The point of this document:

I'm actually not so sure. I think the point is to explain why xv6 does stuff, in broad strokes, as a complement to the explanations of the actual code of xv6.

Important-ish note:

Often, I'll refer to things that haven't been explained yet. Just keep in mind that not everything is explained at first, and that's okay.

These explanations are not full. They may not even be accurate. I'm just jotting these down during class, hoping it more or less gives a picture of what's going on.

Anyways, here we go.

Kernel

When I say "kernel", I mean the operating system. This is the mother of all programs that hogs the computer to itself and manages all the other programs and how they access the computer resources (such as memory, registers, time (that is, who runs when), etc.).

The kernel also keeps control over the "mode" bit (in the `%cs` register), which marks whether current commands are running in user-mode (and therefore can't do all kinds of privileged stuff) or kernel-mode (and therefore can do whatever it wants).

Boot

The processor (that is, the actual computer) does not know what operating system is installed, what it looks like, or how large it is.

So how can the processor load xv6 into the memory?

The processor instruction pointer `%eip` register points - by default - to a certain memory place in the ROM, which contains a simple program that:

1. Copies the very first block (512 bytes. AKA the boot block) from the disk to the memory
2. Sets the instruction pointer `%eip` to point to the beginning of the newly-copies data

So what?

Every operating system needs to make sure that its first 512 bytes are a small program (it has to be small; it's only 512 bytes!) that loads the rest of the operating system to the memory and sets the PC to whatever place it needs to be in. If 512 bytes aren't enough for this, the program can actually call a slightly larger program that can load and set up more.

In short:

1. `%eip` points to hard-coded ROM code
2. ROM code loads beginning of OS code
3. Beginning of OS code loads the rest of the code
4. Now hear the word of the Lord!

Processes

A process is the running of a program, including the program's state and data. The state includes such things as:

- Memory the program occupied
- Memory contents
- Register values
- Files
- Kernel structures

This is managed by the kernel. The kernel has a simple data structure for each process, organized in some list. The kernel juggles between the processes, using a context switch, which:

1. Saves state of old process to memory
2. Loads state of new process from memory

Context switch can be preemptive (i.e. the kernel decides "next guy's turn!", using the scheduler (there'll be a whole lot of talk about this scheduler guy later on)) or non-preemptive (i.e. the hardware itself decides). In non-preemptive, it is asked of the programmers to make calls to the kernel once in a while, in order to let the kernel choose to let the next guy run.

Xv6 is preemptive.

Processes are created by the kernel, after another process asks it to. Therefore, the kernel needs to run the first process itself, in order to create someone who will ask for new processes to be created.

`fork()`

Every process has a process ID (or pid for short).

A process can call the kernel to do `fork()`, which creates a new process, which is entirely identical to the parent process (registers, memory and everything). The only differences are:

- The pid
- The value returned from fork:
  - In the new process - 0
  - In the parent process - the new pid
  - In case of failure - some negative error code

`exec()`

`Fork()` creates a new process, and leaves the parent running. `Exec()`, on the other hand, replaces the process's program with a new program. It's still the same process, but with new code (and variables, stack, etc.). Registers, pid, etc. remain the same.

It is common practice to have the child of `fork` call `exec` after making sure it is the child. So why not just make a single function that does both fork and exec together? There is a brilliant explanation for this, and I don't remember what it is.

Process termination

Trigger warning: sad stuff ahead. And also zombies.

A process will be terminated if (and only if) one of the following happens:

1. The process invokes `exit()`
2. Some other process invokes `kill()` with its pid
3. The process generates some exception

Note that `kill` does not actually terminate the process. What it does is leave a mark of "You need to kill yourself" on the process, and it'll be the process itself that commits suicide (after it starts running again, when the scheduler loads it).

Note also that not any process can `kill` any other process. The kernel makes sure of that.

Once `kill`ed, the process's resources (memory, etc.) are not released yet, until its parent (that is, the process which called for its creation) allows this to happen. A process that `kill`ed itself but whose parent did not acknowledge this is called a zombie.

In order to a parent to "acknowledge" its child's termination, it needs to call `wait()`. When it calls `wait()`, it will not continue until one of its children exits, and then it will continue. If there are a few children, the parent will need to call `wait` once for each child process.

`Wait` returns the pid of the exited process.

What happens if a parent process `exit`s before its children? Its children become orphans, and the The First Process (whose pid is 1) will make them His children.

System calls

Many commands will only run if the "mode" bit is set to kernel-mode. However, all processes run on user-mode only; xv6 makes sure of that.

In order to run a privileged command, a process must ask the kernel (which runs in kernel-mode, of course) to carry out the command. This "asking" is called a system call.

Here's an example of system call on Linux with an x86 processor:

```
movl flags(%esp), %ecx
lea name(%esp), %ebx
movl ,%eax    ; loads the value 5 into eax register, which is the command "open" in
```

```
Linux
int 8    ; invokes system call. Always 128!
; eax should now contain success code
```

The kernel has a vector with a bunch of pointers to functions (Yay, pointers!).

Addresses

This one's a biggie. Hold on to your seatbelts, kids.

Programs refer to memory addresses. They do this when they refer to a variable (that's right; once code's compiled, all mentions of the variable are turned into the variable's address). They do this in every `if` or loop. When the good old `%eip` register holds the address of the next instruction, it's referring to a memory address.

There are two issues that arise from this:

- The compiled code does not know where in the memory the program is going to be, and therefore these addresses must be relative to the program's actual address. (This is a problem with loops and ifs, not with the `%eip`.)
- We'll want to make sure no evil program tries to access the memory of another program.

So, each process has to have its "own" addresses, which it thinks are the actual addresses. It has *nothing* to do with the actual RAM, just with the *addresses* that the process knows and refers to. (**Process**, not **program**; this includes the kernel.)

