Introduction to UNIX System Calls

Why OS?

Computer systems are layered as abstractions: transistors, gates, functional units (adders / multipliers), ISAs, high-level languages, higher-level languages. Transistors allow large designs, because of their composable nature without loss of signal quality. For example, a typical computer has billions of transistors. However, programming these large designs require abstractions. Abstraction design is guided by tradeoffs in efficiency, ease-of-use, security, reliability, etc.

OS is one such abstraction; actually, it provides several abstractions for different parts of the system. Consider a multi-processing system running an editor, a browser, a window manager, and so on. This system can be written as one large event-driven loop. However, that would be extremely complex to manage, as it would need to implement diverse functionalities, yet interoperate cleanly. Instead, an OS provides abstractions, whereby these different functionalities can be implemented as different programs, and run as different *processes*. The OS implements services as *system calls*, which a process can invoke. The OS also ensures safety among these processes. This course is about the design of these abstractions/services that OS provides, their resulting tradeoffs, and the implementation of these abstractions on modern ISAs.

One of the must successful set of abstractions, were the ones used in the UNIX operating system in the 1970s. Unlike its predecessors (e.g., DOS), UNIX was a multi-processing operating system, i.e., multiple processes could co-exist at the same time. Most production operating systems today are based on UNIX abstractions. We will begin by studying the UNIX abstractions.

Outline

System Calls: fork, exec, exit, wait, open, read, write, close
Case Study: Unix/xv6 shell (simplified)

System Calls

"Kernel functions" that perform privileged operations on behalf of the process. As an OS designer, one of the goals is to minimize the system call interface.

Case study: Unix/xv6 shell (simplified)

provides an interactive command execution and programming language
typically handles login session, runs other processes
look at some simple examples of shell operations, how they use different OS abstractions, and how those abstractions fit together. See Unix paper if you are unfamiliar with the shell.
Basic structure:

```
        while (1) {
            write (1, "$ ", 2);             // 1 = STDOUT_FILENO
            readcommand (0, command, args);  // parse user input, 0 = STDIN_FILENO
            if ((pid = fork ()) == 0) {      // child?
                exec (command, args, 0);
            } else if (pid > 0) {            // parent?
                wait (0);                    // wait for child to terminate
            } else {
                perror ("Failed to fork\n");
            }
        }
```

system calls: `read`, `write`, `fork`, `exec`, `wait`. conventions: -1 return value signals error, error code stored in `errno`, `perror` prints out a descriptive error message based on `errno`.

What's the shell doing? fork, exec, wait: process diagram (PID, address space -- memory of the process, parent links). fork returns twice, in some sense!

why call "wait"? to wait for the child to terminate and collect its exit status. (if child finishes, child becomes a zombie until parent calls wait.)

Example:

```
        $ ls
```

how does ls know which directory to look at? (cwd variable is copied during fork)
how does it know what to do with its output? (fd[1] is initialized by the shell)
I/O: process has file descriptors, numbered starting from 0.
system calls: open, read, write, close
numbering conventions:
file descriptor 0 for input (e.g., keyboard). read_command:

```
read (0, buf, bufsize)
```

file descriptor 1 for output (e.g., terminal)

```
write (1, "hello\n", strlen("hello\n"))
```

file descriptor 2 for error (e.g., terminal)
on fork, child inherits open file descriptors from parent (show in process diagram).
on exec, process retains file descriptors, except those specifically marked as close-on-exec: `fcntl(fd, F_SETFD, FD_CLOEXEC)`
How does the shell implement:

```
$ ls > tmp1
```

just before exec insert:

```
    close(1);
    creat("tmp1", 0666);   // fd will be 1
```

The kernel always uses the first free file descriptor, 1 in this case. Could use `dup2()` to clone a file descriptor to a new number.
Good illustration for why fork + exec vs. CreateProcess on Windows. (CreateProcess takes 10 arguments.)

Outline

System Calls: fork, exec, exit, wait, open, read, write, close, dup, pipe
Case Study: Unix/xv6 shell (simplified)

System Calls

UNIX's `fork()` and `exec()` versus Windows' `CreateProcess()`
UNIX process inheritance and file sharing

Case study: Unix/xv6 shell (simplified)

```
while (1) {
    write (1, "$ ", 2);                 // 1 = STDOUT_FILENO
    readcommand (0, command, args);     // parse user input, 0 = STDIN_FILENO
    if ((pid = fork ()) == 0) {         // child?
        exec (command, args, 0);
    } else if (pid > 0) {               // parent?
        wait (0);                       // wait for child to terminate
    } else {
        perror ("Failed to fork\n");
    }
}
```

why call "wait"? to wait for the child to terminate and collect its exit status. (if child finishes, child becomes a zombie until parent calls wait.). Zombies are required as a parent may need the exit status later. What happens if the parent exits before the child? Attach to init process.
Example:

```
$ ls
```

How does the shell implement:

```
$ ls > tmp1
```

just before exec insert:

```
close(1);
creat("tmp1", 0666);   // fd will be 1
```

The kernel always uses the first free file descriptor, 1 in this case. Could use `dup2()` to clone a file descriptor to a new number.
Good illustration for why fork + exec vs. CreateProcess on Windows. (CreateProcess takes 10 arguments.)
The split of process creation into fork and exec turns out to have been an inspired choice, though that might not have been clear at the time; see today's assigned paper.
What if you run the shell itself with redirection?

```
$ sh < script > tmp1
```

If for example the file `script` contains

```
echo one
echo two
```

FD inheritance makes this work well.
Discuss how the same interface can be used for both devices and files; and how the OS now needs to keep track of the file-offset internally due to this interface.
Discuss how the offset is a property of the resource, and not of the file-descriptor. Being a property of the resource allows sharing of offsets (thus producing interleaving effects). Separate offsets would preclude interleaving completely. On the other hand, separate offsets can still be created with the current abstractions.
What if we want to redirect multiple FDs (stdout, stderr) for programs that print to both?

```
$ ls f1 f2 nonexistant-f3 > tmp1 2>&1
```

after creat, insert:

```
        close(2);
        creat("tmp1", 0666);   // fd will be 2
```

why is this bad? illustrate what's going on with file descriptors. better:

```
        close(2);
        dup(1);                // fd will be 2
```

(Read Section 3 of "Advanced Programming in the UNIX environment" by W. R. Stevens, esp. Section 3.10) or in bourne shell syntax,

```
    $ ls f1 f2 nonexistant-f3 > tmp1 2>&1
```

Read Chapter 3 of *Advanced Programming in the UNIX Environment* by W. Richard Stevens for a detailed understanding of how file descriptors are implemented. In particular, read Section 3.10 to understand how file sharing works.
how to run a series of programs on some data?

```
        $ sort < file.txt > tmp1
        $ uniq tmp1 > tmp2
        $ wc tmp2
        $ rm tmp1 tmp2
```

can be more concisely done as:

```
        $ sort < file.txt | uniq | wc
```

Draw a figure with boxes as programs, and each box containing two ports (STDIN/0 and STDOUT/1). Advantages of second option: no temporary space, no extra reads/writes to the disk, scheduling visibility to OS
A pipe is a one-way communication channel. Here is a simple example:

```
        int fdarray[2];
        char buf[512];
        int n;

        pipe(fdarray);
        write(fdarray[1], "hello", 5);
        n = read(fdarray[0], buf, sizeof(buf));
        // buf[] now contains 'h', 'e', 'l', 'l', 'o'
```

file descriptors are inherited across `fork()`, so this also works:

```
        int fdarray[2];
        char buf[512];
        int n, pid;

        pipe(fdarray);
        pid = fork();
        if(pid > 0){
          write(fdarray[1], "hello", 5);
        } else {
          n = read(fdarray[0], buf, sizeof(buf));
        }
```

How does the shell implement pipelines (i.e., cmd 1 | cmd 2 |..)? We want to arrange that the output of cmd 1 is the input of cmd 2. The way to achieve this goal is to manipulate stdout and stdin.
The shell creates processes for each command in the pipeline, hooks up their stdin and stdout, and waits for the last process of the pipeline to exit. Here's a sketch of what the shell does, in the child process of the `fork()` we already have, to set up a pipe:

```
int fdarray[2];

if (pipe(fdarray) < 0) panic ("error");
if ((pid = fork ()) == 0) {  child (left end of pipe)
   close (1);
   tmp = dup (fdarray[1]);   // fdarray[1] is the write end, tmp will be 1
   close (fdarray[0]);       // close read end
   close (fdarray[1]);       // close fdarray[1]
   exec (command1, args1, 0);
} else if (pid > 0) {        // parent (right end of pipe)
   close (0);
   tmp = dup (fdarray[0]);   // fdarray[0] is the read end, tmp will be 0
   close (fdarray[0]);
   close (fdarray[1]);       // close write end
   exec (command2, args2, 0);
} else {
   printf ("Unable to fork\n");
}
```

Who waits for whom? (draw a tree of processes)
Why close read-end and write-end? ensure that every process starts with 3 file descriptors, and that reading from the pipe returns end of file after the first command exits.

Outline

UNIX Signals: kill, signal system calls
Kernel-level Threads
User-level Threads
Intro to Concurrency and Locks

UNIX Signals

Signals are a limited form of inter-process communication
When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver a signal.
If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.
Examples:
Ctrl-C sends SIGINT; by default, this causes the process to terminate
Ctrl-Z sends SIGTSTP; by default, this causes the process to suspend execution
SIGCHLD signal is sent to a process when a child process terminates.
SIGFPE is the floating point exception (for example, on divide by zero)
SIGSEGV on segmentation fault
SIGUSR1 and SIGUSR2 are user-defined signals
`kill(pid, signum)`: send signal SIG to PID.
`signal(signum, void (*handler)(int))`: associates HANDLER with signal SIGNUM.

Shell Backgrounding

How do you create a background job?

```
        $ compute &
```

How does the shell implement "&", backgrounding? (Don't call wait immediately).
Q: What if a background process exits while sh waits for a foreground process?
Think about how a shell implements the following functionalities.
Lists of commands, separated by ";" (e.g., touch a; ls)
Sub shells implemented using "(" and ")". For example, a subshell can be used to setup a "dedicated environment" for a command group:

```
  COMMAND1;
  COMMAND2;
  (
    PATH=/bin
    COMMAND3
    COMMAND4
  );
  COMMAND5
```

Interesting uses of open/read/write/close

Linux has a nice representation of a process and its FDs, under /proc/PID/
maps: VA range, perms (p=private, s=shared), offset, dev, inode, pathname
fd: symlinks to files pointed to by each fd. (what's missing in this representation?)
can do fd manipulation in shell and see it reflected in /proc/$$/fd
can read or write kernel parameters using read() and write() syscalls!

Threads

Figure on process address space: code, static data, stack, heap. On fork, the whole address space gets replicated. On thread create, the created thread has a different program counter, registers, and stack (through stack pointer). Everything else is shared between threads.

Kernel-level threads are just processes minus separate address spaces. Discuss the kernel scheduler which is invoked at every timer interrupt. Each thread is an independent entity for the kernel.

Write the `cswitch` function for processes and threads. Notice that switching among threads requires no privileged operations. Switching the stack can be done by switching the `sp` register. A cswitch needs to be fast (typically a few 100 microseconds).

Advantages of threads over processes

Much more lightweight than processes. Faster creation, deletion, switching.

Much faster communication among threads: allow shared data structures to be maintained.

User-level threads can be implemented inside a process by writing `scheduler()` and `cswitch()` functions. The scheduler can be called periodically using SIGALRM signal.

Pros of user-level threads:

Lighter-weight

Faster cswitch

Do not need kernel's permission

Cons of user-level threads:

Will not be scheduled on different cores, because they look like one process to the kernel.

If one thread blocks (e.g., on I/O), all threads block.

Threading models (slide)

PC Architecture and Processor Setup

Outline

PC architecture
x86 instruction set

PC architecture

A full PC has:
one or more x86 CPUs, each containing:
integer registers (can you name them?) and execution unit
floating-point/vector registers and execution unit(s)
memory management unit (MMU)
multiprocessor/multicore: local interrupt controller (APIC)
memory
disk (IDE, SCSI, USB)
keyboard
display
other resources: BIOS ROM, clock, ...
We will start with the original 16-bit 8086 CPU (1978)
CPU runs instructions:

```
for(;;){
        run next instruction
}
```

Draw figure with common bus, I/O, and CPU. The CPU has registers, cache, etc.
Draw figure showing EIP and how it gets incremented automatically after executing each instruction.
Needs work space: registers
four 16-bit data registers: AX, BX, CX, DX
each in two 8-bit halves, e.g. AH and AL
very fast, very few
More work space: memory
CPU sends out address on address lines (wires, one bit per wire)
Data comes back on data lines
or data is written to data lines
Add address registers: pointers into memory
SP - stack pointer
BP - frame base pointer
SI - source index
DI - destination index
Instructions are in memory too!
IP - instruction pointer (PC on PDP-11, everything else)
increment after running each instruction
can be modified by CALL, RET, JMP, conditional jumps
Want conditional jumps
FLAGS - various condition codes
whether last arithmetic operation overflowed
... was positive/negative
... was [not] zero
... carry/borrow on add/subtract
... etc.
whether interrupts are enabled
direction of data copy instructions
JP, JN, J[N]Z, J[N]C, J[N]O ...

