Toggle navigation 6.828: Operating System Engineering

- **Schedule**
- **Class**
  - Overview
  - Administrivia
  - Related classes
  - 6.828 2017
- **Labs**
  - Tools
  - Lab guide
  - Lab 1
  - Lab 2
  - Lab 3
  - Lab 4
  - Lab 5
  - Lab 6
  - Lab 7
- **xv6**
  - xv6
  - xv6 printout
  - xv6 book
- **References**
- **Piazza**

2018

Links to notes etc. on future days are copies of materials from 2017 to give you an idea of what the future will bring. We will update the notes as the course progresses. The lecture notes may help you remember the lecture content, but they are not a replacement for attending lectures.

sep 3	sep 4 Reg Day	sep 5 **LEC 1 (fk):** Operating systems (handouts: xv6 source, xv6 book) **Preparation**: Unix intro **Assignment**: Lab 1: C, Assembly, Tools, and Bootstrapping	sep 6	sep 7
sep 10 **LEC 2 (ab):** PC hardware and x86 programming slides, notes **Preparation**: Read Bootstrap/PC hardware appendices and the related xv6 source files **Assignment**: HW: Boot xv6	sep 11	sep 12 **LEC 3 (TAs):** C and gdb (pointers example) **Homework 1 due:** Boot xv6 **Preparation**: Read 2.9 (Bitwise operators), 5.1 (Pointers and Addresses) through 5.5(Character Pointers and Functions) and 6.4 (pointers to structures) in K&R **Assignment**: HW: shell **Assignment**: Lab 2: Memory management	sep 13 **DUE**: Lab 1	sep 14
sep 17 **LEC 4 (fk):** Shell & OS organization **Preparation**: Read chapter 0 of the xv6 book. **Homework 2 due:** shell	sep 18	sep 19 **LEC 5 (fk):** Isolation mechanisms **Preparation**: Read "Chapter 1: Operating system organization" and the related xv6 source files **Assignment**: HW: system calls	sep 20	sep 21

sep 24 **LEC 6 (ab):** Virtual memory (1) (handouts: page table translation and registers) (slides) **Preparation**: Read "Chapter 2: Page Tables" **Homework 3 due:** system calls **Assignment**: HW lazy page allocation	sep 25	sep 26 **LEC 7 (ab):** Virtual memory (2) (handout: JOS virtual memory layout) (slides) **Homework 4 due:** HW lazy page allocation **Assignment**: HW xv6 CPU alarm **Assignment**: Lab 3: User-Level Environments	sep 27 **DUE**: Lab 2	sep 28
oct 1 **LEC 8 (fk):** System calls, interrupts, exceptions (handouts: IDT) **Preparation**: Read "Traps, interrupts, and drivers" and the related xv6 source files **Homework 5 due:** HW xv6 CPU alarm **Assignment**: HW multithreaded programming	oct 2	oct 3 **LEC 9 (ab):** Multiprocessors and locking (slides) **Preparation**: Read "Locking" with spinlock.c and skim mp.c **Homework 6 due:** HW multithreaded programming **Assignment**: HW xv6 locks	oct 4 **DUE**: Lab 3 (Part A)	oct 5 ADD DATE
oct 8 - oct 9 Columbus Day		oct 10 Hacking day **Assignment**: Lab 4: Preemptive Multitasking	oct 11 **DUE**: Lab 3 (Part B)	oct 12
oct 15 **LEC 10 (ab):** Processes and switching **Preparation**: Read "Scheduling" up to "Sleep and wakeup" and proc.c, swtch.S **Homework 7 due:** HW xv6 locks **Assignment**: HW uthreads	oct 16	oct 17 **LEC 11 (fk):** sleep&wakeup **Preparation**: Read remainder of "Scheduling", and corresponding parts of proc.c **Homework 8 due:** HW uthreads **Assignment**: HW barrier	oct 18 **DUE**: Lab 4 (Part A)	oct 19
oct 22 **LEC 12 (fk):** File systems **Preparation**: Read bio.c, fs.c, sysfile.c, file.c and "File system" except for the logging sections **Homework 9 due:** HW barrier **Assignment**: HW big files	oct 23	oct 24 **LEC 13 (fk):** Crash recovery **Preparation**: Read log.c and the logging sections of "File system" **Homework 10 due:** HW big files **Assignment**: HW crash **Assignment**: Lab 6: Networking **Assignment**: Lab 7: Final project	oct 25 **DUE**: Lab 4 (Part B)	oct 26
oct 29 **Quiz #1 open book and notes** **scope**: Lectures 1 through 13, HW 1 through 10, labs 1 through 3 **practice:** previous years' quizzes.	oct 30	oct 31 Hacking day **Assignment**: Lab 5: File system, spawn, and sh	nov 1 **DUE**: Lab 4 (Part C)	nov 2 **DUE**: Piazza discussion final project
nov 5 **LEC 14 (fk):** File system performance and fast crash recovery **Homework 11 due:** HW crash **Preparation**: Read Journaling the Linux ext2fs Filesystem (1998) **Assignment**: mmap()	nov 6	nov 7 **LEC 15 (ab):** Virtual Memory (3) (slides) **Preparation**: Read Virtual Memory Primitives for User Programs (1991) **Homework 12 due:** mmap	nov 8 **DUE**: Final project proposal (if doing project)	nov 9
nov 12 Veteran's Day	nov 13	nov 14 **LEC 16 (fk):** OS Organization **Preparation**: Read Exokernel (1995)	nov 15 **DUE**: Lab 5	nov 16