Behold! A sketch of what a process's addresses looks like in xv6:

Address	Who uses this
`[0xFFFF FFFF]`	Kernel
`[0xFFFF FFFE]`	Kernel
...	Kernel
`[0x8000 0000]`	**Kernel**
`[0x7FFF FFFF]`	**Process**
...	Process
`[0x0000 0000]`	Process

In order to pull off this trick, we use a hardware piece called the Address Translation Unit, which actually isn't officially called that.
Its real name is the MMU (Memory Management Unit), for some reason.

The MMU is actually comprised of two units:

1. Segmentation Unit
2. Paging Unit.

The MMU sits on the address bus between then CPU and the memory, and decides which actual addresses the CPU accesses when it reads and writes. Each of the smaller units (segmenataion and paging) can be turned on or off. Note that Paging can only be turned on if Segmentation is on. (We actually won't really use the Segmentation Unit in xv6. LATER EDIT: This is a lie. A LIE! We actually use it for all kinds of unwholesome tricks.)

Addresses coming from CPU are called **virtual/logical addresses**. Once through the Segmentation Unit, they're called **linear addresses**. Once through the Paging Unit, they're **physical addresses**.
In short: CPU [virtual] -> Segmentation [linear] -> Paging [physical] -> RAM

Here's a bunch of 16-bit registers that are used by the Segmentation Unit:

- `%cs` - **C**ode. This guy actually messes with our `%eip` register (that's the guy who points to the next command!).
- `%ds` - **D**ata. By default, messes with all registers except `%eip`, `%esp` and `%ebp`. (In assembly, we can override the choice of messing register.)
- `%ss` - **S**tack. By default, messes with `%esp` and `%ebp` registers.
- `%es`
- `%fs`
- `%gs`

The address `%eip` points to after going through `%cs` is written as `CS`:`EIP`.

In the Segmentation Unit there is a register called `%GDTR`. The kernel uses this to store the address of a table called GDT (Global Descriptor Table). Every row is 8 bytes, and row #0 is not used. It can have up to 8192 rows, but no more.

The first two parts of each row are **Base** and **Limit**.

When the Segmentation Unit receives and address, the CPU gives it an *index*. This index is written in one of the segmentation registers (`%cs`, `%ds`, ... `%gs`), thusly:

- Bits 0-1: permissions
- Bit 2: "use GDT or LDT?" (Let's pretend this doesn't exist, because it does not interest us at all.)
- Bits 4-15: The index

The Segmentation Unit receives a logical address and uses the index to look at the GDT. Then:

- If the logical address is greater than the `Limit`, crash!
- Else, the Segmentation adds the `Base` to the logical address, and out comes a linear address.

Note: In the past, the Segmentation Unit was used in order to make sure different processes had their own memory. For example, they could do this by making sure that each time the kernel changes a process, its segmentation regiesters would all point to an index that "belongs" to that process (and each row in the GDT would contain appropriate data). Another way this could be done would be by maintaining a seperate GDT for each process. Or maintaining a single row in the GDT and updating it each time we switch a process. There is no single correct way.

Note that all the addresses used by each process must be consecutive (along the physical memory).

In xv6, we don't want any of this.

Therefore, we will make sure that the GDT `Limit` is set to max possible, and the `Base` is set to 0.

In order to allow consecutive virtual addresses to be mapped to different areas in the physical memory, we use **paging**.

In the Paging Unit, there is a register named `%CR3` (Control Register 3), which points to the **physical** address of the Page Table (kinda like GDT). A row in the Page Table has a whole bunch of data, such as page address, "is valid", and some other friendly guys.

When the Paging Unit receives a linear address, it acts thusly:

- The left-side bits are used as an *index* (AKA "page number") for the Page Table. (There are 20 of these.)
- In the matching row, if the "is valid" bit = 0, crash!
- (There are also "permission" bits, but let's ignore them for now.)
- Those "page number" bits from the linear address are replaced by the actual page in our row (which is already *part* of the actual real live physical address)
- The page is "glued" to the right-side bits of the linear address (you know, those that aren't the page number. There are 12 of these.)
- Voila! We have in our hands a physical address.

Note that each page can be in a totally different place in the physical memory. The pages can be scattered (in page-sized chunks) all along the RAM.
Also note: Hardware demands that the 12 right-most bits of `%CR3` be 0. (If not, the hardware'll zero 'em itself.)

Uh oh:
Each row in the Page Table takes up 4KB (that's 12 bits).
The Page Table has 1024 rows.
4KB * 1024 = 4MB. That's 4 whole consecutive MBs. That's quite a large area in the memory, which kind of defeats the whole purpose of the Page Table. Well, not the *whole* purpose, but definitely some of it.

The solution: The Page Table gets its very own Page Table!

- First (small) page table contains - in each row - the (physical) address of another (small) page table.
- Each of the (small) 2nd-level page tables (which are scattered) contain actual page addresses.
- So: instead of 20 bits for single index, we have 10 bits for 1st-level table index and 10 bits for 2nd-level table index.

Thus, a virtual address can be broken down like so:
`[i0][i1][offset]`

- `i0` is the index of the First Table
- `i1` is the index of a second table
- `offset` is the offset

Addresses - double mapping

Every single physical address is mapped by the kernel to virtual address by adding KERNBASE to it.

When a process is given a virtual address, this new virtual address is *in addition* to the kernel mapping.

So when we want to get the physical address of a virtual address:

1. If the user-code is asking, it needs to access the page tables.
2. Is the kernel is asking, it can simply subtract KERNBASE from the address.

Likewise, the kernel can determine *its* virtual address of *any* physical address by adding KERNBASE.

KERNBASE = 0x80000000.

`1217 main`

In the beginning, we know that from `[0x0000 0000]` till `[0x0009 FFFF]` there are 640KB RAM.
From `[0x000A 0000]` till `[0x000F FFFF]` is the "I/O area" (384KB), which contains ROM and stuff we must not use (it belongs to the hardware).
From `[0x0010 0000]` (1MB) till `[0xFF00 0000]` (4GB - 1MB in total) there is, once again, usable RAM.
After that comes "I/O area 2".