What if we want to use more than 2^16 bytes of memory?
8086 has 20-bit physical addresses, can have 1 Meg RAM
the extra four bits usually come from a 16-bit "segment register":
CS - code segment, for fetches via IP
SS - stack segment, for load/store via SP and BP
DS - data segment, for load/store via other registers
ES - another data segment, destination for string operations
virtual to physical translation: pa = va + seg*16
e.g. set CS = 4096 to execute starting at 65536
tricky: can't use the 16-bit address of a stack variable as a pointer
a *far pointer* includes full segment:offset (16 + 16 bits)
tricky: pointer arithmetic and array indexing across segment boundaries
But 8086's 16-bit addresses and data were still painfully small, so 80386 added support for 32-bit data and addresses (1985)
boots in 16-bit mode, bootasm.S switches to 32-bit mode
registers are 32 bits wide, called EAX rather than AX
operands and addresses that were 16-bit became 32-bit in 32-bit mode, e.g. ADD does 32-bit arithmetic
prefixes 0x66/0x67 toggle between 16-bit and 32-bit operands and addresses: in 32-bit mode, MOVW is expressed as 0x66 MOVW
the .code32 in bootasm.S tells assembler to generate 0x66 for e.g. MOVW
80386 also changed segments and added paged memory...
Example instruction encoding

```
	b8 cd ab		16-bit CPU,  AX <- 0xabcd
	b8 34 12 cd ab		32-bit CPU, EAX <- 0xabcd1234
	66 b8 cd ab		32-bit CPU,  AX <- 0xabcd
```

...and even 32 bits eventually wasn't enough, so AMD added support for 64-bit data addresses (1999)
registers are 64 bits wide, called RAX, RBX, etc.
8 more general-purpose registers: R8 thru R15
boot: *still* go thru 16-bit and 32-bit modes on the way!

x86 Instruction Set

Intel syntax: `op dst, src` (Intel manuals!)
AT&T (gcc/gas) syntax: `op src, dst` (labs, xv6)
uses b, w, l suffix on instructions to specify size of operands
Operands are registers, constant, memory via register, memory via constant
Examples:

<u>AT&T syntax</u>	<u>"C"-ish equivalent</u>	
movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

Instruction classes
data movement: MOV, PUSH, POP, ...
arithmetic: TEST, SHL, ADD, AND, ...
i/o: IN, OUT, ...
control: JMP, JZ, JNZ, CALL, RET
string: REP MOVSB, ...
system: IRET, INT

PC Architecture and Processor Setup

Outline

x86 instruction set
GCC calling conventions

x86 Instruction Set

Intel syntax: `op dst, src` (Intel manuals!)
AT&T (gcc/gas) syntax: `op src, dst` (labs, xv6)
uses b, w, l suffix on instructions to specify size of operands
Operands are registers, constant, memory via register, memory via constant
Examples:

AT&T syntax	"C"-ish equivalent	
movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

Instruction classes
data movement: MOV, PUSH, POP, ...
arithmetic: TEST, SHL, ADD, AND, ...
i/o: IN, OUT, ...
control: JMP, JZ, JNZ, CALL, RET
string: REP MOVSB, ...
system: IRET, INT
Intel architecture manual Volume 2 is *the* reference

gcc x86 calling conventions

x86 dictates that stack grows down:

Example instruction	What it does
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax addl $4, %esp
call 0x12345	pushl %eip (*) movl $0x12345, %eip (*)
ret	popl %eip (*)

(*) *Not real instructions*
Use example of a function foo() calling a function bar() with two arguments. Assume that bar returns the sum of the two arguments; write bar's code in C and assembly, and explain the conventions.
Formally, introduce the conventions. GCC dictates how the stack is used. Contract between caller and callee on x86:
at entry to a function (i.e. just after call):
%eip points at first instruction of function
%esp+4 points at first argument
%esp points at return address
after ret instruction:
%eip contains return address

%esp points at arguments pushed by caller
called function may have trashed arguments
%eax (and %edx, if return type is 64-bit) contains return value (or trash if function is `void`)
%eax, %edx (above), and %ecx may be trashed
%ebp, %ebx, %esi, %edi must contain contents from time of `call`
Terminology:
%eax, %ecx, %edx are "caller save" registers
%ebp, %ebx, %esi, %edi are "callee save" registers
Discuss the frame pointer, and why it is needed --- so that the name of an argument, or a local variable does not change throughout the function body. Discuss the implications of using a frame pointer on the prologue and epilogue of the function body.
Functions can do anything that doesn't violate contract. By convention, GCC does more:
each function has a stack frame marked by %ebp, %esp

```
               +------------+   |
               | arg 2      |   \
               +------------+    >- previous function's stack frame
               | arg 1      |   /
               +------------+   |
               | ret %eip   |   /
               +============+
               | saved %ebp |   \
        %ebp-> +------------+   |
               |            |   |
               |   local    |   \
               | variables, |    >- current function's stack frame
               |    etc.    |   /
               |            |   |
               |            |   |
        %esp-> +------------+   /
```

%esp can move to make stack frame bigger, smaller
%ebp points at saved %ebp from previous function, chain to walk stack
function prologue:

```
pushl %ebp
movl %esp, %ebp
```

function epilogue can easily find return EIP on stack:

```
movl %ebp, %esp
popl %ebp
```

The frame pointer (in `ebp`) is not strictly needed because a compiler can compute the address of its return address and function arguments based on its knowledge of the current depth of the stack pointer (in `esp`).
The frame pointer is useful for debugging purposes, especially the current backtrace (function call chain) can be computed by following the frame pointers. The current function is based on the current value of `eip`. The current value of `*(ebp+4)` provides the return address of the caller. The current value of `*((*ebp) + 4)` (where `*ebp` contains the saved `ebp` of the caller) provides the return address of the caller's caller, the current value of `*(*(*ebp) + 4)` provides the return address of the caller's caller's caller, and so on . . .
Discuss the code for the "backtrace" function in gdb.
Compiling, linking, loading:
Preprocessor takes C source code (ASCII text), expands #include etc, produces C source code
Compiler takes C source code (ASCII text), produces assembly language (also ASCII text)
Assembler takes assembly language (ASCII text), produces `.o` file (binary, machine-readable!)
Linker takes multiple '`.o`'s, produces a single *program image* (binary)

Loader loads the program image into memory at run-time and starts it executing

Physical Memory, I/O, Segmentation

Outline

x86 Physical Memory Map
I/O
Example hardware for address spaces: x86 segments

x86 Physical Memory Map

The physical address space mostly looks like ordinary RAM
Except some low-memory addresses actually refer to other things
Writes to VGA memory appear on the screen
Reset or power-on jumps to ROM at 0x000ffff0 (so must be ROM at top of BIOS)

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

I/O

Original PC architecture: use dedicated *I/O space*
Works same as memory accesses but set I/O signal
Only 1024 I/O addresses
Accessed with special instructions (IN, OUT)
Example: write a byte to line printer:

```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
```

```
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

Memory-Mapped I/O
Use normal physical memory addresses
Gets around limited size of I/O address space
No need for special instructions
System controller routes to appropriate device
Works like ``magic" memory:
Addressed and *accessed* like memory, but ...
... does not *behave* like memory!
Reads and writes can have ``side effects"
Read results can change due to external events

Example hardware for address spaces: x86 segments

The operating system can switch the x86 to protected mode, which supports virtual and physical addresses, and allows the O/S to set up address spaces so that user processes can't change them. Translation in protected mode is as follows:
selector:offset (virtual / logical addr)
==SEGMENTATION==>
linear address
==PAGING ==>
physical address
Next lecture covers paging; now we focus on segmentation.
Protected-mode segmentation works as follows (see handout):
segment register holds segment selector
selector: 13 bits of index, local vs global flag, 2-bit RPL
selector indexes into global descriptor table (GDT)
segment descriptor holds 32-bit base, limit, type, protection
la = va + base ; assert(va < limit);
choice of seg register usually implicit in instruction
ESP uses SS, EIP uses CS, others (mostly) use DS
some instructions can take far addresses:
`ljmp $selector, $offset`
GDT lives in memory, CPU's GDTR register points to base of GDT
LGDT instruction loads GDTR
you turn on protected mode by setting PE bit in CR0 register
What about protection?
instructions can only r/w/x memory reachable through seg regs
not before base, not after limit
can my program change a segment register? yes, but... to one of the permitted (accessible) descriptors in the GDT
can my program re-load GDTR? no!
how does h/w know if user or kernel?
Current privilege level (CPL) is in the low 2 bits of CS
CPL=0 is privileged O/S, CPL=3 is user
why can't app modify the descriptors in the GDT? it's in memory...
what about system calls? how do they transfer to kernel?
app cannot **just** lower the CPL

Segmentation and Traps

Outline

Segmentation : Address spaces and Protection
Trap Handling

Segmentation

Protected-mode segmentation works as follows (see handout):
segment register holds segment selector
selector: 13 bits of index, local vs global flag, 2-bit RPL
selector indexes into global descriptor table (GDT)
segment descriptor holds 32-bit base, limit, type, protection
la = va + base ; assert(va < limit);
choice of seg register usually implicit in instruction
ESP uses SS, EIP uses CS, others (mostly) use DS
some instructions can take far addresses:
`ljmp $selector, $offset`
GDT lives in memory, CPU's GDTR register points to base of GDT
LGDT instruction loads GDTR
you turn on protected mode by setting PE bit in CR0 register
What about protection?
instructions can only r/w/x memory reachable through seg regs
not before base, not after limit
can my program change a segment register? yes, but... to one of the permitted (accessible) descriptors in the GDT
can my program re-load GDTR? no!
how does h/w know if user or kernel?
Current privilege level (CPL) is in the low 2 bits of CS
CPL=0 is privileged O/S, CPL=3 is user
why can't app modify the descriptors in the GDT? it's in memory...
what about system calls? how do they transfer to kernel?
app cannot **just** lower the CPL

Traps

The x86 processor uses a table known as the *interrupt descriptor table* (IDT) to determine how to transfer control when a trap occurs. The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 256. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's IDT, which the kernel sets up in kernel-private memory of the kernel's choosing, much like the GDT. From the appropriate entry in this table the processor loads:
the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. In PIOS, all exceptions are handled in kernel mode, privilege level 0.

Entering and Returning from Trap Handlers

When an x86 processor takes a trap while in kernel mode, it first pushes a *trap frame* onto the kernel stack, to save the old values of certain registers before the trap handling mechanism modifies them. The processor then looks up the CS and EIP of the trap handler in the IDT, and transfers control to that instruction address. The following diagram illustrates the format of the basic kernel trap frame, defining the state of the kernel stack on entry to the trap handler:

```
                     +--------------------+ <---- old ESP
                     |     old EFLAGS     |     " - 4
                     | 0x00000 | old CS   |     " - 8
                     |      old EIP       |     " - 12
                     +--------------------+ <---- ESP
```

For certain types of x86 exceptions, in addition to the basic three 32-bit words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the x86 manuals to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the trap handler:

```
                     +--------------------+ <---- old ESP
                     |     old EFLAGS     |     " - 4
                     | 0x00000 | old CS   |     " - 8
                     |      old EIP       |     " - 12
                     |     error code     |     " - 16
                     +--------------------+ <---- ESP
```

The x86 processor provides a special instruction, `iret`, to return from trap handlers. It expects the kernel's stack to look like the *first* figure above, with ESP pointing to the old EIP. When the processor executes an `iret` instruction, pops the saved values of EIP, CS, and EFLAGS off the stack and back into the corresponding registers, and resumes instruction execution at the popped EIP.

Note that when returning from a trap, the processor doesn't actually know or care whether the "old" values it is popping off the stack are really the exact same values that it originally pushed onto the stack on entry to the trap handler. Think about what would happen - for better or worse - if the kernel trap handler changes these values during its execution.

The Task State Segment. The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of `EIP` and `CS` before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel: for example, one user mode thread could change the kernel state of another thread while the latter is in a system call, or user code could simply point ESP to unmapped or read-only memory, making it impossible for the processor to push the trap frame and causing an immediate reset as described above.

For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.

Although the TSS is large and can potentially serve a variety of purposes, PIOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in PIOS/Pintos/xv6 is privilege level 0 on the x86, the processor uses the `ESP0` and `SS0` fields of the TSS to define the kernel stack when entering kernel mode. PIOS/Pintos/xv6 don't use any other TSS fields.

Combined, the IDT and TSS provide the kernel with a mechanism to ensure that traps are handled only by calling well-defined entrypoints in the kernel (the interrupt vectors in the IDT) and that trap handlers will have a well-defined, protected workspace (the stack pointers in the TSS). Exactly *where* these entrypoints and kernel stacks are located is up to the kernel, however.

Traps, Handlers

Outline

Traps : entry and return
Handler examples

Traps

The x86 processor uses a table known as the *interrupt descriptor table* (IDT) to determine how to transfer control when a trap occurs. The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 256. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's IDT, which the kernel sets up in kernel-private memory of the kernel's choosing, much like the GDT. From the appropriate entry in this table the processor loads:
the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. In most kernels, all exceptions are handled in kernel mode, privilege level 0.

Entering and Returning from Trap Handlers

When an x86 processor takes a trap while in kernel mode, it first pushes a *trap frame* onto the kernel stack, to save the old values of certain registers before the trap handling mechanism modifies them. The processor then looks up the CS and EIP of the trap handler in the IDT, and transfers control to that instruction address. The following diagram illustrates the format of the basic kernel trap frame, defining the state of the kernel stack on entry to the trap handler:

```
+--------------------+ <---- old ESP
|     old EFLAGS     |     " - 4
| 0x00000 | old CS   |     " - 8
|      old EIP       |     " - 12
+--------------------+ <---- ESP
```

For certain types of x86 exceptions, in addition to the basic three 32-bit words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the x86 manuals to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the trap handler:

```
+--------------------+ <---- old ESP
|     old EFLAGS     |     " - 4
| 0x00000 | old CS   |     " - 8
|      old EIP       |     " - 12
|     error code     |     " - 16
+--------------------+ <---- ESP
```

The x86 processor provides a special instruction, `iret`, to return from trap handlers. It expects the kernel's stack to look like the *first* figure above, with ESP pointing to the old EIP. When the processor executes an `iret` instruction, pops the saved values of EIP, CS, and EFLAGS off the stack and back into the corresponding registers, and resumes instruction execution at the popped EIP.
Note that when returning from a trap, the processor doesn't actually know or care whether the "old" values it is popping off the stack are really the exact same values that it originally pushed onto the stack on entry to the trap handler. Think about what would happen - for better or worse - if the kernel trap handler changes these values during its execution.

The Task State Segment. The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of `EIP` and `CS` before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel: for example, one user mode thread could change the kernel state of another thread while the latter is in a system call, or user code could simply point ESP to unmapped or read-only memory, making it impossible for the processor to push the trap frame and causing an immediate reset as described above.
For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.
Although the TSS is large and can potentially serve a variety of purposes, we will use it only to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in most kernels is privilege level 0 on the x86, the processor uses the `ESP0` and `SS0` fields of the TSS to define the kernel stack when entering kernel mode. We don't use any other TSS fields.
Combined, the IDT and TSS provide the kernel with a mechanism to ensure that traps are handled only by calling well-defined entrypoints in the kernel (the interrupt vectors in the IDT) and that trap handlers will have a well-defined, protected workspace (the stack pointers in the TSS). Exactly *where* these entrypoints and kernel stacks are located is up to the kernel, however.

Kernel Stack Management

A particularly important kernel design issue is *how many* kernel stacks there are, and which OS abstraction they are associated with. There are basically two common models:
The xv6 kernel, like most Unix kernels, uses a *process model*: it associates a kernel stack with each user process, so that whenever the kernel is running on behalf of a particular process, it runs on that process's kernel stack, and it switches stacks whenever it switches between processes.
Some kernels, in contrast, is an *interrupt model* kernel, which means it maintains one kernel stack per *physical CPU*, irrespective of the number of processes. Kernel code running on a given CPU always runs on that CPU's permanently-assigned kernel stack regardless of what user process is running, and kernel code thus never switches kernel stacks once it has booted. The downside of this simple design is that the kernel code cannot use its stack to maintain any state on behalf of processes that are not currently running: if the kernel is working on behalf of a process and needs to put the process to sleep waiting on some event, the kernel must explicitly store all relevant information about what it was doing somewhere else, such as in the process control structure, or it would be lost when the kernel switches to another process.

An Example

Let's put the above pieces together and trace through an example. Suppose the processor is executing code in user mode and encounters a divide instruction that attempts to divide by zero.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the TSS.
2. The processor pushes the following basic trap frame onto the kernel stack, starting at `kstackhi`:

```
+--------------------+ &cpu->kstackhi
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+  <---- ESP
```

3. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets `CS:EIP` to point to the handler function defined there.
4. The handler function takes control and handles the exception, for example by terminating the user environment.

As you can see, the trap frame the processor pushes on kernel entry from user is similar to the one it pushes when it is *already* in the kernel, except in this case the processor also pushes the SS and ESP registers *before* pushing the frame discussed in the preivous part of the lab. You can see this difference reflected in the comments in the definition of `trapframe` in `inc/trap.h`, and on the definitions of the `trapframe_usize()` and `trapframe_ksize()` macros in the same header file.
For those traps for which the processor also pushes an error code when taking a trap from kernel mode, as discussed in the last part, it pushes an error code in the same fashion for traps from user mode. For these traps, therefore, a trap from user mode will leave the kernel stack in the following state:

```
+--------------------+ &cpu->kstackhi
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20
|     error code     |     " - 24
+--------------------+  <---- ESP
```

Entering User Mode

What piece of register state in the processor actually defines which privilege level it is executing in at a given moment? Many architectures use a "kernel mode" flag in a control register of some kind, but x86 processors uses the low two bits of the CS register, effectively treating privilege level as a property of the currently running code segment.
We described above how the processor *leaves* user mode and enters the kernel via a trap, but how does the kernel *enter* user in the first place? Simple: the kernel "returns" to user mode - even if the processor has never been there before!
When the processor executes an `iret` instruction, it pops its standard trap frame off the stack starting with the old EIP, but it doesn't actually know or care whether *it* actually pushed that frame on the stack or if it got there some other way. Thus, the kernel can always *manufacture* a trap frame representing whatever user mode state it wants to load into the processor, and "return" from it via `iret` to enter user mode. (There are other ways to switch to user mode on the x86, but this is the most general method.)

Software Interrupts

Now that your kernel has basic exception handling capabilities and can enter user mode, you will refine it to handle traps that user mode code may cause deliberately for various purposes; we refer to such traps as *software interrupts*. We do not want user code to be able to invoke *just any* interrupt vector in the IDT deliberately via `int` instructions, however: that might allow user code to confuse the kernel into thinking that some special event has occurred when it has not. For this reason, all IDT descriptors have a *descriptor privilege level* indicating what privilege level is required for software to invoke that interrupt vector deliberately via a software interrupt instruction. Most of these vectors are normally set to "privileged" (ring 0), but we want the vectors used for software interrupts to be set so user mode code can invoke them.

Review : Segmentation, Interrupt Descriptor Table, Kernel stacks

If there is only one CPU, only one process could be running on the CPU at any time. The kernel time-slices the CPU execution time, so that multiple processes appear to run concurrently. As we saw last time, while a process is running the kernel is *not* controlling the CPU at that time. The kernel regains control of the CPU if one of the following occurs, for example:
The process makes a system call
The process causes an exception
An external interrupt occurs
The Interrupt Descriptor table specifies what code to execute, and at what privilege level, on any of these events, based on the corresponding interrupt vector number.
If there are multiple CPUs, each CPU independently executes in this fashion. For example, one CPU could be executing a process, while another CPU could be executing in the kernel mode. Notice that by definition, one process (or one thread of a process, if a process could have multiple threads) may only be executing on at most one CPU at any time.
On a switch from process execution to kernel execution (from unprivileged to privileged mode), the CPU also switches the stacks. The stack for the kernel mode is obtained using the Task State Segment (TSS). If the kernel uses a *process model*, each process will have a separate kernel stack. Hence, the kernel must load the kernel stack of the process into TSS, before transferring execution control to that process.
If the kernel uses an *interrupt model*, each CPU has only one stack, which remains constant throughout the execution of the system. In this case, every switch from the process to the kernel starts on a *clean* stack, i.e., on an entry it will only contain the values pushed by this switch (e.g., saved SS, saved ESP, saved EFLAGS, saved CS, saved EIP). A kernel may want to interrupt its own execution, while it was running on behalf of a process. For example, this can happen if the process made a system call to read contents from a disk file. Reading the file contents may take a long time, and so the kernel may want to *context switch* to another process while the disk is responding. In this case, the kernel must save the contents of the current stack (somewhere in its data structures where it maintains per-process information), before loading the new process. In future, if the kernel wants to switch back to the old process (e.g., if the disk device has supplied the requested contents of the file), the kernel must initialize the stack using the saved contents, before transferring control to the old process.
In the process model, the kernel is always executing in the context of the stacks of one of the processes. If the kernel wants to interrupt its execution, it simply saves the current stack pointer, and loads the stack pointer of the new process. On a return to the old process, the old stack pointer is restored.
The kernel needs to store information per process. For example, it needs to store the list of all processes, their state (e.g., are they ready to run, are they waiting for an event to complete, are they already running, etc.), values that define their address space (e.g., segment selectors), and values that define their kernel stacks (stack pointers in process model, or contents of the stack in the interrupt model).
This information is typically stored as a list of ``Process Control Blocks" (PCBs). Each PCB contains process state:
pid
state : ready, running, blocked, etc.
address space (e.g., segment selectors)
kernel stack
saved registers
other information
This list is allocated in the kernel address space (inaccessible to the other processes). The "handlers" consult these data structures to decide what action to take on any interrupt/exception.
For the most part, the kernel starts execution in one of the handlers. For example, the kernel implements a system call handler that internally calls the functions that implement the functionality of that call. One of the system calls is `yield()`, which tells the kernel that the process is interested in relinquishing the CPU to another process (if any exists). This may happen if a process is waiting for an event, and knows that it is going to take a long time. In this case, instead of occupying the CPU, a process may make the "good gesture" of calling `yield()`.
If a process does not call `yield()`, the kernel has other mechanisms to take control away from the process. This is called preemption of a process. Preemption is necessary to ensure that the process cannot keep executing forever, and the OS can acquire control back. A typical way to do this is to use a hardware timer device. The OS configures the timer device to generate an interrupt periodically. It then sets up the interrupt handler for the timer interrupt,

before tranferring control to the process. On a timer interrupt the kernel's handler is invoked, and it is free to let the process continue execution (by simply returning from the interrupt), or to *context-switch* to another process (which involves saving the current state of the execution of the current process and loading the state of execution of the new process, before returning from the interrupt).

Typically, the kernel will have 4-5 descriptor entries in the GDT. One of those entries will define the kernel address space (using `kbase` and `klimit`), and another one will define the user (or process) address space (using `ubase` and `ulimit`). On a context switch, the kernel will overwrite the descriptor entry containing `ubase` and `ulimit` with the base and limit of the new process, before transferring control to it.

FAQ

GDT can have up to 2^13 entries. Why can't we use one entry per process? Why do we need to overwrite GDT entries on every context switch?
Because when one process (say P1) is running, the address space of another process (say P2) should not be accessible through the GDT. So it makes sense to overwrite P2's GDT descriptor (its base and limit) with P1's GDT descriptor before context switching to P1.
Why does the GDT have 2^13 entries? Isn't this very wasteful, if we are going to use only 4-5 entries?
Actually the hardware allows you to specify the size of the GDT (which can be up to 2^13 entries). The GDTR register actually points to a memory area of six bytes that specifies the GDT base and the GDT size. (see the semantics of the LGDT instruction in Intel Reference Manual volume 2 (page 427) for full details.
Can a process customize interrupt/exception handlers?
The UNIX abstractions do not allow this. But one could potentially design an OS where this is allowed. Customization in UNIX is done through *signal handlers*, but the mapping of exceptions to signals is fixed inside UNIX. Also signal handlers execute in unprivileged mode.
Why have two abstractions: signal handlers (for process) and trap handlers (for kernel)?
Traps are a hardware abstraction. Signals are an operating system abstraction. It is possible for an OS to not have signals (this is an OS design decision), and yet support traps from the hardware. Similarly, not all signals are caused due to hardware traps (e.g., signals can be generated using the kill system call).
If multiple devices (e.g., USB mouse and USB keyboard) are connected to the same hardware port (with the same interrupt vector), and one of them raises and interrupt, how does the CPU find out whom to serve?
Interrupts are only a way of requesting immediate attention by *some* device registered on that vector. If multiple devices are registered, the interrupt handler will typically ask each of them if any of them needs its attention (using I/O ports or memory mapped I/O).

Memory Management

Let's now continue our discussion on memory management.
Discuss malloc(), free() implementations in kernel, and at process-level.
Discuss palloc(), free() implementation in kernel.

Fragmentation

Fragmentation within kernel data structures : use kmalloc and kfree
Fragmentation within process data structures : use process's malloc and free
Fragmentation of process address space :
Last time : use multiple segments (complicates programming model)
Next time : Paging

Segmentation Review

Typically, the kernel uses only a few GDT entries: some for the hardware's data structures (e.g., Task State Segment), and others for the OS to use. For example, the kernel may use only two segment descriptors, one for the user's address space (USEGMENT), and another for the kernel's address space (KSEGMENT). The contents of the KSEGMENT descriptor remain constant throughout the execution of the system, as the kernel does not change its own location. The contents of the USEGMENT descriptor are overwritten on every context switch, and loaded with the base and limit values of the current process.

Caching of Segment Descriptors

A segment descriptor is cached on the CPU (on chip) each time a segment register is loaded. Thus the logic to translate a virtual address (VA) through segmentation does not require a memory access.

Address translation and sharing using page tables

Why do we care about x86 address translation?
It can simplify s/w structure: addresses in one process not constrained by what other processes might be running.
It can implement tricks like demand paging and copy-on-write.
It can isolate programs to contain bugs or increase security.
It can provide efficient sharing between processes.
Why aren't protected-mode segments enough?
Why did the 386 add translation using page tables as well?
Isn't it enough to give each process its own segments?
Programming model, fragmentation
In practice, segments are little-used
Translation using page tables (on x86):
segmentation hardware first computes the linear address
in practice, most segments (e.g. in `pintos`, Linux) have base 0 and max limit, making the segmentation step a no-op.
paging hardware then maps linear address (la) to physical address (pa)
(we will often interchange "linear" and "virtual")
when paging is enabled, every instruction that accesses memory is subject to translation by paging
paging idea: break up memory into 4096-byte chunks called pages
independently control mapping for each page of linear address space
compare with segmentation (single base + limit): many more degrees of freedom
4096-byte pages means there are 2^20 = 1,048,576 pages in 2^32 bytes
conceptual model: array of 2^20 entries, called a page table, specifying the mapping for each linear page number
table[20-bit linear page #] => 20-bit phys page #
PTE entries: bottom of handout
20-bit phys page number, present, read/write, user/supervisor, etc
puzzle: can supervisor read/write user pages?
can use paging hardware for many purposes
(seen some of this two lectures ago)
flat memory
segment-like protection: contiguous mappings
solve fragmentation problems when allocating more memory (xv6-like process memory layout)
demand-paging (%cr2 stores faulting address)
copy-on-write
sharing, direct access to devices (e.g. /dev/fb on linux)
switching between processes
where is this table stored? back in memory.
in our conceptual model, CPU holds the physical address of the base of this table.
%cr3 serves this purpose on the x86 (with one more detail below)
for each memory access, access memory again to look up in table
why not just have a big array with each page #'s translation?

same problems that we were trying to solve with paging! (demand-paging, fragmentation)
so, apply the same trick
we broke up our 2^32-byte memory into 4096-byte chunks and represented them in a 2^22-byte (2^20-entry) table
now break up the 2^22-byte table into 4096-byte chunks too, and represent them in another 2^12-byte (2^10-entry) table
just another level of indirection
now all data structures are page-sized
386 uses 2-level mapping structure
one page directory page, with 1024 page directory entries (PDEs)
up to 1024 page table pages, each with 1024 page table entries (PTEs)
so la has 10 bits of directory index, 10 bits table index, 12 bits offset
%cr3 register holds physical address of current page directory
puzzle: what do PDE read/write and user/supervisor flags mean?

Address translation and sharing using page tables

386 uses 2-level mapping structure
one page directory page, with 1024 page directory entries (PDEs)
up to 1024 page table pages, each with 1024 page table entries (PTEs)
so la has 10 bits of directory index, 10 bits table index, 12 bits offset
%cr3 register holds physical address of current page directory
puzzle: what do PDE read/write and user/supervisor flags mean?
now, access memory twice more for every memory access: really expensive!
optimization: CPU's TLB caches vpn => ppn mappings
if you change any part of the page table, you must flush the TLB!
by re-loading %cr3 (flushes everything)
by executing `invlpg` *va*
Is TLB write through? Is it write back? If not, what is it?
turn on paging by setting CR0_PG bit of %cr0
Here's how the MMU translates an la to a pa:

```
uint
translate (uint la, bool user, bool write)
{
  uint pde;
  pde = read_mem (%CR3 + 4*(la >> 22));
  access (pde, user, write);
  pte = read_mem ( (pde & 0xfffff000) + 4*((la >> 12) & 0x3ff));
  access (pte, user, write);
  return (pte & 0xfffff000) + (la & 0xfff);
}

// check protection. pxe is a pte or pde.
// user is true if CPL==3
void
access (uint pxe, bool user, bool write)
{
  if (!(pxe & PG_P)
     => page fault -- page not present
  if (!(pxe & PG_U) && user)
     => page fault -- not access for user

  if (write && !(pxe & PG_W)) {
    if (user)
       => page fault -- not writable
    if (%CR0 & CR0_WP)
       => page fault -- not writable
  }
}
```

Can we use paging to limit what memory an app can read/write?
user can't modify cr3 (requires privilege)
is that enough?
could user modify page tables? after all, they are in memory.
Who stores what?
cr3: physical address
GDTR: linear address
GDT descriptor: linear address
IDTR: linear address
IDT descriptor: virtual address
Page tables vs. Segmentation
Good: Pages are easy to allocate (keep a list of available pages and just allocate the first available).
Good: Pages are easy to swap as everything is same size and pages are usually same size as disk blocks.

Bad: Page tables can become very large (need one entry for each page-sized unit of virtual memory).

Process Address Spaces Using Paging

Paging allows non-contiguous regions to be mapped in Virtual Address (VA) space and Physical Address (PA) space.
To map a region a VA space, the page directory and page table entries at the corresponding offsets should be mapped.
To map a region of PA space, the values in the page table entries should point to the corresponding physical pages.
When paging is enabled, all addresses (including EIP, ESP, direct-addressing, etc.) go through the paging hardware.
Each entry in the Page Directory (PDE - page directory entry) and in the Page Table (PTE - page table entry) is 32-bit wide. The top 20 bits store a pointer (using physical address) to the page table or the page itself. The last 12 bits contains flags: `present`, `writable`, and `user-accessible`.
The final access permissions are determined by looking at the stricter of the permissions specified by the flags in the corresponding PDE and PTE.
CR3, PDE, and PTE contain physical addresses
Each process has a separate page table, so a different address is loaded into the CR3 register on every context switch.
Also, each page table also maps the kernel into its address space above a certain address. This address is 0x80000000 (2GB) on xv6.
Thus the kernel is mapped at the same addresses in every page table. So, copies of the same PDEs/PTEs are present in the page directory and page tables of all process to map the kernel at these addresses. Let's call this address space region, the kernel's address space. Another way to put this is that each process has two halves: kernel half and user half. The kernel half is shared among all processes. The user half is separate. Thus every user process (separate address spaces) also acts as a kernel thread (shared address space).
On xv6, the kernel maps the entire physical memory into its address space starting at 0x80000000. Thus, virtual address `(0x80000000+x)` always translates to physical address `x`.
The kernel's address space holds the kernel's code and the kernel's data. All the other space is managed as a heap (using `malloc` and `free` for example).
The kernel's heap is used to allocate memory for its own data structures (e.g., PCBs), processes' kernel stacks, processes' address spaces, among other things.
For example, if a kernel wants to allocate an address space for a process, it simply mallocs some space from its heap, converts the returned pointer to its physical address, and then creates a mapping into the process's user-side address space (by creating entries in its page table) so the process can access it in future.
Thus every page that is accessible by the user also has a mapping in the kernel's address space. Thus two entries in the page table point to the same physical page (one in the *kernel-space*, and another in the *user-space*).
The Interrupt Descriptor Table (IDT) is setup to hold kernel pointers, i.e., the CS:EIP entries are setup such that the handler runs in priviliged mode (last two bits of CS are zero), and EIP points to a kernel address (above `0x80000000`).
On a trap (due to an interrupt, exception, or system call), execution control transfers to the kernel's handler running in privileged mode. Because the kernel is mapped in every process's address space, the pointers to the handler (and the kernel stack of the process) in the kernel address space are always valid. Thus, the handler can start executing immediately on a trap, without requiring any address-space switch.
The kernel's handler executes the necessary logic. This sharing of address space between the process user-space and the kernel also allows very fast communication between the kernel and the user spaces. For example, if the user wants to provide a string argument to a system call, it can simply pass a pointer to the string stored in its own address space. The kernel can de-reference that pointer, and because the user's address space is still mapped, the de-referencing of the pointer will result in the desired data (as supplied by the user). Thus communication from user-space to kernel-space can be done only by sending a pointer - this is very fast. Recall that kernel can always access user pages (the user bit in PDE/PTE only prevents the user from accessing a kernel page).
This fast communication between the user and the kernel is the primary reason, why the kernel maps itself entirely in the process page table. Linux maps itself starting at `0xc0000000` (3GB) and Windows maps itself starting at `0x80000000` (2GB), but can be configured to map itself starting at `0xc0000000`. As already discussed, xv6 maps itself starting at `0x80000000`.
Because xv6 maps the entire available physical memory in the kernel space, there is a limit to the size of physical memory that it can support (less than 2GB). Full operating systems (like Linux/Windows) handle this by mapping

a part of the physical memory at all times (esp. the parts which contain kernel's code and data). For the other parts of the physical memory, the kernel maps and unmaps those regions into a VA space, to access them, depending on what needs to get accessed.

The page directory itself is allocated from the kernel address space (per process), and its corresponding physical address is stored in the `cr3` register when that process is running. Security is ensured by disallowing a process from accessing its own page table (by mapping the pages containing the page tables and page directory only in the kernel address space and not in the user address space).

Compare this organization to a kernel which only uses segmentation. In that case, a trap would switch the `CS` register (and the associated base and limit values of the descriptor) and thus the kernel would be executing in a different address space. All other segment registers will also be loaded with the kernel's base, so they can access the kernel's data. However, if the kernel wants to read an argument from the user-space (passed as a pointer), it needs to switch one of its segment registers to the user's descriptor before de-referencing that pointer through that segment register.

Notice that the kernel cannot simply de-reference a pointer supplied by the user in this case, as the address space is now different - the kernel needs to switch to user address space to de-reference that pointer.

Compare this organization to another paging-based organization of the kernel, where the entire kernel is not mapped into the process address space, but only a small slice of the kernel address space (which stores the trap handler and the kernel stack) is mapped in the process address space. In this case, the trap handler will switch to the kernel's page table and then execute the kernel's logic in a different address space (by switching to a different page table). However, if the kernel needs to de-reference a user pointer, it cannot do so. In this case, the trap handler should also de-reference all user pointers and copy them into its space before switching to the kernel's address space.

Because transitions between user and kernel require page table switches, and because the kernel cannot access user memory directly in this organization (the trap handler needs to copy portions of user memory for the kernel to read), this organization is relatively more complex and less performant.

Bootup : Executing the first instruction after power-on

Switching on your computer, makes it start from a clean state, where memory contents are completely uninitialized. Only the disk contains state that persists across power cycles. The x86 architecture specifies that when a computer is powered-on, the first block (or sector) on disk will be read and its contents pasted at address `0x7c00`, and control will be transferred to the instruction at its first byte (address `0x7c00`). Recall that the x86 architecture boots in 16-bit mode, with no paging. Also, the segmentation hardware in 16-bit mode simply multiplies the segment register's value by 16 and adds it to the virtual address to obtain a physical address.

A disk block (or sector) is sized at 512 bytes. Thus the machine reads the 512 bytes in the first block and pastes them at addresses `0x7c00-0x7e00` and transfers control to `0x7c00`. This code, which must fit in 512 bytes then loads the kernel from the disk (it must know the location of the kernel on disk) and pastes it into memory (it must know the location in memory where it needs to be pasted), before transferring control to it.

TLBs, Large Pages, Boot sector

TLB

optimization: CPU's TLB caches vpn => ppn mappings
Typical size: 12-4096 entries
typical hit time: 0.5-1 clock cycle
typical miss penalty: memory access (10-100 clock cycles)
typical miss rate: 0.01-1%
Typically fully associative (to minimize miss rate)
Cache replacement policy: e.g., FIFO, LRU, etc.
It is important for TLB miss rates to be really low (e.g., 0.1-1%) for paging to be a useful idea. Fortunately, this is true in practice because programs typically exhibit large temporal and spatial locality.
if you change any part of the page table, you must flush the TLB!
by re-loading %cr3 (flushes everything). e.g., on context switch
by executing `invlpg` *`va`*
So, neither write back, nor write through. Requires explicit invalidations and flushes. Thus OS needs to be careful, e.g., if it overwrites the page table but does not execute an appropriate `invlpg` instruction, a user process may be able to access another process's page.

Large Pages

x86 also supports "large pages" of size 4MB. This allows large processes to be mapped with less page table space overhead, less translation time, and most importantly, less TLB pressure. For example, if a 4MB page can be placed contiguously in physical memory, it requires only 1 TLB entry to be mapped. If we had used regular 4KB pages, 1024 entries would have been required.
With large pages, the top 10 bits of the virtual address (VA) are used to index into the page directory. The top 10 bits of the PDE are used to identify the Physical Page Number (PPN). The last 22 bits of the virtual address are used as an offset into the physical page. The last 22 bits of the PDE are unused (or used for storing flags), when using large pages.
Depending on the size of the process and its access patterns in memory, the kernel may decide to use large or small pages for it (to optimize performance).
One immediate use of large pages is to map the kernel's pages. Because the kernel typically maps the entire physical memory as one contiguous chunk in the VA space, large pages can significantly reduce the number of page table entries required to implement this mapping (thus saving physical memory space). More importantly, mapping the kernel in this way reduces TLB pressure and also improves translation time in case of TLB miss.

Boot sector

Only disk state persists across power cycles (off and on). Thus the kernel code/data live on disk, so they can be loaded on bootup.
The disk is divided into *sectors* (also called *blocks*) of size 512 bytes each.
Logically, the disk can be thought of as a linear array of sectors, starting with sector number `0`.
The first sector (number `0`) is special, and is called the bootsector.
The hardware (or BIOS) promises to load the bootsector (of 512 bytes) at physical addresses `0x7c00-0x7e00`, and transfers control to address `0x7c00`.
The OS developer needs to write the contents of the bootsector such that it contains the code instructions and the associated data to load the kernel from the disk into memory and transfer control to it. The code and data to do this, must fit within 512 bytes.
We will next look at the bootsector code of xv6 for a concrete example.

xv6 Bootsector

Look at Sheet 84, line 8412 (`bootasm.S`)
The first instruction at physical address 0x7c00 is the instruction at this line, namely `cli`.