		Homework 13 due: HW exokernel question		
nov 19 **LEC 17 (cc):** Kernels and HLL **Preparation**: Read the Biscuit paper (2018) **Homework 14 due:** HW Biscuit question	nov 20	nov 21 **DROP DATE** **LEC 18 (ab):** Scalable locks (slides) (code) **Preparation**: Read Non-scalable locks paper (2012) **Homework 15 due:** ticket lock question	nov 22 - nov 23 Thanksgiving	
nov 26 **LEC 19 (ab):** Scaling OSes slides **Preparation**: Read RCU (2013) **Homework 16 due:** RCU question **DUE**: Email us a status update on your final project (a paragraph)	nov 27	nov 28 **LEC 20 (ab):** Virtual Machines (1) (slides) **Preparation**: Read Software vs Hardware Virtualization (2006) **Homework 17 due:** VM question	nov 29	nov 30
dec 3 **LEC 21 (ab):** Virtual Machines (2) (slides) **Preparation**: Read Dune: Safe User-level Access to Privileged CPU Features (2012) **Homework 18 due:** Dune question	dec 4	dec 5 Hacking day	dec 6 **DUE**: Lab 6 (networking) or lab 7 (Final project)	dec 7
dec 10 **LEC 22 (fk):** High-performance networking **Preparation**: Read IX: A Protected Dataplane Operating System for High Throughput and Low Latency (2014) **Homework 19 due:** IX question	dec 11	dec 12 LAST DAY OF CLASSES **LEC 23:** demos in class **Project sign-offs**	dec 13	dec 14
dec 17	dec 18	dec 19	dec 20 **Quiz #2** **DUPONT-Gym, 9am-11pm** **open book, notes, and papers.** **scope**: Lectures 14 through last lecture (dec 12), labs 4 and 5. **practice:** previous years' quizzes.	dec 21

Questions or comments regarding 6.828? Send e-mail to the TAs at *6828-staff@lists.csail.mit.edu*.

Top // **6.828 home** // *Last updated Wednesday, 25-Mar-2020 15:11:02 EDT*

Homework: boot xv6

Submit your solutions before the beginning of the lecture (i.e., 1pm) on the due date mentioned on the schedule page to the submission web site.

Boot xv6

Login to Athena (e.g., ssh -X athena.dialup.mit.edu) and attach the course locker: (You must run this command every time you log in; or add it to your ~/.environment file.)

```
$ add -f 6.828
```

Fetch the xv6 source:

```
$ mkdir 6.828
$ cd 6.828
$ git clone git://github.com/mit-pdos/xv6-public.git
Cloning into 'xv6-public'...
...
$
```

Build xv6 on Athena:

```
$ cd xv6-public
$ make
...
gcc -O -nostdinc -I. -c bootmain.c
gcc -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
...
$
```

If you are not using Athena for 6.828 JOS labs, but build on your own machine, see the instructions on the tools page. If you have a build infrastructure on your own machine for lab 1, then you should be able to use that infrastructure for building xv6 too.

Finding and breaking at an address

Find the address of `_start`, the entry point of the kernel:

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

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

Run the kernel inside QEMU GDB, setting a breakpoint at `_start` (i.e., the address you just found).

```
$ make qemu-gdb
...
<leave "make qemu-gdb" running, and in a new terminal, navigate to the same
directory and run the following. If you are trying this by logging into
athena.dialup.mit.edu, check the hostname to make sure that you are running
both the commands on the same physical machine.>
$ gdb
```

```
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) br * 0x0010000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

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

The details of what you see are likely to differ from the above output, depending on the version of gdb you are using, but gdb should stop at the breakpoint, and it should be the above `mov` instruction. Your gdb may also complain that auto-loading isn't enabled. In that case, it will print instructions on how to enable auto-loading, and you should follow those instructions.

Exercise: What is on the stack?

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

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

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

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

- Begin by restarting qemu and gdb, and set a break-point at 0x7c00, the start of the boot block (bootasm.S). Single step through the instructions (type `si` at the gdb

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

Submit: The output of `x/24x $esp` with the valid part of the stack marked, plus your comments, in a file named hwN.txt (where N is the homework number as listed on the schedule).

Homework: shell

This assignment will make you more familiar with the Unix system call interface and the shell by implementing several features in a small shell, which we will refer to as the 6.828 shell. You can do this assignment on any operating system that supports the Unix API (a Linux Athena machine, your laptop with Linux or MacOS, etc.). Submit your 6.828 shell to the submission web site as a text file with the name "hwN.c", where N is the homework number as listed on the schedule.

Read Chapter 0 of the xv6 book.

If you are not familiar with what a shell does, do the Unix hands-on from 6.033.

Download the 6.828 shell, and look it over. The 6.828 shell contains two main parts: parsing shell commands and implementing them. The parser recognizes only simple shell commands such as the following:

```
ls > y
cat < y | sort | uniq | wc > y1
cat y1
rm y1
ls |  sort | uniq | wc
rm y
```

Cut and paste these commands into a file `t.sh`

To compile `sh.c`, you need a C compiler, such as gcc. On Athena, you can type:

```
$ add gnu
```

to make gcc available. If you are using your own computer, you may have to install gcc.

Once you have gcc, you can compile the skeleton shell as follows:

```
$ gcc sh.c
```

which produces an `a.out` file, which you can run:

```
$ ./a.out < t.sh
```

This execution will print error messages because you have not implemented several features. In the rest of this assignment you will implement those features.

Executing simple commands

Implement simple commands, such as:

```
$ ls
```

The parser already builds an `execcmd` for you, so the only code you have to write is for the ' ' case in `runcmd`. You might find it useful to look at the manual page for exec; type "man 3 exec", and read about `execv`. Print an error message when exec fails.

To test your program, compile and run the resulting a.out:

```
6.828$./a.out
```

This prints a prompt and waits for input. `sh.c` prints as prompt `6.828$` so that you don't get confused with your computer's shell. Now type to your shell:

```
6.828$ ls
```

Your shell may print an error message (unless there is a program named `ls` in your working directory or you are using a version of `exec` that searches `PATH`). Now type to your shell:

```
6.828$ /bin/ls
```

This should execute the program `/bin/ls`, which should print out the file names in your working directory. You can stop the 6.828 shell by typing ctrl-d, which should put you back in your computer's shell.

You may want to change the 6.828 shell to always try `/bin`, if the program doesn't exist in the current working directory, so that below you don't have to type "/bin" for each program. If you are ambitious you can implement support for a `PATH` variable.

I/O redirection

Implement I/O redirection commands so that you can run:

```
echo "6.828 is cool" > x.txt
cat < x.txt
```

The parser already recognizes ">" and "<", and builds a `redircmd` for you, so your job is just filling out the missing code in `runcmd` for those symbols. You might find the man pages for open and close useful.

Note that the `mode` field in `redircmd` contains access modes (e.g., `O_RDONLY`), which you should pass in the `flags` argument to `open`; see `parseredirs` for the mode values that the shell is using and the manual page for `open` for the `flags` argument.

Make sure you print an error message if one of the system calls you are using fails.

Make sure your implementation runs correctly with the above test input. A common error is to forget to specify the permission with which the file must be created (i.e., the 3rd argument to open).

Implement pipes

Implement pipes so that you can run command pipelines such as:

```
$ ls | sort | uniq | wc
```

The parser already recognizes "|", and builds a `pipecmd` for you, so the only code you must write is for the '|' case in `runcmd`. You might find the man pages for pipe, fork, close, and dup useful.

Test that you can run the above pipeline. The `sort` program may be in the directory `/usr/bin/` and in that case you can type the absolute pathname `/usr/bin/sort` to run sort. (In

your computer's shell you can type `which sort` to find out which directory in the shell's search path has an executable named "sort".)

Now you should be able to run the following command correctly:

```
6.828$ a.out < t.sh
```

Make sure you use the right absolute pathnames for the programs.

Don't forget to submit your solution to the submission web site (using the "hwN.c" naming convention as outlined above), with or without challenge solutions.

Optional challenge exercises

The following exercises are entirely optional and won't affect your grade. Add any feature of your choice to your shell, for example:

- Implement lists of commands, separated by ";"
- Implement sub shells by implementing "(" and ")"
- Implement running commands in the background by supporting "&" and "wait"

All of these require making changes to the parser and the `runcmd` function.

Homework: xv6 system calls

Submit your solutions before the beginning of the next lecture to the submission web site.

You will modify xv6 to add a system call. You can use the same setup as for the boot homework.

Part One: System call tracing

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

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

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

That's init forking and execing sh, sh making sure only two file descriptors are open, and sh writing the $ prompt. (Note: the output of the shell and the system call trace are intermixed, because the shell uses the write syscall to print its output.)

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

Optional challenge: print the system call arguments.

Part Two: Date system call

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

You should create a user-level program that calls your new date system call; here's some source you should put in `date.c`:

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

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

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