(Why the 640KB, the break, and then the rest? Because in the olden days they thought no one would ever use more than 640KB.)

Address	Who uses this
`[Who cares]`	I/O
...	I/O
`[0xFF00 0001]`	I/O
`[0xFF00 0000]`	Usable RAM
...	Usable RAM
`[0x0010 0000]`	Usable RAM
`[0x000F FFFF]`	I/O
...	I/O
`[0x000A 0000]`	I/O
`[0x0009 FFFF]`	Usable RAM
...	Usable RAM
`[0x0000 0000]`	Usable RAM

Remember Mr. Boot? He loads xv6 to `[0x0010 0000]`.
By the time xv6 loads (and starts running `main`), we have the following setup:

1. `%esp` register is pointing to a stack with 4KB, for xv6's use. (That's not a lot.)
2. Segmentation Unit is ready, with a (temporaray) GDT that does no damage (first row inaccessible, and another two with 0 `Base` and max `Limit`.
3. Paging Unit is ready, with a temporary page table. The paging table works thusly:
   - Addresses from `[0x800- ----]` till `[0x803- ----]` are mapped to `[0x000- ----]` through `[0x003- ----]` repectively. (That is, the left-most bit is simply zeroed.)
   - ALL the the above addresses have 1000000000b as their 10 left-most bits, so our 1st-level page table has row 512 (that's 1000000000b) as "valid", and all the rest marked as "not valid".
   - Row 512 points to a single 2nd-level table.
   - The 2nd-level table uses all 1024 of its rows (that's exactly the next 10 bits of our virtual addresses), so they're all marked as valid.
   - Each row in 2nd-level table contains a value which - coincidentally - happens to be the exact same number as the row index.

Let's look at some sample virual address just to see how it all works:
`[0x8024 56AB]` -> `[1000 0000 0010 0100 0101 0110 1010 1011]` -> `[1000 0000 00` (that's row 512) `10 0100 0101` (that's row 581) `0110 1010 1011` (and that's the offset) `]` -> row 581 in the 2nd-level table will turn `[1000 0000 0010 0100 0101...]` to `[0000 0000 0010 0100 0101...]`, which is exactly according to the mapping rule we mentioned a few lines ago.

Available pages (free!)

Processes need pages to be mapped to. Obviously, we want to make sure that we keep track of which page are available.
The free pages are managed by a **linked list**. This list is held by a global variable named `kmem`. Each item is a `run` struct, which contains only a pointer to the next guy.

`kmem` also has a lock, which makes sure (we'll learn later how) that different processes don't work on the memory in the same time and mess it up. (An alternative would be to give each processor its own pages. However, that could cause some processors to run out of memory while another has spare. (There are ways to work around it.))

`main` calls `kinit1` and `kinit2`, which call `freerange`, which calls `kfree`.
In `kfree`, we perform the following 3 sanity checks:

- We're not in the middle of some page
- We're not trying to free part of the kernel
- We're not pushing beyond the edge of the physical memory

Building a page table

Let's examine what needs to be done (not necessarily in xv6) in order to make our very own paging table.

Our table should be able to "translate" virtual address *va* to physical address *vp* according to some rule.

Note that we'll be using `v2p`, which is a hard-coded mother-function that translates **virtual** addresses to **physical** by subtracting `KERNBASE` (which equals `0x8000 0000`) from the virtual addresses.

Step 1: Call `kalloc` and get a page for our First Table. (Save (virtual!) address of new table in `pgdir` variable)

Step 2: Call `memset` to clear entire page (thus marking all rows as invalid).

Step 3: Do the following for **every single *va*** we want to map:

- **Step 3.1**: Create and clear subtable, and save address in `pgtab` (similar to what we did in steps 1 and 2). (SEE NOTE AFTER THIS LIST)
- **Step 3.2**: Figure out i0 (index in `pgdir`) (using *va* & `v2p` function), write `pgtab` there, mark as valid.
- **Step 3.3**: Figure out i1 (index in `pgtab`) (using *va* & `v2p` function), write *pa* there, mark as valid.

Step 4: Set `%CR3` to point at `pgdir`, using `v2p`.

THE NOTE MENTIONED EARLIER: After we already have some subtables, we only need to create new subtables if the requested subtable does not exist yet.
How do we know whether it exists already? Simply by looking at the current i0 and seeing whether it's already marked as valid.

ANOTHER NOTE: What if two different *va*->*vp* rules clash? xv6 will crash (deliberately).

For more details about how the kernel builds its mapping tables, please refer to [xv6 Code Explained.md] (`kvmalloc`).

Moar GDT stuff!

Remember how we said we'd make sure GDT has `base=0` (so it won't alter addresses) and `limit=max` (so it won't interfere)?
Well, it turns out the hardware still uses the GDT to check various user-mode/kernel-mode stuff.

We need four rows in the GDT:

1. Kernel-mode, can only execute and read
2. Kernel-mode, can write
3. User-mode, can only execute and write
4. User-mode, can write.

There is a macro named `SEG`, which helps us build all these annoying bits.

Okay, that's enough of this.
Don't pretend you understand, because you don't and it doesn't matter.

Per-CPU variables

We've got an array of CPU data, `cpus`.

We can access specific CPU stuff via `cpus[SOME_CPU_IDENTIFIER]`.
In order to get current CPU identifier, we call `getcpu()`, which is slow.

When we do this, we *MUST* stop interrupts from happening, to make sure we stay within the same CPU.

Problem: Calling `getcpu()` is slow, and stopping (and then resuming) interrupts can be extremely slow.

Solution: Instead of using `getcpu()`, we can use a special register in each CPU!

Problem: Registers can be overriden by users.

Solution: Special register that only kernel can use!

Problem: *There are no such registers.*

Solution: Cheating, using the GDT!

So we have a list of GDTs, one per CPU.
When we initialize a CPU (just once, yeah?), we set up its very own GDT.
In this GDT, we add a *fifth* row (remember, we have four rows for kernel/user x read/write).
In this new row, we set the base to point at the place in the memory where the CPU data sits, and set the limit to 8 (because we have two important variables of CPU data, and each one take 4).