`cli` is used to clear the interrupt flag (IF). The interrupt flag is a bit in the EFLAGS register which indicates if the processor accepts external interrupts or not. If `IF=1`, and an external interrupt occurs (by setting the `INTR` pin in the chip by an external device), the corresponding interrupt handler is invoked by transferring control to it through the IDT. On the other hand, if `IF=0`, any external interrupt is ignored.
Before this instruction, the BIOS may have installed some interrupt handlers. The kernel now wants to reinitialize everything, and will install its own interrupt handlers. However, at this point, it has not installed any handlers, so it must disable any external interrupts using the `cli` instruction.
Look at lines 8415-8418. These instructions load zero into all the relevant segments, i.e. `ds, es, ss`. Recall that the processor is executing in 16-bit mode at this time, and so segmentation works by simply multiplying the segment register by 16 and adding it to the offset to obtain a physical address. By zeroing out the segment registers, the programmer intends to use a flat address space.

Ignore lines 8422-8436. This code enables the processor to access physical addresses above 2^{20}. Recall that the original 16-bit 8086 processor did not allow physical addresses above 2^{20}, and so this sequence of instructions ensures that this is allowed from now on.
Line 8441: `lgdt` loads the global descriptor table. At this point, the programmer is preparing to switch to 32-bit mode with segmentation enabled (also called protected mode). The operand of the `lgdt` instruction is a GDT descriptor `gdtdesc`, [see line 8487] which contains the size of the GDT (first two bytes) and the base address of the GDT (next four bytes).
The base address of the GDT (as given in `gdtdesc`) is specified by the `gdt` variable, which is defined at lines 8482-8485. These lines specify the first three entries of GDT (the size of the GDT is only 3 entries). The contents of these three entries are constants that refer to the corresponding : the first entry specifies a NULL segment, the second entry specifies a segment that is readable and executable and starts at 0 (base) and ends at 0xffffffff (limit), and the third entry specifies a segment that is writable and also starts at 0 (base) and ends at 0xffffffff (limit). Thus, the programmer initializes a flat address space (no segmentation), given that it initializes its base to 0 and limit to 2^{32}-1. (Notice that the macros SEG_NULLASM and SEG_ASM are macros defined elsewhere).
Also, the programmer will load the first descriptor into CS, and the third descriptor into the other segments (DS, ES, SS).
Ignore the next three instructions (8442-8444). They set the protected-mode bit in the control register `cr0`.
Finally, the programmer executes `ljmp` instruction to load the CS and EIP registers. Notice that the CS register is loaded with `SEG_KCODE * 8` - this effectively points it to descriptor number 1 (SEG_KCODE=1), in privileged mode (the last two bits are zero). Also, it loads the EIP with `start32` which is the address of the following instruction.
The assembly code uses the `.code32` directive to instruct the assembler to assemble all future code for 32-bit mode.
The first few instructions in 32-bit mode (8458-8461) setup DS, ES, and SS registers to point to the second segment descriptor in privileged mode (SEG_KDATA * 8).
Also, FS and GS are made to point to the NULL segment descriptor as the programmer is not planning to use these segments at all. (Recall that DS, ES, and SS are default segments for some instructions, so they definitely need to point to a valid segment).
Line 8467: Here, the stack is initialized to point to `start` (whose value is 0x7c00, as that's where the execution starts). Because the stack grows downwards, the space below 0x7c00 is used for the stack frames. The stack is initialized because in the next instruction, a function call will be made which will push the return address to stack.
Finally, the assembly makes a function call into the C function, `bootmain`. `bootmain` also points in the bootsector region (0x7c00-0x7e00), just like `start` and `start32`. `bootmain` is written in C and compiled into binary code before storing it in the bootsector.
Because the stack has been initialized, all local variables of `bootmain` will be allocated on the stack.

Loading the kernel, Initializing the page table

Loading the kernel

So far, we have looked at the first few instructions, written in assembly, to initialize 32-bit mode, segment registers, stack pointer, and a call to the C function `bootmain`.
`bootmain` is not supposed to return (it is supposed to load the kernel and jump into it). However, it may return if an error was observed during the loading process. In which case, the code following the `call` instruction executes some instructions which allow a user to debug the error.
The `bootmain` function (on Sheet 85) will read the kernel image (stored in ELF format) from the disk, load its program segments into physical memory, and branch to its first instruction.
The xv6 disk has the kernel image starting at sector no. 1 (second sector on disk). (Recall that sector no. 0 was the boot sector).
ELF is a standard format used to represent executable and object files. In this case, we use this format for the kernel. Typically, the OS loader loads an executable by parsing the ELF file. In this case, the "bootloader" will load the kernel by parsing the ELF formatted data stored in disk blocks (starting at disk block 1).
ELF Format: The first few bytes contain the ELF header, which must have a fixed "magic" number at a fixed offset (to confirm that it is indeed a valid ELF file), and should have a pointer to the "program headers" which store information about "program segments". Each program header represents a program segment (e.g., code, data, etc.). Also, the ELF header will store the number of program segments (or headers).
Each program header contains the following information:
`offset`: the file offset at which the program segment is in the file (or on the disk).
`filesz`: the size of the program segment in the file
`paddr`: the physical address at which the segment should be loaded.
`vaddr`: the virtual address at which the segment should be loaded.
`memsz`: the size of the program segment in memory. This must be greater than `filesz`. If `memsz` is not equal to `filesz`, the remaining `(memsz - filesz)` bytes in the segment are initialized to zero by the loader.
On Linux, type `man elf` to get full details about the ELF format.
Let's get back to the code in `bootmain.c`. Line 8527 makes a call to `readseg()` to read the first few bytes starting at disk sector 1 (notice the disk sector computed at line 8589 starts at sector 1) into a memory area named by the variable `elf`.
The `elf` variable is treated as an ELF header and its values are read (e.g., `magic` field, `phoff` field to obtain the location of program headers, `phnum` field to obtain the number of program headers).
The loop in lines 8536-8541 iterates over the program headers, and loads the program segment corresponding to each program header from disk to memory.
The call to `readseg()` in line 8538 loads `ph.filesz` bytes from disk at offset `ph.off` and stores them at *physical address* `ph.paddr`.
Recall that because we are currently working with physical memory (paging has not yet been enabled), the bootloader uses the physical address in the program header (`paddr`) to load the corresponding segment.
The lines 8539-8540 check if `ph.memsz` is greater than `ph.filesz`, and if so, zeroes out the remaining bytes (as required).
Finally, the ELF header also contains the starting instruction address in the `entry` field, to which the bootloader makes a function call at line 8546. At this point the control has shifted from the bootloader to the first instruction in the kernel.
To understand exactly what is happening, type the following command to look inside the kernel ELF file:

```
xv6$ objdump -p kernel
Program Header:
    LOAD off    0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
         filesz 0x0000b596 memsz 0x000126fc flags rwx
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

The first segment is supposed to be loaded from offset `0x1000`, at virtual address `0x80100000`, physical address `0x100000`. Its size in the file is `0xb596` bytes, and its size in memory is `126fc` bytes.
The second segment indicates a stack segment, and we can ignore it for now.
Thus the kernel has only one relevant segment that is loaded at physical address of 1MB (0x100000) and virtual address of KERNBASE + 1MB (0x80100000). KERNBASE is the starting address of the kernel's address space (0x80000000 or 2GB in xv6).
Why load at paddr 0x100000? that's 1MB. Why not 0x0? there are memory-mapped devices at physical address less than 1MB. Why not 2MB? that would work too but that may waste more physical memory space.
To see the starting instruction address of the kernel, type the following command:

```
xv6$ objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED

start address 0x0010000c
```

Notice that the start address is a physical address (and deliberately so). This is correct because at the time of branching to the start address, we are still operating in the physical address space.
All other symbols in the kernel are assigned virtual addresses (above KERNBASE).

Initializing the page table

The first few instructions in the kernel (`entry.S`) execute from physical address space and initialize the page table and enable paging. After paging is enabled, the kernel will start executing from virtual address space.
Ignore lines 1042-1044. They enable large pages in the hardware
Lines 1046-1047 load a page directory into cr3. The contents of the page directory are provided in a global array called `entrypgdir`. `entrypgdir` is assigned a virtual address (like all other kernel variables). Thus the code converts it into a physical address (by subtracting KERNBASE from it using V2P_WO macro) before loading it into `cr3` register (as `cr3` needs a physical address).
Let's look at `entrypgdir` defined at Sheet 13. `entrypgdir` is an array of 1024 `pde_t` entries. In this case, all other entries are set to zero (which means the corresponding pages are not present), except the first entry (entry #0) and the `(KERNBASE >> PDXSHIFT)`'th entry (entry #512) which both point to the same first 4MB page in the physical address space. Notice that these are the only two entries where the present bit (indicated by PTE_P) is set. Also, notice that both entries use large pages and allow writes.
As soon as paging is enabled (line 1051), the address space changes, and it is possible that the program counter `EIP` may now point to a different byte. To avoid this, the kernel developer has carefully used an identity mapping for the first 4MB, i.e., the first 4MB of VA space (virtual address space) map to the first 4MB of PA space (physical address space). Through this, the developer ensures that the current execution addresses will remain valid even after paging is enabled (all these addresses are less than 4MB by design).
Also, the size of the kernel (code and data) is less than 4MB, so this ensures that the kernel is fully mapped in the virtual address space at their corresponding physical addresses, even after paging is enabled.
Moreover, the first 4MB of the physical address space is also mapped at virtual address KERNBASE (from KERNBASE to KERNBASE+4MB) in this address space. The next few instructions will re-initialize the stack and the program counter to virtual addresses, and the kernel will operate in the virtual address space from there on (leaving the bottom 2GB of virtual address space for user processes).
Line 1054 pushes the value of the `stack` variable into `esp`. Note that this will be a virtual address value (greater than KERNBASE), just like all other variables in the kernel. Because the address space in that region is mapped, this value will point to a valid address.
Finally, lines 1060-1061 branch to `main`, which is also a virtual address (greater than KERNBASE). After this instruction, the execution remains in virtual address mode.
Notice that the kernel used a temporary page table (`entrypgdir`) to switch from the physical address space to the virtual address space. This temporary page table had both physical-side mappings (0-4MB) and virtual-side

mappings (KERNBASE to KERNBASE+4MB) for the same physical addresses (0-4MB). This allowed transitioning from one address space to another.

After the transition has taken place, the kernel will install a new page table which will only have virtual-side mappings. All virtual addresses below KERNBASE will be used for user processes from there on.

Let's look at the `main()` function at line 1217. This function initializes the physical memory, page tables, I/O devices, and other things, before initializing the first user process (line 1239) and going into an infinite scheduling loop thereafter (which schedules the available processes). Because xv6 is a multiprocessor OS, the first CPU that has booted also initializes the other CPUs in the system (line 1237).