```
    exit();
  }

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

  exit();
}
```

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

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

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

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

Optional challenge: add a dup2() system call and modify the shell to use it.

> **Submit**: Your explanations of the modifications for date in a file named hwN.txt where N is the homework number as posted on the schedule.

Homework: xv6 lazy page allocation

Submit your solutions before the beginning of the next lecture to the submission web site.

One of the many neat tricks an O/S can play with page table hardware is lazy allocation of 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. There are programs that allocate memory but never use it, for example to implement large sparse arrays. Sophisticated kernels delay allocation of each page of memory until the application tries to use that page -- as signaled by a page fault. You'll add this lazy allocation feature to xv6 in this exercise.

Part One: Eliminate allocation from sbrk()

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
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004--kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. Make sure you understand why this page fault occurs. The "addr 0x4004" indicates that the virtual address that caused the page fault is 0x4004.

Part Two: Lazy allocation

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 `cprintf` call that produced the "pid 3 sh: trap 14" message. Your code is not required to cover all corner cases and error situations; it just needs to be good enough to let sh run simple commands like echo and ls.

Hint: look at the cprintf arguments to see how to find the virtual address that caused the page fault.

Hint: steal code from allocuvm() in vm.c, which is what sbrk() calls (via growproc()).

Hint: use PGROUNDDOWN(va) to round the faulting virtual address down to a page boundary.

Hint: `break` or `return` in order to avoid the `cprintf` and the `myproc()->killed = 1`.

Hint: you'll need to call mappages(). In order to do this you'll need to delete the `static` in the declaration of mappages() in vm.c, and you'll need to declare mappages() in trap.c. Add this declaration to trap.c before any call to `mappages()`:

```
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

Hint: you can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in trap().

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) in the shell, and perhaps two.

By the way, this is not a fully correct implementation. See the challenges below for a list of problems we're aware of.

Optional challenges: Handle negative sbrk() arguments. Handle error cases such as sbrk() arguments that are too large. Verify that fork() and exit() work even if some sbrk()'d address have no memory allocated for them. Correctly handle faults on the invalid page below the stack. Make sure that kernel use of not-yet-allocated user addresses works -- for example, if a program passes an sbrk()-allocated address to read().

Submit: The code that you added to trap.c in a file named *hwN.c* where *N* is the homework number as listed on the schedule.

Homework: xv6 CPU alarm

Submit your solutions before the beginning of the next lecture to the submission web site.

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.

You should add a new `alarm(interval, handler)` system call. If an application calls `alarm(n, fn)`, then after every `n` "ticks" of CPU time that the program consumes, the kernel will cause application function `fn` to be called. When `fn` returns, the application will 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.

You should put the following example program in `alarmtest.c`:

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

void periodic();

int
main(int argc, char *argv[])
{
  int i;
  printf(1, "alarmtest starting\n");
  alarm(10, periodic);
  for(i = 0; i < 25*500000; i++){
    if((i % 250000) == 0)
      write(2, ".", 1);
  }
  exit();
}

void
periodic()
{
  printf(1, "alarm!\n");
}
```

The program calls `alarm(10, periodic)` to ask the kernel to force a call to `periodic()` every 10 ticks, and then spins for a while. After you have implemented the `alarm()` system call in the kernel, `alarmtest` should produce output like this:

```
$ alarmtest
alarmtest starting
.....alarm!
....alarm!
.....alarm!
......alarm!
.....alarm!
....alarm!
....alarm!
```

```
......alarm!
.....alarm!
...alarm!
...$
```

(If you only see one "alarm!", try increasing the number of iterations in `alarmtest.c` by 10x.)

Hint: you'll need to modify the Makefile to cause `alarmtest.c` to be compiled as an xv6 user program.

Hint: the right declaration to put in `user.h` is:

```
int alarm(int ticks, void (*handler)());
```

Hint: You will also have to update `syscall.h` and `usys.S` to allow `alarmtest` to invoke the alarm system call.

Hint: Your `sys_alarm()` should store the alarm interval and the pointer to the handler function in new fields in the `proc` structure; see `proc.h`.

Hint: here's a `sys_alarm()` for free:

```
int
sys_alarm(void)
{
  int ticks;
  void (*handler)();

  if(argint(0, &ticks) < 0)
    return -1;
  if(argptr(1, (char**)&handler, 1) < 0)
    return -1;
  myproc()->alarmticks = ticks;
  myproc()->alarmhandler = handler;
  return 0;
}
```

Add it to `syscall.c` and add an entry for `SYS_ALARM` to the `syscalls` array in that file.

Hint: 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`.

Hint: Every tick, the hardware clock forces an interrupt, which is handled in `trap()` by `case T_IRQ0 + IRQ_TIMER`; you should add some code here.

Hint: You only want to manipulate a process's alarm ticks if there's a process running and if the timer interrupt came from user space; you want something like

```
if(myproc() != 0 && (tf->cs & 3) == 3) ...
```

Hint: In your `IRQ_TIMER` code, when a process's alarm interval expires, you'll want to cause it to execute its handler. How can you do that?

Hint: You need to arrange things so that, when the handler returns, the process resumes executing where it left off. How can you do that?

Hint: You can see the assembly code for the alarmtest program in alarmtest.asm.

Hint: 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
```

It's OK if your solution doesn't save the caller-saved user registers when calling the handler.