We do this in `seginit()`.

```
struct cpus *c = &cpus[getcpu()]; // costly, but used just once!

// Set first four rows...
c->gdt[SEG_KCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, 0);
c->gdt[SEG_KDATA]=SEG(STA_S, 0, 0xffffffff, 0);
c->gdt[SEG_UCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
c->gdt[SEG_UDATA]=SEG(STA_S, 0, 0xffffffff, DPL_USER);

// Set our special row
c->gdt[SEG_KCPU]=SEG(STA_W, &c->proc, 8, 0);

// Load the GDT
lgdt(c->gdt, sizeof(c->gdt));

// Set %gs register to point at our fifth row in the GDT.
loadgs(SEG_KCPU << 3); // shift 3 left so we get value and not permission bits.

// Declare out two variables,
// telling code generator to use gs register
// whenever we access proc or cpu
extern struct proc *proc asm("%gs:0");
extern struct proc *cpu asm("%gs:4");
```

```
proc = NULL;
cpu = c;
```

So how does this affect our accessing per-CPU data?
Normally, if we do `proc = 3;`, then it would be complied to `movl $3, proc`.
However, now that we did that weird `extern struct proc *proc asm("%gs:0");` part, it is compiled to `movl $3, %gs:0`.

Therefore, whenever in the code we call `proc` or `cpu`, we will automagically be referring to the address held in `GS:0` or `GS:4`, which happens to be the `proc` or `cpu` belonging to the specific CPU in which the code is executed.

Note that variable `cpu` points to the variables `cpu` and `proc`, so calling `proc` is the same as (yet faster than) calling `cpu->proc`.
Note also that this means that calling `cpu->cpu->cpu->cpu` is the same as calling `cpu`.

Important note: `GS` can be changed by code in user-mode.
So, every time there is an interrupt, we reload our value to `%gs` using `loadgs(SEG_KCPU << 3);`.
(Don't worry, we back up all user-mode's registers beforehand.)

Processes

Every process has a struct `proc` that holds a bunch of its data:

- `sz` - memory size
- `pgdir` - its very own page table
- `kstack` - small *kernel* stack (4KB), for data we don't want the process to touch and break (such as return address from system call)
- `state`
- `pid` - process ID
- `parent` - parent process
- `tf` - pointer to place where register values are saved during interrupt
- `context` = another interrupt thing
- `chan` - channel, marks what event process is waiting for (while sleeping). If none, then set to 0.
- `killed` - 0 if not killed yet
- `ofile` - vector of open files
- `cwd` - current working directory (like when doing `cd...` in command)
- `name`

A process can move through its processes thusly:

Embryo -> Runnable -> Running
Running -> Zombie
Running -> Sleeping (waiting for whatever event is marked in `chan`)
Running -> Runnable

The processes are held in a struct named `ptable`, who has a vector of proceses named `proc`.

Preparing the First Process

When running The First Process, we do the following in `userinit`:

- `allocproc`: allocate `proc` structure
    - Allocate process kernel stack
    - Set up data on kernel stack (in the "trapframe" area) *as if the process already existed and went into kernel mode*. As a result, the data will be loaded to proc in due time
- `setupkvm`: create page table (without knowing yet where to map user addresses (only kernel addresses))
- `inituvm`:
    - Allocate free page
    - Copy code to page
    - Update page table to map user memory
- Fix the data on the kernel stack for some reason.

Trapframe contains all register data.

When we return from an interrupt (or, in our case, start the process for the first time), we do the following:

1. Pop the first 4 values to their appropriate registers
2. Pop `eip` field to `%eip` register (now `%eip` register is pointing to `forkret` function)
3. Goto `forkret` function, which (somehow) takes us to the next guy in the stack...
4. ...Trapret which pops another whole bunch of values from the stack to the registers
5. And so on. We end up with all registers holding good values, and `%eip` pointing to 0.

The First Process

All it does it execute the program "init" (with the arguments {"/init", 0} (the 0 is to mark the end of the list)).

This is how the code *would* have looked like if it were written in c:

```
char *argv[] = {"/init", 0};
main() {
    exec(argv[0], argv);
    for (;;) exit();
}
```

...But it's not written in c.
It's written in assembly.

Here's the arguments code:

```
# these are the arguments:

init:
    .string "/init\0"

    .p2align 2
argv:
    .long init
    .long 0
```

...And here's the *code* code:

```
    .globl start
start:
    pushl   $argv ; push second argument
    pushl   $init ; push first argument
    pushl


  cli();
  something_dangerous();
  sti();



    movl    $SYS_exec, %eax ; load the canons (perpare to call "exec"
system-call)
    int $T_SYSCALL ; FIRE!

# once end up back here, we need to quit.
exit:
    movl    $SYS_exit, %eax
    int $T_SYSCALL
    jmp exit
```

So we have this code, but we need to load it during `inituvm`.
This is done in the script in the **initcode.S** file, which compiles that code to binary and shoves it all into the data section, with the exact same labels that are used in `inituvm`.

Actually running the First Process

This is done in `mpmain`, which calls the scheduler, which loops over process tables and runs processes.

Since at this point there's only one process at this point, it's the only one that'll run.

Accessing process's kernel stack during interrupt

When we're in user-mode and there's an interrupt, the kernel needs its own stack on the process to push real data onto it (before leaving the process).

We need this kernel-stack to be accessable only by the kernel.

The hardware supports this, with a magnificent Rube Goldberg machine:

- `%tr` register can only be accessed in kernel-mode
- `%tr` contains `SEG_TSS`, which points to special place in GDT
- special place in GDT contains address and size of some weird struct
- weird `TSS` struct contains field `esp0`, which actually (finally) points to top of process's kernel stack (and `ss0` with stack size)

So during `switchuvm` we set up this whole thing, so that in due time the kernel (and only the kernel) can access its stack on the proc.