Virtual Memory review, Initializing page tables for user processes

Virtual Memory Review

VA = seg:offset. All instructions have default seg. The default seg can be overridden by specifying it explicitly.
xv6 initializes all segs to 0:ffffffff, so we have a flat address space. All translation/protection done through paging.
Segmentation is primarily used to specify the current privilege level (last two bits of CS register). Thus we need separate segment descriptors for user and kernel.
The processor boots in physical address space (16-bit 8086 mode).
As soon as it switches on 32-bit mode, segmentation hardware becomes active. However because the base of all segments is set to zero, the physical address is the same as the "offset".
The kernel enables paging. As soon as the paging hardware becomes active, all addresses are translated through the page tables. In the first page table (`entrypgdir`), the kernel maps the first 4MB of VA space identically to the first 4MB of PA space. Assuming that the loaded kernel is less than 4MB, at the time of enabling paging, all virtual addresses are translated identically to physical addresses, and so everything works as before.
In the first page table (`entrypgdir`), the kernel also maps the first 4MB starting at KERNBASE (0x80000000) to the first 4MB of PA space. This space above KERNBASE, will be the kernel's virtual address space in future.
The kernel switches its EIP and ESP registers to point to the kernel's virtual address space. From then on, the kernel operates purely out of its virtual address space (above KERNBASE), and the identity mapping (below KERNBASE) can be removed and used for user processes.
All symbols (variables, pointers) within the kernel image are given values in the kernel's virtual address space (above KERNBASE), so all future execution through de-referencing those variables will remain in the kernel address space.
Interesting Note: Even the symbols in the entry code of the kernel (`entry.S`) were given virtual addresses (above 0x80000000) by the linker. However, when we jumped to the entry code from the bootloader, we ran it from its physical addresses. So this is the only exception, when code compiled for one address space (virtual address space) is run using its physical addresses.

Initializing page tables for user processes

```
big picture of xv6's virtual addressing scheme
  [diagram]
  0x00000000:0x80000000 -- user addresses below KERNBASE
  0x80000000:0x80100000 -- map low 1MB devices (for kernel)
  0x80100000:?          -- kernel instructions/data
  ?         :0x8E000000 -- 224 MB of DRAM mapped here
  0xFE000000:0x00000000 -- more memory-mapped devices

where does the paging h/w map these regions, in phys mem?
  [diagram]
  note double-mapping of user pages

main
  long sequence of calls to initialize subsystems
  first -- call kvmalloc to generate *another* page table

why another page table?
  map kernel only once, leaving space for low user memory
  map more physical memory, not just first 4 MB
  use 4KB pages instead of 4 MB pages

4 KB pages
  entrypgdir was simple: array of 1024 PDEs, each mapping 4 MB
  but 4 MB is too large a page size!
    very wasteful if you have small processes
    xv6 programs are a few dozen kilobytes
    4 MB pages require allocating full 4 MB of phys mem
  solution: x86 MMU supports 4 KB pages
```

```
how much space required for a flat page table w/ 4 KB pages?
  there are a million of them on a 32-bit machine
  32 bits per entry, so 4 MB for a full page table
  that also wastes lots of memory for small programs!
    you only need mappings for a few hundred pages
    so the rest of the million entries would be there but not needed

detailed structure of x86 page table
  see handout
  page table translates va to pa
    replacing top 20 bits in va with a [different] pa
    leaving bottom 12 bits alone
  logical view:
    a table of 32-bit PTEs
    2^20 PTEs, one for each 4096 bytes of address
    each PTE holds 20-bit pa of page
      or is marked invalid
    flags in low 12 bits
  implementation view:
    split page table into 4096-byte chunks ("page table pages")
    diagram
    each holds 1024 32-bit PTEs
    omit parts you don't need
    how to know where the parts of the table are? and which are valid?
  page-directory page
    1024 32-bit entries (PDE)
    each entry holds pa of a page-table page
      with entries for a range of 1024 pages
      or is blank (invalid)
    really top 20 bits of pa of page-table page
      pages are aligned, so bottom 12 bits always zero
    PDE holds some flags in those 12 bits
  %cr3 holds phys addr of page directory

how does x86 paging hardware translate a va?
  need to find the right PTE
  top 10 bits index PD to get address of PT
  next 10 bits index PT to get PTE
  flags in PTE
    P, W, U
    xv6 uses U to forbid user from using anything above 640 K



the page table is stored in DRAM
  thus xv6 has to allocate physical memory for PD and PTs
  each PD and PT fits in a physical page (4096 bytes) of memory

back to vm.c (sheet 17)
  main calls kvmalloc
  kvmalloc calls setupkvm

setupkvm
  creates a page table
  install mappings the kernel will need
  will be called once for each newly created process

setupkvm
  must allocate PD, allocate some PTs, put mappings in PTEs
  alloc allocates a physical page for the PD
    returns that page's virtual address above 0x8000000
    so we'll have to call V2P before installing in cr3
  memset so that default is no translation (no P bit)
  a call to mappages for each entry in kmap[]
```

```
what is kmap[]?
  an entry for each region of kernel virtual address space
  same as address space setup from earlier in lecture
  0x80000000 -> 0x00000000 (memory-mapped devices)
  0x80100000 -> 0x00100000 (kernel instructions and data)
  data -> data-0x80000000  (phys mem after where kernel was loaded)
  DEVSPACE -> DEVSPACE     (more memory mapped devices)
  note no mappings below va 0x80000000 -- future user memory there

mappages (sheet 16)
  (called by setupkvm for each kernel mapping range)
  arguments are PD, va, size, pa, perm
  adds mappings from a range of va's to corresponding pa's
  rounds b/c other uses pass in non-page-aligned addresses
  for each page-aligned address in the range
  call walkpgdir to find address of PTE
    need the PTE's address (not just content) b/c we want to modify

diagram of PD &c, as following steps build it

walkpgdir
  mimics how the paging h/w finds the PTE for an address
  refer to the handout
  a little complex b/c xv6 allocates page-table pages lazily
    might not be present, might have to allocate
  PDX extracts top ten bits
  &pgdir[PDX(va)] is the address of the relevant PDE
  now *pde is the PDE
  if PTE_P
    the relevant page-table page already exists
    PTE_ADDR extracts the PPN from the PDE
    p2v() adds 0x80000000, since PTE holds physical address
  if not PTE_P
    alloc a page-table page
    fill in PDE with PPN -- thus v2p
  now the PTE we want is in the page-table page
    at offset PTX(va)
    which is 2nd 10 bits of va

back to mappages
  put the desired pa into the PTE
  mark PTE as valid w/ PTE_P
  repeat for all pages in the range
```

Creating kernel's page table, Initializing physical memory allocator, Running Processes in Action

Creating kernel's page table

Initializing physical memory allocator

- Remember that setupkvm allocated phys mem for page directory.

How does memory allocation work?
sheet 27

- Physical memory allocator interface:

allocates a page at a time
va = kalloc()
kfree(va)
always allocates from phys mem above where kernel was loaded
so pa is va - KERNBASE

- what does kernel use phys mem allocator for?

page table pages
kernel data structures (pipe buffers, stacks, &c)
user memory

- How does the allocator work?

data structure: a list of free physical pages
allocation = remove first entry from list
free = add page to head of list
list's "next" pointers stored in first 4 bytes of each free page
that memory is available since they are free

- Allocator depends on phys pages having virtual addresses!

since it must write them to manipulate free list
just like VM code needed to write page table pages
thus xv6 maps all phys pages into kernel address space
which burns up a lot of virtual address space, limits max user mem
other arrangements are possible

- where does allocator get initial pool of free physical memory?

kinit1
called by main
end is first address beyond the end of the kernel *as a virtual address*
memory beyond that is unused

- freerange:

PGROUNDUP since newend may not be page-aligned

- Q: why must allocated pages have page-aligned addresses?
- kinit1 assumes physical memory goes up to 4MB. Hence P2V(4*1024*1024).
- Uses this space for allocation of initial data structures (e.g., first page table kpgdir
- kinit2 assumes physical memory goes up to PHYSTOP (lame)
- kfree each page

usually called on previously-allocated memory
kinit is abusing kfree a little bit

- kfree

linked list
note cast at 2827
2828 is where we depend on phys mem being mapped at a virt addr

- kalloc takes the first element of the free list

- Q: how to allocate mem for a data structure (e.g. array) > 4096 bytes?

Processes in Action

```
process execution states:
  diagram: user/kernel, process mem, kernel thread, kern stack, pagetable
  process might be executing in user space
    with cr3 pointing to its page table
    user mode, so can use PTE_U PTEs, < 0x80000000
  or might be in a system call in the kernel
    e.g. open() finding a file on disk
    process's "kernel thread"
    kernel mode, so can use non-PTE_U PTEs
    using kernel stack
  or not currently executing

xv6 has two kinds of transitions
  trap + return: user->kernel, kernel->user
    system calls, interrupts, divide-by-zero, &c
    hw+sw saves user registers on process's kernel stack
    save user process state ... run in kernel ... restore state
  process switch: between kernel threads
    one process is waiting for input, run another
      or time-slicing between compute-bound processes
    save p1's kernel-thread state ... restore p2's kernel-thread state

Q: why per-process kernel stack?
   what would go wrong if syscall used a single global stack?

how does xv6 store process state?
  struct proc sheet 20
  kernel proc[] table has an entry for each process
  discuss pgdir, kstack, state, pid fields.
        then discuss ofile field (fd table)

discuss how any system call works.
discuss how fork system call works.
discuss how external timer interrupt works.
discuss context switching

discuss proc.tf (trapframe)
discuss proc.context (for context switch)
```

Process Structure, Switching

```
process execution states:
  diagram: user/kernel, process mem, kernel thread, kern stack, pagetable
  process might be executing in user space
    with cr3 pointing to its page table
    user mode, so can use PTE_U PTEs, < 0x80000000
  or might be in a system call in the kernel
    e.g. open() finding a file on disk
    process's "kernel thread"
    kernel mode, so can use non-PTE_U PTEs
    using kernel stack
  or not currently executing

xv6 has two kinds of transitions
  trap + return: user->kernel, kernel->user
    system calls, interrupts, divide-by-zero, &c
    hw+sw saves user registers on process's kernel stack
    save user process state ... run in kernel ... restore state
  process switch: between kernel threads
    one process is waiting for input, run another
      or time-slicing between compute-bound processes
    save p1's kernel-thread state ... restore p2's kernel-thread state

Q: why per-process kernel stack?
   what would go wrong if syscall used a single global stack?

how system call works:
   user process uses software interrupt instruction
     int $0x80
   trapframe pushed on top of kstack
     partly by hardware (eip, cs, eflags, esp, ss)
     rest by software (first few instructions in handler are push instructions)
     handler calls the appropriate functions (e.g., syscall functions)
   syscall arguments passed from user to kernel in registers
     kernel function can access user register values in trapframe
   syscall return value passed in register (eax)
     kernel function overwrites trapframe->eax with the return value
   on syscall return (through iret), the trapframe is popped from kstack
     partly by software (the last few instructions before iret are pops)
     partly by hardware (iret pops eip, cs, eflags, esp, ss)

Discuss how syscall arguments and return values can be passed through pointers
  as the kernel and the user process are living in the same address space. This
  makes for very efficient communication between user and kernel, and justifies
  the 2GB of virtual address space that the kernel eats up from the
  process.

In an alternate organization, the kernel could have lived in a different
  address space, in which case, communication would have involved copying
  data from one address space to another. This has the advantage of
  strong isolation between different components (E.g., user/kernel), but
  are less efficient. "Microkernels" take this approach, and are used in
  safety-critical applications, like embedded systems.

if process calls "yield" syscall or if an external timer interrupt is received:
   if executing in user mode (definitely true for yield)
      hardware switches to kernel stack (per process) and pushes eip, cs, eflags, esp, ss at top
      first few instructions push of handler push the rest of user registers
      handler calls the appropriate functions (e.g., scheduler)
      scheduler decides if the current process should continue running?
      if so
          it simply returns from the scheduler
          eventually, returning to the user mode by popping the trapframe from kstack
          trapframe popped partly by software and partly by hardware (just as before)
      if not
         do context-switch:
         the scheduler switches to new process's kstack (which needs to run next)
         old process's kstack contains all information about its trapframe,
              and its callchain from handler to scheduler
         the old process kstack is linked through the its pcb (process control block)
         the saved kstacks are in a state such that switching to them starts running
              the process exactly from where it was preempted
              -  this means that they contain the eip value from where to resume (in kernel)
              -  they also contain the values of other registers, so execution can resume
                 as though it was never interrupted.
              -  typically, on resumption of the execution, the new process will return
                 from the scheduler, eventually returning to user mode by popping the
                 (saved) trapframe from kstack
              -  trapframe popped partly by software and partly by hardware (just as before)

   if executing in kernel mode
      hardware does not need to switch stacks. it simply pushes eip, cs, eflags
      handler saves other registers and calls scheduler (as  before)
      scheduler either returns or context-switches.
      if it returns, it simply resumes execution in kernel mode from where it
         was preempted
      it it context-switches, it saves kstack, so that future scheduling of
         this process can resume from where it was preempted (by returning from
         the scheduler, as though it was never switched-out).


Thus, a saved kstack contains a trapframe at its top
somewhere it also contains a "context" frame, which is used
    to save resume kernel-mode execution
A kstack may contain more than one trapframes
  - for example, if a timer interrupt was received in the middle of kernel execution

kernel stack diagram (saved):
  top ->
              esp, ss
              eip, cs
              ...
              gs fs es ds
  p->tf ->    edi & 7 other registers
              ---

              ... (information about function call chain inside kernel)
              ... (e.g., return address, arguments, local vars, etc)

              ---
              eip
              ebp
              ebx
              esi
  p->context -> edi
              ---

              ...
  p->kstack ->  ...
  (bottom)
```

