order to generate contention on `bcache.lock`; its output looks like this (before you complete this lab):

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33035
lock: bcache: #fetch-and-add 16142 #acquire() 65978
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 162870 #acquire() 1188
```

```
lock: proc: #fetch-and-add 51936 #acquire() 73732
lock: bcache: #fetch-and-add 16142 #acquire() 65978
lock: uart: #fetch-and-add 7505 #acquire() 117
lock: proc: #fetch-and-add 6937 #acquire() 73420
tot= 16142
test0: FAIL
start test1
test1 OK
```

You will likely see different output, but the number of `acquire` loop iterations for the `bcache` lock will be high. If you look at the code in `kernel/bio.c`, you'll see that `bcache.lock` protects the list of cached block buffers, the reference count (`b->refcnt`) in each block buffer, and the identities of the cached blocks (`b->dev` and `b->blockno`).

Modify the block cache so that the number of `acquire` loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget` and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure usertests still passes. `make grade` should pass all tests when you are done.

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32954
lock: kmem: #fetch-and-add 0 #acquire() 75
lock: kmem: #fetch-and-add 0 #acquire() 73
lock: bcache: #fetch-and-add 0 #acquire() 85
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4159
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2118
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4274
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4326
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6334
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6321
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6704
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6696
lock: bcache.bucket: #fetch-and-add 0 #acquire() 7757
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6199
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4136
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4136
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2123
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 158235 #acquire() 1193
lock: proc: #fetch-and-add 117563 #acquire() 3708493
lock: proc: #fetch-and-add 65921 #acquire() 3710254
lock: proc: #fetch-and-add 44090 #acquire() 3708607
lock: proc: #fetch-and-add 43252 #acquire() 3708521
tot= 128
test0: OK
```

```
start test1
test1 OK
$ usertests
  ...
ALL TESTS PASSED
$
```

Please give all of your locks names that start with "bcache". That is, you should call `initlock` for each of your locks, and pass a name that starts with "bcache".

Reducing contention in the block cache is more tricky than for kalloc, because bcache buffers are truly shared among processes (and thus CPUs). For kalloc, one could eliminate most contention by giving each CPU its own allocator; that won't work for the block cache. We suggest you look up block numbers in the cache with a hash table that has a lock per hash bucket.

There are some circumstances in which it's OK if your solution has lock conflicts:

- When two processes concurrently use the same block number. `bcachetest` `test0` doesn't ever do this.
- When two processes concurrently miss in the cache, and need to find an unused block to replace. `bcachetest` `test0` doesn't ever do this.
- When two processes concurrently use blocks that conflict in whatever scheme you use to partition the blocks and locks; for example, if two processes use blocks whose block numbers hash to the same slot in a hash table. `bcachetest` `test0` might do this, depending on your design, but you should try to adjust your scheme's details to avoid conflicts (e.g., change the size of your hash table).

`bcachetest`'s `test1` uses more distinct blocks than there are buffers, and exercises lots of file system code paths.

Here are some hints:

- Read the description of the block cache in the xv6 book (Section 8.1-8.3).
- It is OK to use a fixed number of buckets and not resize the hash table dynamically. Use a prime number of buckets (e.g., 13) to reduce the likelihood of hashing conflicts.
- Searching in the hash table for a buffer and allocating an entry for that buffer when the buffer is not found must be atomic.
- Remove the list of all buffers (`bcache.head` etc.) and instead time-stamp buffers using the time of their last use (i.e., using `ticks` in kernel/trap.c). With this change `brelse` doesn't need to acquire the bcache lock, and `bget` can select the least-recently used block based on the time-stamps.
- It is OK to serialize eviction in `bget` (i.e., the part of `bget` that selects a buffer to re-use when a lookup misses in the cache).
- Your solution might need to hold two locks in some cases; for example, during eviction you may need to hold the bcache lock and a lock per bucket. Make sure you avoid deadlock.
- When replacing a block, you might move a `struct buf` from one bucket to another bucket, because the new block hashes to a different bucket. You might have a tricky case: the new block might hash to the same bucket as the old block. Make sure you avoid deadlock in that case.
- Some debugging tips: implement bucket locks but leave the global bcache.lock acquire/release at the beginning/end of bget to serialize the code. Once you are sure it is correct without race conditions, remove the global locks and deal with concurrency issues. You can also run `make CPUS=1 qemu` to test with one core.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- make lookup in the buffer cache lock-free. Hint: use gcc's `__sync_*` functions. How do you convince yourself that your implementation is correct?