Locks - the problem

Because we have multiple CPUs, we need to make sure two CPUs don't both grab the same data (such as the same `proc` struct during `allocproc`, or the same page during `kalloc`).
This could (theoretically) even happen with a single CPU, because of interrupts.

Locks - a possible solution (just for interrupts)

We can block interrupts (!) from happening by setting the `IF` bit in `%eflags` register to 0.
This can be controlled (in kernel-mode, yes?) using the assembly commands:

- `cli` (clears bit)
- `sti` (sets bit to 1)

(Also, the entire `eflags` register can be pushed into `eax` using some other assembly command.)

So each time we want to do something unsafe, we can do:

```
function b() {
    cli();
    // ...
    sti();
}

function a() {
    cli();
    b(); // at the end of this we set sti()!
    // ... Uh oh. We're open to interrupts!
    sti();
}
```

...But this is still dangerous, because of cases such as the following:

```
function b() {
    pushcli();
    // ...
    popcli();
}

function a() {
    pushcli();
    b(); // at the end of this we DO NOT set sti()
    // ... the counter is still 1
    popcli(); // NOW the counter became 0, so now we call cli().
}
```

Locks - an actual solution (just for interrupts)

Solution: Keep track of how many times we called `cli()`, so that `sti()` only releases if down to 0.
We can do this by wrapping all `cli()`s with `pushcli()` and `sti()`s with `popcli()`.
In `popcli()`, if our count reaches 0, we don't automatically call `sti()`, but revert to the **previous setting**.

```
lock ; xchgl mem, reg
```

(Note that each CPU runs `pushcli()` and `popcli()` separately for each CPU.)

Locks - an actual solution (for multiple CPUs)

There is a magical assembly word, `lock`.
Usage example:

```
old = xchg(&lock, b);
```

If `lock` is added to different commands run by different CPUs, then the `lock`ed commands will execute *serially*.

Using this, the following c call:

```
acquire(struct spinlock *lk) {
    pushcli(); // disable interrupts in this CPU, to avoid deadlocks
    while(xchg(&lk->locked, 1) != 0);
}
```

...Does the following:

1. Sets old = lock
2. Sets lock = b (but only if not locked by someone else!)

How on earth does this help us?

I'll tell you how!

We have a struct called `spinlock`, which holds all kinds of locking data.
In the function `acquire`, we do the following:

```
release (struct spinlock*lk) {
    xchg(&lk->locked, 0);
    popcli();
}
```

`xchg` will always return the *existing* value of `&lk->locked`, so if it's already locked we'll loop until it's released.

In `release`, we do the following:

```
pushl
subl , %esp ; empty word. Some weird convention
movl $SYS_sleep, %eax
int
```

How locks are managed during `scheduler`

When `scheduler` searches for RUNNABLE procs, it `acquire`s a lock on the process table.
This is in order to make sure that `scheduler`s on *other* CPUs don't grab the same process we just did.

The lock is held even as we switch to the process; it's the process's responsibility to release the lock (in order to allow interrupts).
Likewise, the process must re-lock the process table before returning to `scheduler`.

Each lock is re-released by the next process that runs, and then re-re-locked.

The final release is done by `scheduler` once there are no runnable processes left.

Sleep

When a process needs a resource in order to continue (such as disk, input or whatever), it must call `sleep` where it marks itself (`proc->chan`) as waiting for the wanted event, sets its `state` to SLEEPING, and returns to `scheduler`.

When the Event Manager (whom we never heard of yet) sees that the event occurs, it marks the proc as RUNNABLE once again.

Note that the process needs to see that its resource is still available (and hasn't been make unavailable again by the time it woke up) before continuing. If the resource is unavailable again, it should go back to sleep (if it knows what's good for it).

`sleep` also demands a locked `spinlock` for a weird reason we don't know yet.
It replaces the locking from *that* lock to the process table (unless they are the same lock, of course).

Once done, it releases the process table and re-locks the supplied `spinlock`.
We'll get back to you folks on this one once we understand why on earth this is needed.

Interrupts - CPU

There are two basic types of interrupts: internal (thrown by CPU) and external (like keyboard strokes).

We can decide whether to listen to external interrupts using kernel-mode functions `sti()` and `cli()` which sets or clears the `IF` bit on the `%eflags` register.
We cannot ignore internal interrupts.

Interrupts can only occur *between* commands.

During an interrupt, the CPU needs to know what kind of interrupt it is. These range between 0-255, when 0-31 are reserved by Intel for all kinds of hardware stuff.

In order to handle these interrupts, the CPU needs a table with pointers to functions that handle 'em (with the index being the interupt number).
This table is called IDT, and the register that points to it is `%IDTR`.

Each row has a whole descriptor of 64 bits. These include:

- `offset` - the actual address of the function (this guy is loaded to `%eip`)
- `selector` - loaded to `%cs`
- `type` - 1 for trap gate, 0 for interrupt gate (if interrupt, we disable other interrupts)

Before an interrupt call, the CPU must do a bunch of stuff to rescue our registers before they are overriden by the IDT guy.
Reminder: the kernel stack's address is saved in a `tss` structure, which is saved in the GDT, in the index held by `%tr` register.
Therefore: the kernel needs to set this before an iterrupt occurs.
SO: The CPU uses the kernel stack to store our registers.
(And after the interrupt handling is over, the CPU needs to pop these guys again. It does this with assembly `iret` command.)

If the interrups occurs during **kernel-mode**, the CPU performs the following:

- push `%eflags`,`%cs` and `%eip` to the **kernel** stack (so we'll have them again after the interrupt).

If the interrups occurs during **user-mode**, the CPU performs the following:

- Store `%ss` in `%ss3`
- Store `$esp` in `%esp3`
- Store `tss.ss0` in `%ss`
- Store `tss.esp0` in `%esp`
- Push `%ss3`, `%esp3`, `%eflags`, `%cs` and `%eip` to **kernel** stack

Question: Since we push a different number of registers for kernel-mode and user-mode, how do we know how many to pop back out?
Answer: In both cases, we need to first pop `%eip`, `%cs` and `%eflags`.
Having done that, we can look at the two right-most bits of `%cs` to see whether we were in user- or kernel-mode before the interrupt! Hurray!