Optional challenges: 1) Save and restore the caller-saved user registers around the call to handler. 2) Prevent re-entrant calls to the handler -- if a handler hasn't returned yet, don't call it again. 3) Assuming your code doesn't check that `tf->esp` is valid, implement a security attack on the kernel that exploits your alarm handler calling code.

Submit: The code that you added to trap.c.

Homework: Threads and Locking

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this homework on a real computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors. Submit your solutions before the beginning of the next lecture to the submission web site.

Download ph.c and compile it on your laptop or Athena machine (you can use your OS' gcc; you don't need the 6.828 tools):

```
$ gcc -g -O2 ph.c -pthread
$ ./a.out 2
```

The 2 specifies the number of threads that execute put and get operations on the the hash table.

After running for a little while, the program will produce output similar to this:

```
1: put time = 0.003338
0: put time = 0.003389
0: get time = 7.684335
0: 17480 keys missing
1: get time = 7.684335
1: 17480 keys missing
completion time = 7.687856
```

Each thread runs in two phases. In the first phase each thread puts NKEYS/nthread keys into the hash table. In the second phase, each thread gets NKEYS from the hash table. The print statements tell you how long each phase took for each thread. The completion time at the bottom tells you the total runtime for the application. In the output above, the completion time for the application is about 7.7 seconds. Each thread computed for about 7.7 seconds (~0.0 for put + ~7.7 for get).

To see if using two threads improved performance, let's compare against a single thread:

```
$ ./a.out 1
0: put time = 0.004073
0: get time = 6.929189
0: 0 keys missing
completion time = 6.933433
```

The completion time for the 1 thread case (~7.0s) is slightly less than for the 2 thread case (~7.7s), but the two-thread case did twice as much total work during the get phase. Thus the two-thread case achieved nearly 2x parallel speedup for the get phase on two cores, which is very good.

When you run this application, you may see no parallelism if you are running on a machine with only one core or if the machine is busy running other applications.

Two points: 1) The completion time is about the same as for 2 threads, but this run did twice as many gets as with 2 threads; we are achieving good parallelism. 2) The output for 2 threads says that many keys are missing. In your runs, there may be more or fewer keys missing. If you run with 1 thread, there will never be any keys missing.

Why are there missing keys with 2 or more threads, but not with 1 thread? Identify a sequence of events that can lead to keys missing for 2 threads.

To avoid this sequence of events, insert lock and unlock statements in `put` and `get` so that the number of keys missing is always 0. The relevant pthread calls are (for more see the manual pages, man pthread):

```
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
```

Test your code first with 1 thread, then test it with 2 threads. Is it correct (i.e. have you eliminated missing keys?)? Is the two-threaded version faster than the single-threaded version?

Modify your code so that `get` operations run in parallel while maintaining correctness. (Hint: are the locks in `get` necessary for correctness in this application?)

Modify your code so that some `put` operations run in parallel while maintaining correctness. (Hint: would a lock per bucket work?)

Submit: your modified ph.c

Homework: xv6 locking

In this assignment you will explore some of the interaction between interrupts and locking. Submit your solutions before the beginning of the next lecture to the submission web site.

Don't do this

Make sure you understand what would happen if the xv6 kernel executed the following code snippet:

```
struct spinlock lk;
initlock(&lk, "test lock");
acquire(&lk);
acquire(&lk);
```

(Feel free to use QEMU to find out. `acquire` is in `spinlock.c`.)

> **Submit**: Explain in one sentence what happens.

Interrupts in ide.c

An `acquire` ensures that interrupts are off on the local processor using the `cli` instruction (via `pushcli()`), and that interrupts remain off until the `release` of the last lock held by that processor (at which point they are enabled using `sti`).

Let's see what happens if we turn on interrupts while holding the `ide` lock. In `iderw` in `ide.c`, add a call to `sti()` after the `acquire()`, and a call to `cli()` just before the `release()`. Rebuild the kernel and boot it in QEMU. Chances are the kernel will panic soon after boot; try booting QEMU a few times if it doesn't.

> **Submit**: Explain in a few sentences why the kernel panicked. You may find it useful to look up the stack trace (the sequence of `%eip` values printed by `panic`) in the `kernel.asm` listing.