Lab: file system

In this lab you will add large files and symbolic links to the xv6 file system.

Before writing code, you should read "Chapter 8: File system" from the xv6 book and study the corresponding code.

Fetch the xv6 source for the lab and check out the `util` branch:

```
  $ git fetch
  $ git checkout fs
  $ make clean
```

Large files (moderate)

In this assignment you'll increase the maximum size of an xv6 file. Currently xv6 files are limited to 268 blocks, or 268*BSIZE bytes (BSIZE is 1024 in xv6). This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of 12+256=268 blocks.

The `bigfile` command creates the longest file it can, and reports that size:

```
$ bigfile
..
wrote 268 blocks
bigfile: file is too small
$
```

The test fails because `bigfile` expects to be able to create a file with 65803 blocks, but unmodified xv6 limits files to 268 blocks.

You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 256 addresses of singly-indirect blocks, each of which can contain up to 256 addresses of data blocks. The result will be that a file will be able to consist of up to 65803 blocks, or 256*256+256+11 blocks (11 instead of 12, because we will sacrifice one of the direct block numbers for the double-indirect block).

Preliminaries

The `mkfs` program creates the xv6 file system disk image and determines how many total blocks the file system has; this size is controlled by `FSSIZE` in `kernel/param.h`. You'll see that `FSSIZE` in the repository for this lab is set to 200,000 blocks. You should see the following output from `mkfs/mkfs` in the make output:

```
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 25) blocks
199930 total 200000
```

This line describes the file system that `mkfs/mkfs` built: it has 70 meta-data blocks (blocks used to describe the file system) and 199,930 data blocks, totaling 200,000 blocks.
If at any point during the lab you find yourself having to rebuild the file system from scratch, you can run `make clean` which forces make to rebuild fs.img.

What to Look At

The format of an on-disk inode is defined by `struct dinode` in `fs.h`. You're particularly interested in `NDIRECT`, `NINDIRECT`, `MAXFILE`, and the `addrs[]` element of `struct dinode`. Look at Figure 8.3 in the xv6 text for a diagram of the standard xv6 inode.

The code that finds a file's data on disk is in `bmap()` in `fs.c`. Have a look at it and make sure you understand what it's doing. `bmap()` is called both when reading and writing a file. When writing, `bmap()` allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

`bmap()` deals with two kinds of block numbers. The `bn` argument is a "logical block number" -- a block number within the file, relative to the start of the file. The block numbers in `ip->addrs[]`, and the argument to `bread()`, are disk block numbers. You can view `bmap()` as mapping a file's logical block numbers into disk block numbers.

Your Job

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block. You are done with this exercise when `bigfile` writes 65803 blocks and `usertests` runs successfully:

```
$ bigfile
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
...............
wrote 65803 blocks
done; ok
$ usertests
...
ALL TESTS PASSED
$
```

`bigfile` will take at least a minute and a half to run.

Hints:

- Make sure you understand `bmap()`. Write out a diagram of the relationships between `ip->addrs[]`, the indirect block, the doubly-indirect block and the singly-indirect blocks it points to, and data blocks. Make sure you understand why adding a doubly-indirect block increases the maximum file size by 256*256 blocks (really -1, since you have to decrease the number of direct blocks by one).
- Think about how you'll index the doubly-indirect block, and the indirect blocks it points to, with the logical block number.
- If you change the definition of `NDIRECT`, you'll probably have to change the declaration of `addrs[]` in `struct inode` in `file.h`. Make sure that `struct inode` and `struct dinode` have the same number of elements in their `addrs[]` arrays.
- If you change the definition of `NDIRECT`, make sure to create a new `fs.img`, since `mkfs` uses `NDIRECT` to build the file system.
- If your file system gets into a bad state, perhaps by crashing, delete `fs.img` (do this from Unix, not xv6). `make` will build a new clean file system image for you.
- Don't forget to `brelse()` each block that you `bread()`.
- You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.
- Make sure `itrunc` frees all blocks of a file, including double-indirect blocks.