Interrups - xv6

In xv6, all interrups are handled by a *single* C function `trap`.

However:

1. This function needs to end with `iret` (that's the guy who handles all the register poppin' at the end of the call), while C functions end with `ret`!
2. Different interrupts need to send different data to the function (such as interrupt number, error number (which not every interrupt needs)).

In order to accomplish this, we wrap the call to `trap` in assembly code that does the following:

1. Push registers to kernel-stack (except those which were already pushed by CPU (see previous section))
2. Push address of kernel-stack to kernel-stack (so the address is sent as a parameter to `trap`; I'll elaborate more on this in a moment)
3. Call `trap`
4. Pop the registers
5. Do `iret`

The function `trap` receives a pointer to a `trapframe` struct, which is actually the kernel-stack.
The fields in `trapframe` are the actual register values, so `trap` can use them in order to determine all kinds of stuff (such as "were we in user-mode when the interrupt occurred?"). Additionally, we have the interrupt number and error-code (where applicable).

The assembly code that wraps the call to trap is mostly the same for each interrupt, with just the slightest difference:
Which interrupt number to push, and do we push an error-number (and if so, which).

So, the shared code is under a label `alltraps` (which, as it says on the label, is for all traps).
Each interrupt does its own little thing and then jumps to `alltraps`.

The code parts for the different interrupts are labelled `vector0`, `vector1`, ... `vector255`, and are generated by a script (in verctors.pl file).
After generating the vectors, the script creates an array (called - surprise, surprise - "vectors").
Later, during `main`, we call `tvinit` which loops over the vectors array and writes the data in the IDT table.

syscall

When user-code calls a system-call, the system-call number is stored in `%eax`.

So in `syscall`, we can access the system-call number via `proc->tf->eax`.

After running the system-call, we put the return value in `proc->tf->eax`; this will cause `alltraps` to pop it back to `%eax`, and then the user-code will know whether the system-call succeeded.

The system-call numbers are saved as constants in syscall.h file.

Then we have a vector that maps each constant to its appropriate function. This vector is used in `syscall` function in order to execute the system-call.

`sleep` *system-call*

There is a global variable called `ticks`.

Whenever there is a timer interrupt occurs, `ticks` increments by 1.

We use the *address* of `ticks` as the **event number** for the timer.

So...
When a process calls `sleep`, it sets its channel to equal `&ticks`, and stores the current value of `ticks`.
Whenever a timer event occurs, if the current value of `ticks` minus the value of `ticks` that the process stored *reaches* the number of ticks the process asked to sleep for, it becomes RUNNABLE.

Here's a sample of what a call to sleep looks like in assembly (for 10 ticks) :

```
long *a (long*)b->data;
for (int i = 0; i < 128; i++)
    outl(0x1f0, a[i]);
```

Getting arguments in system-calls

In the example of `sleep`, the argument (how many ticks to wait) is stored on the **user**-stack.

This is accessed via `proc->tf->esp + 4` (the +4 is to skip the empty word, which is on the stack because of some weird convension). We do this in `argint` function.

Because this the data is placed on the stack by *user code* (which is pointed at by `%esp`), we need to check that the user didn't set `%esp` to point at some invalid address! We do this in `fetchint` function.

fork() #2

When we `fork` a proc, we need to copy a bunch of its data.
We want to map *new* physical addresses, but copy the old data.
We want to create a *new* kernel stack.
We want to copy context.
We want the state to be RUNNABLE (and not RUNNING like parent).

We want a new `pid`.

A lot of this stuff is already handled by `allocproc`.

The memory stuff is handled by `copyuvm`.

exec() #2

`exec` replaces the current process with new one.

Because this may fail, we don't want to destroy the current memory data until we know we succeeded.
In order to keep the current memory until the end, we need to create a *new* memory mapping, and only switch to it (and destroy the old mapping) at the very end.

The four stages of `exec`:

1. Load file
2. Allocate user stack (and a guard page to protected data from being overwritten by stack)
3. Pass argv[] arguments
4. Fix trapframe and switch context

In order for `exec` to run the file it's told to run, the file needs to be in ELF format.

ELF

Executable and Linkable Format.

In xv6 we only use *static* ELF, but in real life there are also *dynamic* ones but we don't care about that here.

ELF file have:

1. ELF header at the very start
2. `proghdr` vector
3. Program sections

The program sections include the actual code, as well as external code linked by the linker.
There can be many program sections, scattered along different parts of the file (but not at the beginning).
Each program section needs a *program header*, which says where it is in the file and where it is in the RAM.
All the *program headers* are held in the `proghdr` vector.
The location of `proghdr` is held in the ELF header.

ELFHDR (that's our ELF header) has a bunch of fields, but we'll only look here at a few:

- `uint magic` - must equal `ELF_MAGIC`. Just an assurance all's good.
- `ushort machine` - the type of processor. xv6 actually doesn't pay attention to this, so I don't know

why we bothered mentioning this one.
- `uint entry` - virtual address of `main`
- `uint phoff` - PH offset, location of `proghdr`
- `uint phnum` - PH number, length of `proghdr`
- `uint flags` - uh... flags.

Since the ELF is created by user-code, the kernel doesn't trust it.
The kernel treats the data with caution, and fails the `exec` if there are any errors.

`proghdr` has the following important fields in each vector:

- `uint off` - where section starts in file
- `uint filesz` - where sections ends in file (so last byte is in `off+filesz-1`)
- `uint vaddr` - virtual addres to which section is loaded
- `uint memsz` - size section takes in memory. May be larger than `filesz`, because section may include lots of zeros (but not the other way around).

The kernel needs to loop and copy the memory, allocating new memory if either 1) we need more memory or 2) `vaddr` is beyond what is already allocated. This is done using `allocuvm`.