Interrupts in file.c

Remove the `sti()` and `cli()` you added, rebuild the kernel, and make sure it works again.

Now let's see what happens if we turn on interrupts while holding the `file_table_lock`. This lock protects the table of file descriptors, which the kernel modifies when an application opens or closes a file. In `filealloc()` in `file.c`, add a call to `sti()` after the call to `acquire()`, and a `cli()` just before **each** of the `release()`es. You will also need to add `#include "x86.h"` at the top of the file after the other `#include` lines. Rebuild the kernel and boot it in QEMU. It most likely will not panic.

> **Submit**: Explain in a few sentences why the kernel didn't panic. Why do `file_table_lock` and `ide_lock` have different behavior in this respect?
>
> You do not need to understand anything about the details of the IDE hardware to answer this question, but you may find it helpful to look at which functions acquire each lock, and then at when those functions get called.

(On qemu there is a small chance that the kernel will panic with the extra `sti()` in `filealloc()`. If the kernel *does* panic, make doubly sure that you removed the `sti()` call from `iderw`. If it continues to panic and the only extra `sti()` is in `filealloc()`, then think about why this should be unlikely on real hardware. If you are running on real hardware, then mail *6.828-staff@pdos.csail.mit.edu* and think about buying a lottery ticket.)

xv6 lock implementation

> **Submit**: Why does `release()` clear `lk->pcs[0]` and `lk->cpu` *before* clearing `lk->locked`? Why not wait until after?

Homework: User-level threads

In this assignment you will complete a simple user-level thread package by implementing the code to perform context switching between threads. Submit your solutions before the beginning of the next lecture to the submission web site.

Switching threads

Download uthread.c and uthread_switch.S into your xv6 directory. Make sure `uthread_switch.S` ends with `.S`, not `.s`. Add the following rule to the xv6 Makefile after the _forktest rule:

```
_uthread: uthread.o uthread_switch.o
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _uthread uthread.o uthread_switch.o $(ULIB)
	$(OBJDUMP) -S _uthread > uthread.asm
```

Make sure that the blank space at the start of each line is a tab, not spaces.

Add `_uthread` in the Makefile to the list of user programs defined by UPROGS.

Run xv6, then run `uthread` from the xv6 shell. The xv6 kernel will print an error message about `uthread` encountering a page fault.

Your job is to complete `thread_switch.S`, so that you see output similar to this (make sure to run with CPUS=1):

```
~/classes/6828/xv6$ make CPUS=1 qemu-nox
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0190287 s, 269 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 7.2168e-05 s, 7.1 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
291+1 records in
291+1 records out
149040 bytes (149 kB, 146 KiB) copied, 0.000528827 s, 282 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 512
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ uthread
my thread running
my thread 0x2D68
my thread running
my thread 0x4D70
my thread 0x2D68
my thread 0x4D70
my thread 0x2D68
my thread 0x4D70
my thread 0x2D68
...
```

`uthread` creates two threads and switches back and forth between them. Each thread prints "my thread ..." and then yields to give the other thread a chance to run.

To observe the above output, you need to complete `thread_switch.S`, but before jumping into `uthread_switch.S`, first understand how `uthread.c` uses `thread_switch`. `uthread.c` has two global variables `current_thread` and `next_thread`. Each is a pointer to a `thread` structure. The thread structure has a stack for a thread and a saved stack pointer (`sp`, which points into the thread's stack). The job of `uthread_switch` is to save the current thread state into the structure pointed to by `current_thread`, restore `next_thread`'s state, and make `current_thread` point to where `next_thread` was pointing to, so that when `uthread_switch` returns `next_thread` is running and is the `current_thread`.

You should study `thread_create`, which sets up the initial stack for a new thread. It provides hints about what `thread_switch` should do. The intent is that `thread_switch` use the assembly instructions `popal` and `pushal` to restore and save all eight x86 registers. Note that `thread_create` simulates eight pushed registers (32 bytes) on a new thread's stack.

To write the assembly in `thread_switch`, you need to know how the C compiler lays out `struct thread` in memory, which is as follows:

```
--------------------
| 4 bytes for state|
--------------------
| stack size bytes |
| for stack        |
--------------------
| 4 bytes for sp   |
--------------------  <--- current_thread
     ......

     ......
--------------------
| 4 bytes for state|
--------------------
| stack size bytes |
| for stack        |
--------------------
| 4 bytes for sp   |
--------------------  <--- next_thread
```

The variables `next_thread` and `current_thread` each contain the address of a `struct thread`.

To write the `sp` field of the struct that `current_thread` points to, you should write assembly like this:

```
movl current_thread, %eax
movl %esp, (%eax)
```

This saves `%esp` in `current_thread->sp`. This works because `sp` is at offset 0 in the struct. You can study the assembly the compiler generates for `uthread.c` by looking at `uthread.asm`.