Symbolic links (moderate)

In this exercise you will add symbolic links to xv6. Symbolic links (or soft links) refer to a linked file by pathname; when a symbolic link is opened, the kernel follows the link to the referred file. Symbolic links resembles hard links, but hard links are restricted to pointing to file on the same disk, while symbolic links can cross disk devices. Although xv6 doesn't support multiple devices, implementing this system call is a good exercise to understand how pathname lookup works.

Your job

You will implement the `symlink(char *target, char *path)` system call, which creates a new symbolic link at path that refers to file named by target. For further information, see the man page symlink. To test, add symlinktest to the Makefile and run it. Your solution is complete when the tests produce the following output (including usertests succeeding).

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ usertests
...
ALL TESTS PASSED
$
```

Hints:

- First, create a new system call number for symlink, add an entry to user/usys.pl, user/user.h, and implement an empty sys_symlink in kernel/sysfile.c.
- Add a new file type (`T_SYMLINK`) to kernel/stat.h to represent a symbolic link.
- Add a new flag to kernel/fcntl.h, (`O_NOFOLLOW`), that can be used with the `open` system call. Note that flags passed to `open` are combined using a bitwise OR operator, so your new flag

should not overlap with any existing flags. This will let you compile user/symlinktest.c once you add it to the Makefile.

- Implement the `symlink(target, path)` system call to create a new symbolic link at path that refers to target. Note that target does not need to exist for the system call to succeed. You will need to choose somewhere to store the target path of a symbolic link, for example, in the inode's data blocks. `symlink` should return an integer representing success (0) or failure (-1) similar to `link` and `unlink`.
- Modify the `open` system call to handle the case where the path refers to a symbolic link. If the file does not exist, `open` must fail. When a process specifies `O_NOFOLLOW` in the flags to `open`, `open` should open the symlink (and not follow the symbolic link).
- If the linked file is also a symbolic link, you must recursively follow it until a non-link file is reached. If the links form a cycle, you must return an error code. You may approximate this by returning an error code if the depth of links reaches some threshold (e.g., 10).
- Other system calls (e.g., link and unlink) must not follow symbolic links; these system calls operate on the symbolic link itself.
- You do not have to handle symbolic links to directories for this lab.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

Support triple-indirect blocks.

Acknowledgment

Thanks to the staff of UW's CSEP551 (Fall 2019) for the symlink exercise.

Lab: mmap (hard)

The `mmap` and `munmap` system calls allow UNIX programs to exert detailed control over their address spaces. They can be used to share memory among processes, to map files into process address spaces, and as part of user-level page fault schemes such as the garbage-collection algorithms discussed in lecture. In this lab you'll add `mmap` and `munmap` to xv6, focusing on memory-mapped files.

Fetch the xv6 source for the lab and check out the `mmap` branch:

```
  $ git fetch
  $ git checkout mmap
  $ make clean
```

The manual page (run man 2 mmap) shows this declaration for `mmap`:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`mmap` can be called in many ways, but this lab requires only a subset of its features relevant to memory-mapping a file. You can assume that `addr` will always be zero, meaning that the kernel should decide the virtual address at which to map the file. `mmap` returns that address, or 0xffffffffffffffff if it fails. `length` is the number of bytes to map; it might not be the same as the file's length. `prot` indicates whether the memory should be mapped readable, writeable, and/or executable; you can assume that `prot` is `PROT_READ` or `PROT_WRITE` or both. `flags` will be either `MAP_SHARED`, meaning that modifications to the mapped memory should be written back to the file, or `MAP_PRIVATE`, meaning that they should not. You don't have to implement any other bits in `flags`. `fd` is the open file descriptor of the file to map. You can assume `offset` is zero (it's the starting point in the file at which to map).

It's OK if processes that map the same `MAP_SHARED` file do **not** share physical pages.

`munmap(addr, length)` should remove mmap mappings in the indicated address range. If the process has modified the memory and has it mapped `MAP_SHARED`, the modifications should first be written to the file. An `munmap` call might cover only a portion of an mmap-ed region, but you can assume that it will either unmap at the start, or at the end, or the whole region (but not punch a hole in the middle of a region).

You should implement enough `mmap` and `munmap` functionality to make the `mmaptest` test program work. If `mmaptest` doesn't use a `mmap` feature, you don't need to implement that feature.

When you're done, you should see this output:

```
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
$ usertests
usertests starting
...
ALL TESTS PASSED
$
```

Here are some hints:

- Start by adding `_mmaptest` to `UPROGS`, and `mmap` and `munmap` system calls, in order to get `user/mmaptest.c` to compile. For now, just return errors from `mmap` and `munmap`. We defined `PROT_READ` etc for you in `kernel/fcntl.h`. Run `mmaptest`, which will fail at the first mmap call.
- Fill in the page table lazily, in response to page faults. That is, `mmap` should not allocate physical memory or read the file. Instead, do that in page fault handling code in (or called by) `usertrap`, as in the lazy page allocation lab. The reason to be lazy is to ensure that `mmap` of a large file is fast, and that `mmap` of a file larger than physical memory is possible.
- Keep track of what `mmap` has mapped for each process. Define a structure corresponding to the VMA (virtual memory area) described in Lecture 15, recording the address, length, permissions, file, etc. for a virtual memory range created by `mmap`. Since the xv6 kernel doesn't have a memory allocator in the kernel, it's OK to declare a fixed-size array of VMAs and allocate from that array as needed. A size of 16 should be sufficient.
- Implement `mmap`: find an unused region in the process's address space in which to map the file, and add a VMA to the process's table of mapped regions. The VMA should contain a pointer to a `struct file` for the file being mapped; `mmap` should increase the file's reference count so that the structure doesn't disappear when the file is closed (hint: see `filedup`). Run `mmaptest`: the first `mmap` should succeed, but the first access to the mmap-ed memory will cause a page fault and kill `mmaptest`.
- Add code to cause a page-fault in a mmap-ed region to allocate a page of physical memory, read 4096 bytes of the relevant file into that page, and map it into the user address space. Read the file with `readi`, which takes an offset argument at which to read in the file (but you will have to lock/unlock the inode passed to `readi`). Don't forget to set the permissions correctly on the page. Run `mmaptest`; it should get to the first `munmap`.
- Implement `munmap`: find the VMA for the address range and unmap the specified pages (hint: use `uvmunmap`). If `munmap` removes all pages of a previous `mmap`, it should decrement the reference count of the corresponding `struct file`. If an unmapped page has been modified and the file is mapped `MAP_SHARED`, write the page back to the file. Look at `filewrite` for inspiration.
- Ideally your implementation would only write back `MAP_SHARED` pages that the program actually modified. The dirty bit (`D`) in the RISC-V PTE indicates whether a page has been written. However, `mmaptest` does not check that non-dirty pages are not written back; thus you can get away with writing pages back without looking at `D` bits.
- Modify `exit` to unmap the process's mapped regions as if `munmap` had been called. Run `mmaptest`; `mmap_test` should pass, but probably not `fork_test`.
- Modify `fork` to ensure that the child has the same mapped regions as the parent. Don't forget to increment the reference count for a VMA's `struct file`. In the page fault handler of the child, it is OK to allocate a new physical page instead of sharing a page with the parent. The latter would be cooler, but it would require more implementation work. Run `mmaptest`; it should pass both `mmap_test` and `fork_test`.

Run `usertests` to make sure everything still works.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenges

- If two processes have the same file mmap-ed (as in `fork_test`), share their physical pages. You will need reference counts on physical pages.
- Your solution probably allocates a new physical page for each page read from the mmap-ed file, even though the data is also in kernel memory in the buffer cache. Modify your implementation to use that physical memory, instead of allocating a new page. This requires that file blocks be the same size as pages (set `BSIZE` to 4096). You will need to pin mmap-ed blocks into the buffer cache. You will need worry about reference counts.
- Remove redundancy between your implementation for lazy allocation and your implementation of mmap-ed files. (Hint: create a VMA for the lazy allocation area.)
- Modify `exec` to use a VMA for different sections of the binary so that you get on-demand-paged executables. This will make starting programs faster, because `exec` will not have to read any data from the file system.
- Implement page-out and page-in: have the kernel move some parts of processes to disk when physical memory is low. Then, page in the paged-out memory when the process references it.

Lab: networking

In this lab you will write an xv6 device driver for a network interface card (NIC).

Fetch the xv6 source for the lab and check out the `net` branch:

```
$ git fetch
$ git checkout net
$ make clean
```

Background

Before writing code, you may find it helpful to review "Chapter 5: Interrupts and device drivers" in the xv6 book.

You'll use a network device called the E1000 to handle network communication. To xv6 (and the driver you write), the E1000 looks like a real piece of hardware connected to a real Ethernet local area network (LAN). In fact, the E1000 your driver will talk to is an emulation provided by qemu, connected to a LAN that is also emulated by qemu. On this emulated LAN, xv6 (the "guest") has an IP address of 10.0.2.15. Qemu also arranges for the computer running qemu to appear on the LAN with IP address 10.0.2.2. When xv6 uses the E1000 to send a packet to 10.0.2.2, qemu delivers the packet to the appropriate application on the (real) computer on which you're running qemu (the "host").

You will use QEMU's "user-mode network stack". QEMU's documentation has more about the user-mode stack here. We've updated the Makefile to enable QEMU's user-mode network stack and the E1000 network card.

The Makefile configures QEMU to record all incoming and outgoing packets to the file `packets.pcap` in your lab directory. It may be helpful to review these recordings to confirm that xv6 is transmitting and receiving the packets you expect. To display the recorded packets:

```
tcpdump -XXnr packets.pcap
```

We've added some files to the xv6 repository for this lab. The file `kernel/e1000.c` contains initialization code for the E1000 as well as empty functions for transmitting and receiving packets, which you'll fill in. `kernel/e1000_dev.h` contains definitions for registers and flag bits defined by the E1000 and described in the Intel E1000 Software Developer's Manual. `kernel/net.c` and `kernel/net.h` contain a simple network stack that implements the IP, UDP, and ARP protocols. These files also contain code for a flexible data structure to hold packets, called an `mbuf`. Finally, `kernel/pci.c` contains code that searches for an E1000 card on the PCI bus when xv6 boots.

Your Job (hard)

Your job is to complete `e1000_transmit()` and `e1000_recv()`, both in `kernel/e1000.c`, so that the driver can transmit and receive packets. You are done when `make grade` says your solution passes all the tests.

While writing your code, you'll find yourself referring to the E1000 Software Developer's Manual. Of particular help may be the following sections:

- Section 2 is essential and gives an overview of the entire device.
- Section 3.2 gives an overview of packet receiving.
- Section 3.3 gives an overview of packet transmission, alongside section 3.4.
- Section 13 gives an overview of the registers used by the E1000.
- Section 14 may help you understand the init code that we've provided.

Browse the E1000 Software Developer's Manual. This manual covers several closely related Ethernet controllers. QEMU emulates the 82540EM. Skim Chapter 2 now to get a feel for the device. To write your driver, you'll need to be familiar with Chapters 3 and 14, as well as 4.1 (though not 4.1's subsections). You'll also need to use Chapter 13 as a reference. The other chapters mostly cover components of the E1000 that your driver won't have to interact with. Don't worry about the details at first; just get a feel for how the document is structured so you can find things later. The E1000 has many advanced features, most of which you can ignore. Only a small set of basic features is needed to complete this lab.

The `e1000_init()` function we provide you in `e1000.c` configures the E1000 to read packets to be transmitted from RAM, and to write received packets to RAM. This technique is called DMA, for direct memory access, referring to the fact that the E1000 hardware directly writes and reads packets to/from RAM.

Because bursts of packets might arrive faster than the driver can process them, `e1000_init()` provides the E1000 with multiple buffers into which the E1000 can write packets. The E1000 requires these buffers to be described by an array of "descriptors" in RAM; each descriptor contains an address in RAM where the E1000 can write a received packet. `struct rx_desc` describes the descriptor format. The array of descriptors is called the receive ring, or receive queue. It's a circular ring in the sense that when the card or driver reaches the end of the array, it wraps back to the beginning. `e1000_init()` allocates `mbuf` packet buffers for the E1000 to DMA into, using `mbufalloc()`. There is also a transmit ring into which the driver places packets it wants the E1000 to send. `e1000_init()` configures the two rings to have size `RX_RING_SIZE` and `TX_RING_SIZE`.

When the network stack in `net.c` needs to send a packet, it calls `e1000_transmit()` with an mbuf that holds the packet to be sent. Your transmit code must place a pointer to the packet data in a descriptor in the TX (transmit) ring. `struct tx_desc` describes the descriptor format. You will need to ensure that each mbuf is eventually freed, but only after the E1000 has finished

transmitting the packet (the E1000 sets the `E1000_TXD_STAT_DD` bit in the descriptor to indicate this).

When the E1000 receives each packet from the ethernet, it first DMAs the packet to the mbuf pointed to by the next RX (receive) ring descriptor, and then generates an interrupt. Your `e1000_recv()` code must scan the RX ring and deliver each new packet's mbuf to the network stack (in `net.c`) by calling `net_rx()`. You will then need to allocate a new mbuf and place it into the descriptor, so that when the E1000 reaches that point in the RX ring again it finds a fresh buffer into which to DMA a new packet.

In addition to reading and writing the descriptor rings in RAM, your driver will need to interact with the E1000 through its memory-mapped control registers, to detect when received packets are available and to inform the E1000 that the driver has filled in some TX descriptors with packets to send. The global variable `regs` holds a pointer to the E1000's first control register; your driver can get at the other registers by indexing `regs` as an array. You'll need to use indices `E1000_RDT` and `E1000_TDT` in particular.

To test your driver, run `make server` in one window, and in another window run `make qemu` and then run `nettests` in xv6. The first test in `nettests` tries to send a UDP packet to the host operating system, addressed to the program that `make server` runs. If you haven't completed the lab, the E1000 driver won't actually send the packet, and nothing much will happen.

After you've completed the lab, the E1000 driver will send the packet, qemu will deliver it to your host computer, `make server` will see it, it will send a response packet, and the E1000 driver and then `nettests` will see the response packet. Before the host sends the reply, however, it sends an "ARP" request packet to xv6 to find out its 48-bit Ethernet address, and expects xv6 to respond with an ARP reply. `kernel/net.c` will take care of this once you have finished your work on the E1000 driver. If all goes well, `nettests` will print `testing ping: OK`, and `make server` will print `a message from xv6!`.

tcpdump -XXnr packets.pcap should produce output that starts like this:

```
reading from file packets.pcap, link-type EN10MB (Ethernet)
15:27:40.861988 IP 10.0.2.15.2000 > 10.0.2.2.25603: UDP, length 19
        0x0000:  ffff ffff ffff 5254 0012 3456 0800 4500  ......RT..4V..E.
        0x0010:  002f 0000 0000 6411 3eae 0a00 020f 0a00  ./....d.>.......
        0x0020:  0202 07d0 6403 001b 0000 6120 6d65 7373  ....d.....a.mess
        0x0030:  6167 6520 6672 6f6d 2078 7636 21         age.from.xv6!
15:27:40.862370 ARP, Request who-has 10.0.2.15 tell 10.0.2.2, length 28
        0x0000:  ffff ffff ffff 5255 0a00 0202 0806 0001  ......RU........
        0x0010:  0800 0604 0001 5255 0a00 0202 0a00 0202  ......RU........
        0x0020:  0000 0000 0000 0a00 020f                 ..........
15:27:40.862844 ARP, Reply 10.0.2.15 is-at 52:54:00:12:34:56, length 28
        0x0000:  ffff ffff ffff 5254 0012 3456 0806 0001  ......RT..4V....
        0x0010:  0800 0604 0002 5254 0012 3456 0a00 020f  ......RT..4V....
        0x0020:  5255 0a00 0202 0a00 0202                 RU........
15:27:40.863036 IP 10.0.2.2.25603 > 10.0.2.15.2000: UDP, length 17
        0x0000:  5254 0012 3456 5255 0a00 0202 0800 4500  RT..4VRU......E.
        0x0010:  002d 0000 0000 4011 62b0 0a00 0202 0a00  .-....@.b.......
        0x0020:  020f 6403 07d0 0019 3406 7468 6973 2069  ..d.....4.this.i
        0x0030:  7320 7468 6520 686f 7374 21              s.the.host!
```

Your output will look somewhat different, but it should contain the strings "ARP, Request", "ARP, Reply", "UDP", "a.message.from.xv6" and "this.is.the.host".

`nettests` performs some other tests, culminating in a DNS request sent over the (real) Internet to one of Google's name server. You should ensure that your code passes all these tests, after which you should see this output:

```
$ nettests
nettests running on port 25603
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
```

You should ensure that `make grade` agrees that your solution passes.

Hints

Start by adding print statements to `e1000_transmit()` and `e1000_recv()`, and running `make server` and (in xv6) `nettests`. You should see from your print statements that `nettests` generates a call to `e1000_transmit`.

Some hints for implementing `e1000_transmit`:

- First ask the E1000 for the TX ring index at which it's expecting the next packet, by reading the `E1000_TDT` control register.
- Then check if the the ring is overflowing. If `E1000_TXD_STAT_DD` is not set in the descriptor indexed by `E1000_TDT`, the E1000 hasn't finished the corresponding previous transmission request, so return an error.
- Otherwise, use `mbuffree()` to free the last mbuf that was transmitted from that descriptor (if there was one).
- Then fill in the descriptor. `m->head` points to the packet's content in memory, and `m->len` is the packet length. Set the necessary cmd flags (look at Section 3.3 in the E1000 manual) and stash away a pointer to the mbuf for later freeing.
- Finally, update the ring position by adding one to `E1000_TDT` modulo `TX_RING_SIZE`.
- If `e1000_transmit()` added the mbuf successfully to the ring, return 0. On failure (e.g., there is no descriptor available to transmit the mbuf), return -1 so that the caller knows to free the mbuf.

Some hints for implementing `e1000_recv`:

- First ask the E1000 for the ring index at which the next waiting received packet (if any) is located, by fetching the `E1000_RDT` control register and adding one modulo `RX_RING_SIZE`.
- Then check if a new packet is available by checking for the `E1000_RXD_STAT_DD` bit in the `status` portion of the descriptor. If not, stop.
- Otherwise, update the mbuf's `m->len` to the length reported in the descriptor. Deliver the mbuf to the network stack using `net_rx()`.
- Then allocate a new mbuf using `mbufalloc()` to replace the one just given to `net_rx()`. Program its data pointer (`m->head`) into the descriptor. Clear the descriptor's status bits to

zero.

- Finally, update the `E1000_RDT` register to be the index of the last ring descriptor processed.
- `e1000_init()` initializes the RX ring with mbufs, and you'll want to look at how it does that and perhaps borrow code.
- At some point the total number of packets that have ever arrived will exceed the ring size (16); make sure your code can handle that.

You'll need locks to cope with the possibility that xv6 might use the E1000 from more than one process, or might be using the E1000 in a kernel thread when an interrupt arrives.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.SubmitYou will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

If you run make handin and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional Challenges:

Some of the benefits of the challenge exercises below are only measurable/testable on real, high-performance hardware, which means x86-based computers.

- In this lab, the networking stack uses interrupts to handle ingress packet processing, but not egress packet processing. A more sophisticated strategy would be to queue egress packets in software and only provide a limited number to the NIC at any one time. You can then rely on TX interrupts to refill the transmit ring. Using this technique, it becomes possible to prioritize different types of egress traffic. (easy)
- The provided networking code only partially supports ARP. Implement a full ARP cache and wire it in to `net_tx_eth()`. (moderate)
- The E1000 supports multiple RX and TX rings. Configure the E1000 to provide a ring pair for each core and modify your networking stack to support multiple rings. Doing so has the potential to increase the throughput that your networking stack can support as well as reduce lock contention. (moderate), but difficult to test/measure
- `sockrecvudp()` uses a singly-linked list to find the destination socket, which is inefficient. Try using a hash table and RCU instead to increase performance. (easy), but a serious implementation would difficult to test/measure
- ICMP can provide notifications of failed networking flows. Detect these notifications and propagate them as errors through the socket system call interface.
- The E1000 supports several stateless hardware offloads, including checksum calculation, RSC, and GRO. Use one or more of these offloads to increase the throughput of your networking stack. (moderate), but hard to test/measure
- The networking stack in this lab is susceptible to receive livelock. Using the material in lecture and the reading assignment, devise and implement a solution to fix it. (moderate), but hard to test.
- Implement a UDP server for xv6. (moderate)
- Implement a minimal TCP stack and download a web page. (hard)

If you pursue a challenge problem, whether it is related to networking or not, please let the course staff know!