`exec` *- guard page*

After `exec` loads da codes, it allocates another page for the user-stack.

Problem: Stack goes to *lower* addresses as it fills up; eventually, it'll overrun the code-n-data!
Solution: Add guard page, with user-mode bit cleared. That way, user-code will get an error if StackOverflow.

`exec` *- pushing arguments to new process*

New process expects the following to be on the user-stack:

variable	what it holds
`argv`	pointer to vector of pointers to string arguments (with 0 in last place)
`argc`	number of arguments
	| return address

...And this is how `exec` fills the user-stack:

user-stack
argument
...
argument
0
pointer to argument
...
pointer to argument

user-stack
`argv` (pointing to this □ guy)
`argc`
-1

Note that the `pointers to arguments` don't really need to be on the stack; we just put 'em there because we have to put them *somewhere* so why not.

Note also that we don't know in advance how many arguments there are and how long they'll be. Therefore, our code must first place the arguments stuff, and only afterwards copy `argc`, `argv` and return address.
Therefore, we save `argc` & `argv` & return address to a temporary variable `ustack`, which we copy to actual stack at the end of the argument pushing.

I/O

xv6 supplies the following system-call functions for files:

- `open(char *name, int flag)` - opens file and returns file descriptor (i.e. index in file array)
- `close(int fd)` - closes file
- `pipe(int pipefd[2])` - kinda like a file, for sending messages between processes
- `dup(int fd)` - duplicates opening to file (so if we move the cursor/offset of one (such as by reading), it'll affect the other)
- `read`
- `write`
- `stat`

Reading/writing can be much different if we're dealing with keyboard, file, etc., but xv6 strives to make them "act" the same.
We've got the following read/writables:

- pipe
- inode
  - file
  - directory
  - device

Each has its own internal functions for reading, writing, etc., but - as mentioned above - xv6 wraps 'em and hides 'em from user code.

xv6 has an abstract struct `file`, which it uses for all the above read/writables.
xv6 has functions that deal with this `file`, but they accept a *pointer* instead of a file descriptor.
Why? Because:

- xv6 needs a pointer to an actual struct, because it needs to handle data.
- User code isn't trusted to handle pointers; it gives us a file descriptor, and the *kernel* accesses the

actual pointer.

So how to we maintain this translation between file descriptors and pointers?
With an array "ofile", that's how.

These `file` guys have basic data that all read/writables contains, such as "type".
Inodes implement their "inheritance" by having their "type" be "FD_INODE", and containing a pointer (in "inode" field) to a `struct inode` that has its very own data (such as "type", which could be "T_FILE" for a file or "T_DEV" for device).

OK, this explanation is pretty lame :(
You should really just see the "I/O" presentation on Carmi's site, which he changes every semester.
Currently, the presentation is at
http://www2.mta.ac.il/~carmi/Teaching/2016-7A-OS/Slides/Lec%20io.pdf (slides 13-23).

Pipe stuff

A `pipe` is pointed to by a *reader* `file struct` and a *writer* `file struct`.
(Both of which exist *once*, and may be pointed to by many different procs.)

The `pipe` has `readopen` and `writeopen` fields, which contain "1" as long as the *reader* and *writer* guys are still holding him open.
When both are "0", the `pipe` is closed.

Moar file stuff

The kernel has an array `ftable`, which holds all the open files (and a lock to make sure two procs don't fight over a slot in the array).

Files are added to `ftable` in the function `filealloc`.

Transactions

When writing to blocks on disk (or deleting), we really don't want the machine to crash or restart, or else there'll be corrupted data.

In order to alleviate our fears, we can wrap our *writing* in a *transaction* (by calling `begin_trans()` before and `commit_trans()` after).
A transaction ensures that the write will either be *complete* or won't be at all.

A transaction is limited to a certain size, so during `writefile` we need to divide the data to managable chunks.
Therefore, if there's a crash in the middle of the write operation, some blocks might be written and the rest not. However, each block will be fully-written or not written at all (but no hybrid mish-mash).

inode Stuff

xv6 considers a directory to be a file like any other.
Every `struct dirent` has an inode number, and a name. (If the directory is not in use, the number is 0.)

The inodes are managed by the inode layer, which we will talk about later.

`struct inode` has the following amongst its fields:

- `uint dev` - device number
- `uint inum` - inode number
- `int ref` - reference count
- `int flags` - flags, such as 0x1 for I_BUSY (that is, *locked*) and 0x2 for I_VALID
- `short nlink` - the number of names ('links') the inode has on the disk (kinda like a shortcut in Windows)
- `uint size` - the size of the inode on the disk

In order to open an inode, we have `namei()`.
In order to search for an inode, we have `dirlookup()`.

Note that `dirlookup` can be supplied with an optional pointer `poff` that gets the offset of the found inode.
Why?
In case of renaming or deleting, we'll need to update the row in our parent inode. (In case of name change, change the num; in case of deletion, change `inum` to 0.)

So... when we open an inode, we need to supply the inode to search in.
Conveniently, we actually *don't* supply an inode pointer, because we have THE CURRENT WORKING DIRECTORY which is "supplied" automatically.

...Actually, that's not exactly true.
It *is* true if we're looking for some "a/b/c" path, but **not** for "/a/b/c".
In the latter case, we use the **root inode** instead of the current working directory.

As you may or may not have guessed, if we have a path with a few parts (such as "a/b/c/d"), we need to loop over the parts and for each part find the matching inode.

In order to split the path into parts, we use `skipelem`.

inode Stuff - The Inode Layer

The inode layer supplies a bunch of funcs:

- `iget` - open
- `iput` - close
- `idup` - increments ref count

- `ilock` - must lock inode whenever we want to access any field
- `iunlock` - unlocks inode
- `ialloc` - allocates inode both on disk and in memory
- `iupdate` - update disk with whatever chanes we made to inode
- `readi`
- `writei`

Our inode structs are saved in `icache`.
They are saved there the first time they are opened (but not before; they are loaded on-demand).