Overheads of Timer Interrupt Handling

Let's say, you have written a fancy program that really cares about
its speed. Also, assume that when this program is running on the system,
it is the only program on the system. So, whenever a timer interrupt is
received, execution switches to kernel mode, where the scheduler is called,

```
only to realize that it is the only process that is running, and the
scheduler returns without switching the kernel stack (i.e., no context
switch). So, you are naturally worried about the unnecessary overhead
of handling the timer interrupt periodically, because it serves no
purpose (it merely returns to executing the same process).

Let's look at the overhead of timer interrupt handling:

Typical timer interrupt frequencies: 10-100 Hz (i.e., every 10-100ms)

Approximate time to execute one instruction: 1-100 nanosecond (1-100ns)

Approximate number of instructions executed in one timer interval:
    10ms/1ns = 10ms/10ns = 10^6

Approximate number of instructions executed for handling the timer interrupt:
    say 100-1000 (conservatively 10^3)

Conservative overhead: 10^3/10^6 = 0.1%

Hence, the overhead is insignificant. In fact, it is possible to further
increase timer frequency (to say 1000Hz) without incurring noticeable
overheads. However, as we will discuss later in our discussions on
scheduling, timer frequency beyond a certain value is not very useful.
```

Fork, Process kstack, Context Switching

Process kstack

```
How to determine the kstack size?
  - look at max function call depth in your code
      corollary: do not use deep function call chains
  - look at max size of local variables in functions
      corollary: do not allocate large variables on stack. allocate
                 them on heap if needed.
  - look at size of trapframe, etc.

xv6 uses a 4KB kstack (per process). Linux uses 8KB kstacks.

BUT: we said that any trap causes a trapframe to get pushed
on the kstack. So if there are many external interrupts, one after
another, could the kstack overflow?
Solution:
- Ensure that the kernel cannot cause any software interrupt or
  exception while executing a handler.
- Ensure that handlers of external interrupts (e.g., timer, disk, network, etc.)
  always execute with interrupts disabled.

This ensures that there can be at most two trapframes on the kstack: the first
due to a software interrupt/exception, and the second due to an external
interrupt received while executing in the kernel. This allows you to calculate
an upper-bound on the stack size.

What happens if an interrupt is received while the CPU was executing with
disabled interrupts? The interrupt is ignored:
  - Typically, the interrupting hardware expects an acknowledgement from
    the CPU that it received the interrupt. If it does not receive an
    acknowledgement, it retries the interrupt
  - Also, the CPU has a buffer (of size 2 say) where it stores pending
    interrupts (i.e., interrupts that were received but not serviced).
    As soon as the CPU re-enables interrupts, the pending interrupts are
    delivered to it.

How does an OS keep time? One common way is to count the number of timer
interrupts received.
```

Fork

```
how fork system call works:
  the fork function creates a new PCB and a new kstack (child's)
  it copies the trapframe from the current kstack to the child's
  it changes the return value (eax) in the trapframe for the child
  it also initializes a few more entries on the child's kstack
     so that it can control what are the first few instructions that run
     when the child gets scheduled.
  it adds the PCB to the list of schedulable processes, and returns.


There is one kstack per CPU, which is used by the scheduler thread (the
  first thread to run on that CPU). This kstack is not associated with
  any process. Let's call this the scheduler's kstack.

At bootup, the CPU initializes itself to run with its scheduler kstack.
   Now it will allocate the first process (including its kstack), initialize
   it, and switch from the scheduler's kstack to the new process's kstack.
   The new process will get to run, and will likely call exec/fork to
   create more processes.

Each time a process wants to yield (either voluntarily or preemptively),
   it switches to the scheduler's kstack. The scheduler then finds a process
   to run, and switches to its kstack.

The swtch function on sheet 27 switches kstacks. Let's look at it.
   It saves the current registers on stack
   It then saves the current esp into its first argument (struct context **old)
   It loads the new esp from its second argument (struct context *new)

   The kstack of every suspended process looks like it has just been
     switched out from inside the swtch function.

   Similarly, when a process is running on the CPU, the kstack of the
     scheduler (stored in global variable cpu->scheduler) also looks like
     it has just been switched out using the swtch function.

swtch sheet 26
  save current thread's registers and stack
  load new threads stack and registers
  saves current registers on current stack, struct context, sheet 20
  expects new thread's stack to have registers in that format
  stack diagram:
    eip *****
    ebp
    ebx
    esi
    edi
  Q: why these registers?
    callee saved, might have caller's live variables
  same format as struct context
```

Process structure

```
how does xv6 store process state?
  struct proc sheet 20
  kernel proc[] table has an entry for each process
  discuss pgdir, kstack, state, pid fields.
  then discuss ofile field (fd table)

discuss alltraps and trapret on sheet 30
  in the course of a regular trap, trapret gets pushed to stack
     by the call instruction.
  in the case of a newly forked process (or the first process), the
     stack is initialized to contain trapret just below the trap
     frame.
  Why do we save all registers (and not just callee-saved regs) here?
```

Process Creation

```
now let's talk about creating first process and its address space

Q: you'd expect there to be an array or something of pointers
   to the process's user memory. where is it? how does xv6
   know what memory a process is using?

ordinarily p->pgdir and phys mem contents created by fork()
  we will fake them for first process

ordinarily p->tf and p->context created by syscall/switch
  we will fake them for first process

main calls userinit

userinit sheet 22
  only called for first process
    other processes created by fork
  mimics fork+exec
    create a normal-looking process
    ordinary scheduler will run it
  needs to fill in all struct proc entries

allocproc sheet 22
  used by both fork and userinit
  kernel stack setup:
    trapframe w/ "saved user registers"
      for us, initial user registers
      eax, eip, esp, &c
    trapret !
    context w/ "saved kernel thread registers"
      for us, initial kernel thread registers
      eip
    assumes that execution will resume in a special
       assembly function called swtch (sheet 27) at line 2722,
       where there is code to pop the registers
       from stack and execute the return instruction
  Q: where will new kernel thread start executing?
  doesn't set up trapframe b/c ordinarily copied
    from parent by fork, which calls allocproc
  but fork and userinit both always start thread in forkret

trapframe sheet 06
context sheet 20

kernel stack diagram:
  top ->
                esp, ss
                eip, cs
                ...
                gs fs es ds
  p->tf ->      edi & 7 other registers
                ---
                trapret
                ---
                eip = forkret
                ebp
                ebx
                esi
  p->context -> edi
                ---
  p->kstack ->  ...

Q: any guesses why there are *two* saved EIPs?

back to userinit
```

```
    we know setupkvm -- only fills in kernel mappings
    this is a new page table for the new process
      not using it yet, will switch when new kernel thread starts
    call inituvm w/ ptr to new process's user instructions

  initcode.S sheet 77 : becomes _binary_initcode_start
    user program
    exec("/init", args)

  init.c sheet 78 :
    initializes fds 0, 1, 2
    forks shell and waits for it to exit.
    if shell exits, init exits too.

  inituvm sheet 18
    we know kalloc and mappages(pgdir, va, sz, pa)
    initcode is tiny, fits in one page
    diagram: new mapping

  Q: new page is mapped at va=0; could inituvm call memmove(0, init, sz)?

  back to userinit sheet 22
    tf->esp -- user stack at top of page
    tf->eip=0 -- first instruction at bottom of page

  main calls scheduler() sheet 12

  scheduler sheet 24
    no longer initialization: kernel now fully running
    whenever process gives up CPU -> scheduler
    so kernel runs scheduler a lot
    look for a process that wants to run, run it
    p->state: SLEEPING, RUNNABLE, RUNNING
    scheduler looks for RUNNABLE

  switchuvm sheet 17
    tell h/w to use p->stack if re-enters kernel
      sys call or interrupt
    load %cr3

  let's watch switch to new process's page table:
    (gdb) break switchuvm
    (gdb) x/5i 0
    0x0:    Cannot access memory at address 0x0
    next past load %cr3
    (gdb) x/5i 0
    same as initcode sheet 75
    but we are still in the kernel, in scheduler

  back to scheduler sheet 24
    mark RUNNING so no other CPU runs it
    now switch to new process's kernel stack, registers, EIP
    swtch(place to save current ESP, previously saved ESP to switch to)

  let's watch:
    (gdb) break swtch
    si until esp switch...
    (gdb) x/6x $esp
    si past esp switch
    (gdb) x/6x $esp
    after: 4 regs, forkret, trapret

  step into forkret sheet 24
    just returns
    allocproc set up stack to have it return to trapret
    watch out:
      release and initlog cause interrupts
      so hack source to set first=0 and pushcli
      si for iret &c -- si leaves interrupts off
    next into trapret

  look at trapframe sheet 06
    (gdb) x/19x $esp
    0x0 0x23 are eip:cs
    0x1000 0x2b are esp:ss

  trapret sheet 29
    pops trapret registers from stack, mostly zero
    popal pops 8 general-purpose registers
    iret pops ESP, EIP, clears supervisor flag
    x/5x $esp
    now we are executing at address 0x0 in initcode sheet 75

  what does initcode do?
    traps back into the kernel to make exec() system call
```

Handling User Pointers, Concurrency

```
why do we need to take special care for user -> kernel?
  security/isolation
  only kernel can touch devices, MMU, FS, other process' state, &c
  think of user program as a potential malicious adversary
```

System call dispatch, arguments and return value

`trap()` calls syscall(), since trapno in this case is T_SYSCALL (0x40).

syscall() dispatches to a function it finds by indexing into the syscalls array. It uses the eax from the trap frame as the index. What is in that eax? Where was it set?

Now we're in sys_open(). Where are the arguments the user program originally passed to `open()`? How can the kernel get at them?

sys_open() calls argint() to get its 2nd argument. Argint calculates the value `cp->tf_esp + 4 + 4*n`. What is this? Why the first 4? Why the `4*n`?

fetchint() checks that the address is not beyond the end of user memory. But addr was just calculated by kernel code (in argint()); since the kernel code is trustworthy, is this check really neccessary?

Why do we do seemingly redundant checks for addr and then addr+4? Can't we just check addr+4?

Why does fetchint() add `p->mem` to addr?

Back to sys_open(). It does its job (which we will talk about later) and finally returns a file descriptor using the ordinary C return statement. syscall() puts that return value in `cp->tf->eax`. Why?

Trap Return

In Unix, traps often get translated into signals to the process. Some traps, though, are (partially) handled internally by the kernel -- which ones?

Device Interrupts

trap(), when it's called for a time interrupt, does just two things: increment the ticks variable, and call wakeup. At the end of trap, xv6 calls yield. as we will see, may cause the interrupt to return in a different process!

Locking

Why locks?

The point of a multiprocessor is to increase total performance by running different tasks concurrently on different CPUs, but concurrency can cause incorrect results if multiple tasks try to use the same variables at the same time. Coordination techniques prevent concurrency in situations where the programmer has decided that serial execution is required for correctness.

Diagram: CPUs, bus, memory.

ide.c has a linked list.

List and insert example:

```
struct List {
  int data;
  struct List *next;
};

List *list = 0;

insert(int data) {
  List *l = new List;
  l->data = data;
  l->next = list;  // A
  list = l;        // B
}
```

Whoever wrote this code probably believed it was correct, in the sense that if the list starts out correct, a call to insert() will yield a new list that has the old elements plus the new one. And if you call insert(), and then insert() again, the list should have two new elements. Most programmers write code that's only correct under **serial** execution - in this case, insert() is only correct if there is only one insert() executing at a time.

What could go wrong with concurrent calls to insert(), as might happen on a multiprocessor? Suppose two different processors both call `insert()` at the same time. If the two processors execute A and B interleaved, then we end up with an incorrect list. To see that this is the case, draw out the list after the sequence A1 (statement A executed by processor 1), A2 (statement A executed by processor 2), B2, and B1.