To test your code it might be helpful to single step through your `thread_switch` using `gdb`. You can get started in this way:

```
(gdb) symbol-file _uthread
Load new symbol table from "/Users/kaashoek/classes/6828/xv6/_uthread"? (y or n) y
Reading symbols from /Users/kaashoek/classes/6828/xv6/_uthread...done.
(gdb) b thread_switch
Breakpoint 1 at 0x204: file uthread_switch.S, line 9.
(gdb)
```

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 `thread_switch`. Now you can type commands like the following to inspect the state of `uthread`:

```
(gdb) p/x next_thread->sp
$4 = 0x4ae8
(gdb) x/9x next_thread->sp
0x4ae8 :	0x00000000	0x00000000	0x00000000	0x00000000
0x4af8 :	0x00000000	0x00000000	0x00000000	0x00000000
0x4b08 :	0x000000d8
```

What address is `0xd8`, which sits on the top of the stack of `next_thread`?

> **Submit**: your modified uthread_switch.S

Optional challenges

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.

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

Barriers

In this assignment we will explore how to implement a barrier using condition variables provided by the pthread library. A barrier is a point in an application at which all threads must wait until all other threads reach that point too. Condition variables are a sequence coordination technique similar to xv6's sleep and wakeup. Submit your solutions before the beginning of the next lecture to the submission web site.

Download barrier.c and compile it on your laptop or Athena machine:

```
$ gcc -g -O2 -pthread barrier.c
$ ./a.out 2
Assertion failed: (i == t), function thread, file barrier.c, line 55.
```

The 2 specifies the number of threads that synchronize on the barrier (`nthread` in `barrier.c`). Each thread sits in a tight loop. In each loop iteration a thread calls `barrier()` and then sleeps for some 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 all threads should block until `nthread`s have called `barrier`.

Your goal is to achieve the desired behavior. In addition to the lock primitives that you have seen before, you will need the following new pthread primitives (see man pthreads for more detail):

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

`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 won'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 increase `bstate.round` when each round starts.
- 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 `bstate.nthread` 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**: your modified barrier.c

Homework: bigger files for xv6

In this assignment you'll increase the maximum size of an xv6 file. Currently xv6 files are limited to 140 sectors, or 71,680 bytes. 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 128 more block numbers, for a total of 12+128=140. You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 128 addresses of singly-indirect blocks, each of which can contain up to 128 addresses of data blocks. The result will be that a file will be able to consist of up to 16523 sectors (or about 8.5 megabytes).

Submit your solution before the beginning of the next lecture to the submission web site.

Preliminaries

Modify your Makefile's `CPUS` definition so that it reads:

```
CPUS := 1
```

Add

```
QEMUEXTRA = -snapshot
```

right before `QEMUOPTS`

The above two steps speed up qemu tremendously when xv6 creates large files.

`mkfs` initializes the file system to have fewer than 1000 free data blocks, too few to show off the changes you'll make. Modify `param.h` to set `FSSIZE` to:

```
    #define FSSIZE       20000  // size of file system in blocks
```

Download big.c into your xv6 directory, add it to the UPROGS list, start up xv6, and run `big`. It creates as big a file as xv6 will let it, and reports the resulting size. It should say 140 sectors.

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 here 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" -- a block number 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 don't have to modify xv6 to handle deletion of files with doubly-indirect blocks.

If all goes well, `big` will now report that it can write 16523 sectors. It will take `big` a few dozen seconds to finish.

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 16,384 blocks (really 16383, 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 size 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` too to build the initial file systems. If you delete `fs.img`, `make` on Unix (not xv6) will build a new one for you.

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()`.

Submit: your modified fs.c

Homework: xv6 log

This assignment explores the xv6 log in two parts. First, you'll artificially create a crash which illustrates why logging is needed. Second, you'll remove one inefficiency in the xv6 logging system.

Submit your solution before the beginning of the next lecture to the submission web site.

Creating a Problem

The point of the xv6 log is to cause all the disk updates of a filesystem operation to be atomic with respect to crashes. For example, file creation involves both adding a new entry to a directory and marking the new file's inode as in-use. A crash that happened after one but before the other would leave the file system in an incorrect state after a reboot, if there were no log.

The following steps will break the logging code in a way that leaves a file partially created.

First, replace `commit()` in `log.c` with this code:

```
#include "mmu.h"
#include "proc.h"
void
commit(void)
{
  int pid = myproc()->pid;
  if (log.lh.n > 0) {
    write_log();
    write_head();
    if(pid > 1)             // AAA
      log.lh.block[0] = 0; // BBB
    install_trans();
    if(pid > 1)             // AAA
      panic("commit mimicking crash"); // CCC
    log.lh.n = 0;
    write_head();
  }
}
```

The BBB line causes the first block in the log to be written to block zero, rather than wherever it should be written. During file creation, the first block in the log is the new file's inode updated to have non-zero `type`. Line BBB causes the block with the updated inode to be written to block 0 (whence it will never be read), leaving the on-disk inode still marked unallocated. The CCC line forces a crash. The AAA lines suppress this buggy behavior for `init`, which creates files before the shell starts.