The inodes are added to `icache` in `iget`.
When added, we add them *without the actual data from the disk*.
Why?
Because often we call `iget` just to see if the inode exists, and actually reading from the disk is expensive (so we do it just if we need it).

The Disk - Reading and Writing

The disk in divided to blocks.
Every block on our disk is of 512 bytes; no more, no less.
Reading and writing to and from a block is done *just* in the beginning of a block.

So how do we write (or read) only a few bytes?
We have a buffer layer that takes a whole block to memory, writes (or reads) our few bytes *in memory*, then - if we're writing - re-writes the entire block to the disk.

The buffer layer contains the functions:

- `readsb` - reads super-block. Never mind.
- `bread(dev, sector)` reads an entire block. (function returns a `struct buf` which contains a field `data` with the actual 512 bytes of data)
- `log_write`
- `brlse`
- `balloc`
- `IBLOCK` - we'll explain in a moment:

Remember `struct inode`?
Well, there's also `struct dinode` which represents an inode *on the disk*.
It's similar to the inode, but without the meta-data (such as ref, inum, etc.).

`IBLOCK` is a macro that gives us the block number of an inode.
In order to tell *where* within the block is our inode, we can calculate (inum % IPB). IPB is the number of inodes in a block.

ilock

As mentioned before, before accessing inode data we need to `ilock` it.

We don't want to use our regular locks, because they:

- disable interrupts
- when trying to acquire, "spin" over lock till lock is open

This wastes a lot of time.

`ilock` is a *soft* lock, that doesn't do this stuff.
We don't disable interrupts, and instead of spinning, we do `sleep` (so we're not hogging CPU while waiting for acquiring).

Reading from the disk

Every inode on the disk has a vector of `addrs`.
Every entry points at a single block on the disk.

The first 12 entries are *direct* pointers.
The 13th points to a block that serves as vector of pointers.
That is, the last pointer is a pointer to pointers.

It turns out there's actually a reason for having the inode data be partially direct and partially indirect:

- We have the direct blocks because reading from the indirect blocks cost an extra read for each block.
- We have the indirect blocks because the direct blocks are a waste of space for small files.

The buffer layer

The buffer layer supplies the following functions:

- `bread`
- `bwrite`
- `balloc`
- `bfree`

`struct buf` has the following fields:

- `flags` - such as BUSY or DIRTY
- `dev`
- `sector` - device and block num
- `struct buf * prev` - for linked list
- `struct buf * next` - for linked list
- `struct bef * qnext`
- `uchar data[512]`

All our buffers are kept in a cache, `bcache`.
This cache contains a spinlock (obviously), and a linked list of buffers (held in field `head`).

The linked list of buffers is maintained so that the most recently used is in the head, and the least in the tail.

Buffs are never removed from the cache!
If they're not used, they're marked as *not* busy and *not* dirty.
If we need a new buff (such as calling `bget` for a block that hasn't been read yet), we find a `struct buff` at the end of the list that is neither BUSY nor DIRTY, and mark it BUSY.

When we call `brelse` and release a buffer, it is moved to the head of the list (and it's BUSY flag is turned off).

`bcache` is initialized in `binit()`.

The superblock and the bitmap

There is a block on the disk called the *superblock*.
This block is located right after the boot block, and contains meta data regarding the other blocks.

There is another area on the disk called the *bitmap*.
For every block on the disk, there is a bit that marks whether the block is free or not.

The driver layer

Ah, the driver layer. The guy who actually does all the dirty work.

In order to actually access the disk (as well as other peripherals, such as keyboard), we have controllers.
Our guy is the IDE.

The IDE has a bunch of *ports*, which are actually numbers (but "port" sounds a lot more computer-y that "number").
The processor can send/receive values to/from these ports using IN/OUT commands.
The IDE decides what to do for different values it is given to different ports.

If we want the IDE to read (or write) from the disk, we need to tell it:

1. Which disk to read from (turns out there are two)
2. Where on the disk to read (28 bits to tell us which block)
3. How many blocks to read

Each of these needs to be sent to different OUT ports, before we send the command "read" to the port that accepts the command:

- 0x1f2 - number of blocks
- 0x1f3, 0x1f4, 0x1f5 and 0x1f6 - block number (28 bits split to 8 bits per port (0x1f6 gets only 3 bits

  and device number))
- 0x1f7 - the command

The IDE can only handle one single command at a time; so before we do anything, we need to check its IN 0x1f7 port (the STATUS port).
The left-most bit tells us if the IDE is busy (so shouldn't use).
The second bit tells us if the IDE is ready (so can use).

The IDE has its very own RAM attached to it, which it uses to read/write.
(NOTE: We don't know how much is in this RAM. It depends on the controller we chose to put in our PC.)

So how do we access this RAM from our code?

Accessing this RAM from our code

In order to do this, we use OUT port 0x1f0.
For example:

```
long *a (long*)b->data;
for (int i = 0; i < 128; i++)
    a[i] = inl(0x1f0);
```

Although it looks like every `long` is being written to the same place, it isn't.

Same goes for reading:

```
long *a (long*)b->data; for (int i = 0; i < 128; i++) a[i] = inl(0x1f0);
```

Another useful parameter to send is `0` to port 0x3f6.
This tells the IDE to raise an interrupt when it finishes the command.
(Otherwise, the kernel can't tell when the read/write from/to the disk is complete.)

Functions provided by the driver layer

- `idewait` - loops over IDE port 01xf7 until status is READY
- `idestart`- *starts* requesting to read/write (whether to read or write depends on input)
- `iderw` - the *real* read/write function. Handles a queue of blocks to write, because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk.

In order to handle the queue, the buffer layer has a variable `idequeue`.
This is a linked list of bufs, which are actually read/written by `ideint`, which is run whenever there is an interrupt from IDE due to IDE finishing its previous command.

- `ideint` - handles "I'm finished!" interrupt from IDU and managed `idequeue`.