Second, replace `recover_from_log()` in `log.c` with this code:

```
static void
recover_from_log(void)
{
  read_head();
  cprintf("recovery: n=%d but ignoring\n", log.lh.n);
```

```
  // install_trans();
  log.lh.n = 0;
  // write_head();
}
```

This modification suppresses log recovery (which would repair the damage caused by your change to `commit()`).

Finally, remove the `-snapshot` option from the definition of `QEMUEXTRA` in your Makefile so that the disk image will see the changes.

Now remove `fs.img` and run xv6:

```
  % rm fs.img ; make qemu
```

Tell the xv6 shell to create a file:

```
  $ echo hi > a
```

You should see the panic from `commit()`. So far it is as if a crash occurred in a non-logging system in the middle of creating a file.

Now re-start xv6, keeping the same `fs.img`:

```
  % make qemu
```

And look at file `a`:

```
  $ cat a
```

You should see `panic: ilock: no type`. Make sure you understand what happened. Which of the file creation's modifications were written to the disk before the crash, and which were not?

Solving the Problem

Now fix `recover_from_log()`:

```
static void
recover_from_log(void)
{
  read_head();
  cprintf("recovery: n=%d\n", log.lh.n);
  install_trans();
  log.lh.n = 0;
  write_head();
}
```

Run xv6 (keeping the same `fs.img`) and read `a` again:

```
  $ cat a
```

This time there should be no crash. Make sure you understand why the file system now works.

Why was the file empty, even though you created it with `echo hi > a`?

Now remove your modifications to `commit()` (the if's and the AAA and BBB lines), so that logging works again, and remove `fs.img`.

Streamlining Commit

Suppose the file system code wants to update an inode in block 33. The file system code will call `bp=bread(block 33)` and update the buffer data. `write_log()` in `commit()` will copy the data to a block in the log on disk, for example block 3. A bit later in `commit`, `install_trans()` reads block 3 from the log (containing block 33), copies its contents into the in-memory buffer for block 33, and then writes that buffer to block 33 on the disk.

However, in `install_trans()`, it turns out that the modified block 33 is guaranteed to be still in the buffer cache, where the file system code left it. Make sure you understand why it would be a mistake for the buffer cache to evict block 33 from the buffer cache before the commit.

Since the modified block 33 is guaranteed to already be in the buffer cache, there's no need for `install_trans()` to read block 33 from the log. Your job: modify `log.c` so that, when `install_trans()` is called from `commit()`, `install_trans()` does not perform the needless read from the log.

To test your changes, create a file in xv6, restart, and make sure the file is still there.

Submit: your modified log.c

Homework: mmap()

This assignment will make you more familiar with how to manage virtual memory in user programs using the Unix system call interface. You can do this assignment on any operating system that supports the Unix API (a Linux Athena machine, your laptop with Linux or MacOS, etc.). Submit your solution to the submission web site as a text file with the name "hwN.c", where N is the homework number as listed on the schedule.

Download the mmap homework assignment and look it over. The program maintains a very large table of square root values in virtual memory. However, the table is too large to fit in physical RAM. Instead, the square root values should be computed on demand in response to page faults that occur in the table's address range. Your job is to implement the demand faulting mechanism using a signal handler and UNIX memory mapping system calls. To stay within the physical RAM limit, we suggest using the simple strategy of unmapping the last page whenever a new page is faulted in.

To compile `mmap.c`, you need a C compiler, such as gcc. On Athena, you can type:

```
$ add gnu
```

Once you have gcc, you can compile mmap.c as follows:

```
$ gcc mmap.c -lm -o mmap
```

Which produces a `mmap` file, which you can run:

```
$ ./mmap
page_size is 4096
Validating square root table contents...
oops got SIGSEGV at 0x7f6bf7fd7f18
```

When the process accesses the square root table, the mapping does not exist and the kernel passes control to the signal handler code in `handle_sigsegv()`. Modify the code in `handle_sigsegv()` to map in a page at the faulting address, unmap a previous page to stay within the physical memory limit, and initialize the new page with the correct square root values. Use the function `calculate_sqrts()` to compute the values. The program includes test logic that verifies if the contents of the square root table are correct. When you have completed your task successfully, the process will print “All tests passed!”.

You may find that the man pages for mmap() and munmap() are helpful references.

```
$ man mmap
$ man munmap
```

Don't forget to submit your solution to the submission web site (using the "hwN.c" naming convention as outlined above).