This kind of error is called a race (the name probably originates in circuit design). The problem with races is that they are difficult to reproduce. For example, if you put print statements in to debug the incorrect behavior, you might change the timing and the race might not happen anymore.

What's required for insert() to be correct if executed on multiple CPUs? The overall property we want is that both new elements end up appearing in the list. One way to achieve that is to somehow ensure that only one processor at a time is allowed to be executing inside `insert()`; let one of the simultaneous calls proceed, and force the other one to wait until the first has finished. Then two simultaneous executions will have the same result as two serial executions. Assuming `insert()` was correct when executed serially, it will then be correct on a multiprocessor too.

This happens whenever there are two independent computations on *shared state*. e.g., you edit `foo.c` using `vi` and compile it using `gcc`. As a matter of habit we save the source file and wait for the confirmation message before starting the compile job. If we did not wait for the save to finish and started compile simultaneously, what could happen?

Concurrency in real world: one computer, many jobs; one road, many cars; one classroom, many classes. Different solutions for each.

Another example: lost update.

Increment number of hits on a website in each thread. `hits` is a shared variable.

```
hits++;
```

This C statement may get compiled to multiple assemble statements:

```
ld  register, [hits]
add register, register, 1
st  register, [hits]
```

If two threads simultaneously execute this code, what are the possible values of `hits`?

More race condition fun:

```
Thread a:
i = 0;
```

```
while (i < 10)
    i = i + 1;
print "A won!";

Thread b:
i = 0;
while (i > -10)
    i = i - 1;
print "B won!";
```

Who wins? Guaranteed someone wins?

Possible solutions:

- Do nothing: In some cases, this may be acceptable. For example, a website may not care if some updates to `hits` are lost, because this event is likely to be rare. But the same cannot be said for other examples. Even rare races can be menacing.
- Don't share, but duplicate state: Sometimes possible. For example, each thread could maintain a private `hits[i]` counter which can be summed up at the end of the day
- General solution? Prevent *bad* interleavings

The lock abstraction

We'd like a general tool to help programmers force serialization. There are a number of things we'd like to be able to express for lists:

1. It's not enough to serialize just calls to insert(); if there's also delete(), we want to prevent a delete() from running at the same time as an insert().
2. Serializing *all* calls to insert() is too much: it's enough to serialize just calls that refer to the same list.
3. We don't need to serialize all of insert(), just lines A and B.

We're looking for an abstraction that gives the programmer this kind of control. The reason to allow concurrency when possible (items 2 and 3) is to increase performance.

Locking

The lock abstraction

We'd like a general tool to help programmers force serialization. There are a number of things we'd like to be able to express for lists:

1. It's not enough to serialize just calls to insert(); if there's also delete(), we want to prevent a delete() from running at the same time as an insert().
2. Serializing *all* calls to insert() is too much: it's enough to serialize just calls that refer to the same list.
3. We don't need to serialize all of insert(), just lines A and B.

We're looking for an abstraction that gives the programmer this kind of control. The reason to allow concurrency when possible (items 2 and 3) is to increase performance.

While there are many coordination abstractions, locking is among the most common. A lock is an object with two operations: acquire(L) and release(L). The lock has state: it is either locked or not locked. A call to acquire(L) waits until L is not locked, then changes its state to locked and returns. A call to release(L) marks L as not locked. So when you have a data structure on which operations should proceed one at a time (i.e. not be interleaved), associate a lock object L with the data structure and bracket the operations with acquire(L) and release(L).

```
Lock list_lock;

insert(int data){
  List *l = new List;
  l->data = data;

  acquire(&list_lock);

  l->next = list ; // A
  list = l;        // B

  release(&list_lock);
}
```

How can we implement acquire() and release()? Here is a version that **DOES NOT WORK**:

```
struct Lock {
  int locked;
}

void broken_acquire(Lock *lock) {
  while(1){
    if(lock->locked == 0){ // C
      lock->locked = 1;    // D
      break;
    }
  }
}

void release (Lock *lock) {
  lock->locked = 0;
}
```

What's the problem? Two acquire()s on the same lock on different CPUs might both execute line C, and then both execute D. Then both will think they have acquired the lock. This is the same kind of race we were trying to eliminate in insert(). But we have made a little progress: now we only need a way to prevent interleaving in one place (acquire()), not for many arbitrary complex sequences of code (e.g. A and B in insert()).

Atomic instructions

We're in luck: most CPUs provide "atomic instructions" that are equivalent to the combination of lines C and D and prevent interleaving from other CPUs.

For example, the x86 `xchg %eax, addr` instruction does the following:

1. freeze other CPUs' memory activity for address addr
2. temp := *addr
3. *addr := %eax
4. %eax = temp
5. un-freeze other memory activity

The CPUs and memory system cooperate to ensure that no other operations on this memory location occur between when `xchg` reads and when it writes.

We can use `xchg` to make a correct acquire() and release() (you can find the actual syntax in the xv6 source):

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(Lock *lock){
  while(1){
    if(xchg(&lock->locked, 1) == 0)
      break;
  }
}

void release(Lock *lock){
  lock->locked = 0;
}
```

Why does this work? If two CPUs execute `xchg` at the same time, the hardware ensures that one `xchg` does both its steps before the other one starts. So we can say "the first `xchg`" and "the second `xchg`". The first `xchg` will read a 0 and set lock->locked to 1 and the acquire() will return. The second `xchg` will then see lock->locked as 1, and the acquire will loop.

This style of lock is called a spin-lock, since acquire() stays in a loop while waiting for the lock.

Spinning vs blocking

Spin-locks are good when protecting short operations: increment a counter, search in i-node cache, allocate a free buffer. In these cases acquire() won't waste much CPU time spinning even if another CPU holds the lock.

Spin locks are not so great for code that has to wait for events that might take a long time. For example, it would not be good for the disk reading code to get a spin-lock on the disk hardware, start a read, wait for the disk to finish reading the data, and then release the lock. The disk often takes 10s of milliseconds (millions of instructions) to complete an operation. Spinning wastes the CPU; if another process is waiting to execute, it would be better to run that process until the disk signals it is finished by causing an interrupt.

xv6 has sleep() and wakeup() primitives that are better for processes that need to wait for I/O.

Locking

Locking Discipline

Consider the following code where the system maintains multiple accounts (e.g., bank accounts) and provides a function to transfer one unit of money from one account to another.

```
#define N 100
struct account {
  int account_id;
  int money;
};

struct account accounts[N];

bool transfer(struct account *src, struct account *dst)
{
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    return true;
  }
  return false;
}
```

This code is correct if run in a sequential manner, but can result in many problems if multiple threads could execute it simultaneously. Here are two of the many problems that could occur:

- If multiple threads execute `transfer()` on the same `dst` account, there is a race condition at statement C, where an update can be lost (similar to how it was lost in the `hits` example discussed previously).
- If multiple threads execute `transfer()` on the same `src` account, there is a race condition at statements A and B. For example, it is now possible for an account to have negative money: consider the case where the `src` account initially has one unit of money. The first thread checks if `src` has any money; simultaneously, the second thread checks if `src` has any money; both succeed, and both decrement the money eventually leading resulting in the final value of -1 for `src->money`.

We need to use locks to fix this code. One option is to use a global lock for all accounts (*coarse-grained* locking) in the following way:

```
#define N 100
struct account {
  int account_id;
  int money;
};

struct account accounts[N];
struct lock global_lock;

bool transfer(struct account *src, struct account *dst)
{
  acquire(&global_lock);
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    release(&global_lock);
    return true;
  }
  release(&global_lock);
  return false;
}
```

This code is correct. However, it is not necessarily efficient, as it serializes *all* calls to the `transfer()` function. For example, if two threads are operating on separate accounts (one on accounts A and B, and another on accounts C and D), then they do not need to be serialized.

A better option may be to use *finer-grained* locks. In doing so, the programmer must ensure that whenever an account's data is touched, the corresponding lock is held. This will allow much better concurrency and throughput in the system. However, as we will see next, fine-grained locking can be tricky.

We first discuss some incorrect ways of using per-account locks, before converging on the correct implementation. In our first try, the programmer tries to use separate locks for `src` operations and `dst` operations, namely `global_lock1` and `global_lock2`.

```
//ATTEMPT 1
#define N 100
struct account {
  int account_id;
  int money;
};

struct account accounts[N];
struct lock global_lock1, global_lock2;

bool transfer(struct account *src, struct account *dst)
{
  acquire(&global_lock1);
  if (src->money > 0) {     // A
    src->money--;           // B
    release(&global_lock1);
    acquire(&global_lock2);
    dst->money++;           // C
    release(&global_lock2);
    return true;
  }
  release(&global_lock1);
  return false;
}
```

Here, `global_lock1` is intended to protect operations on `src` account, and `global_lock2` is intended to protect operations on `dst` account. However, this code is hopelessly incorrect. Consider the case, when one thread wants to transfer money from account 0 to account 1, and the second thread wants to transfer money from account 1 to account 0.

```
void thread1(void) {
   ...
   transfer(&accounts[0], &accounts[1]);
   ...
}

void thread2(void) {
   ...
   transfer(&accounts[1], &accounts[0]);
   ...
}
```

For `thread1`, `src` refers to `accounts[0]` and `dst` refers to `accounts[1]`. For `thread2`, it is vice-versa. In this case, if the threads run concurrently, both threads could be accessing the same account (e.g., account 0) at the same time, resulting in a race condition. For example, thread1 could be at statement B while thread 2 could simultaneously be at statement C, both operating on the same account (account 0), which will result in potentially incorrect behaviour (e.g., lost update).

This problem occurred because we associated locks with function arguments, but arguments can be aliased to different accounts at the same time. A better strategy will be to associate locks with the accounts themselves, by using per-account locks. Here is another attempt at writing correct code.

```
//ATTEMPT 2
#define N 100
struct account {
  int account_id;
  int money;
  struct lock lock;
};

struct account accounts[N];

bool transfer(struct account *src, struct account *dst)
{
  acquire(&src->lock);
  if (src->money > 0) {     // A
    src->money--;           // B
    release(&src->lock);
                            // B1
    acquire(&dst->lock);
    dst->money++;           // C
    release(&dst->lock);
    return true;
  }
  release(&src->lock);
  return false;
}
```

This code is almost correct, but it has a problem. At statement B1, no locks are held, but the total amount of money in the system appears to be less than its actual value (by one). In other words, this transfer operation is *not* atomic. A race-free implementation of `transfer()` would look like the following:

```
bool transfer(struct account *src, struct account *dst)
{
  acquire(&src->lock);
  acquire(&dst->lock);
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    release(&src->lock);
    release(&dst->lock);
    return true;
  }
  release(&src->lock);
  release(&dst->lock);
  return false;
}
```

Consider another thread running the `sum()` function below:

```
int sum(void)
{
  int i, ret;
  ret = 0
  for (i = 0; i < N; i++) {
    ret += accounts[i].money;
  }
  return ret;
}
```

Clearly, this function is also accessing shared data (`accounts`), and should thus use locks to ensure correctness. One way to ensure correctness is to take the locks for all accounts, before proceeding with the computation of sum, like this:

```
int sum(void)
{
  int i, ret;
  ret = 0

  for (i = 0; i < N; i++) {
    acquire(&accounts[i].lock);
  }
  for (i = 0; i < N; i++) {
    ret += accounts[i].money;
  }
  for (i = 0; i < N; i++) {
    release(&accounts[i].lock);
  }
  return ret;
}
```

This will ensure that the computation of `sum` is atomic with respect to other operations (e.g., `transfer`) in the system. In general, the following locking discipline ensures race-freedom:

- Associate a lock with every shared region (granularity is determined by the choice of shared regions and locks).
- Ensure that before accessing a shared region, the corresponding associated lock is held.
- If an atomic operation involves accessing multiple shared regions, acquire the locks of all the shared regions apriori. Release all these locks after the operation is finished.

In our examples on `transfer()` and `sum()`, this discipline ensured race-freedom.

However, there is a second big problem with locks, namely *deadlocks*. In the previous example, consider the situation where `thread1` attempts to transfer money from account0 to account1, while `thread2` attempts to transfer money from account1 to account0. Hence, `thread1` will first acquire account0's lock, and then account1's lock. Similarly, `thread2` will first acquire account1's lock, and then acquire account0's lock. In a bad schedule, it can so happen that before thread1 acquires account1's lock (after acquiring account0's lock), thread2 acquires account1's lock. Thus, thread1 will have to wait for thread2 to release account1's lock before it proceeds further. However thread2 will next attempt to acquire account0's lock and will have to wait for thread1 to release it.

At this stage, both thread1 and thread2 will be waiting for each other to release a lock that they need. And so they will keep waiting forever (as none of them will be able to proceed to the release statement). This is an example of a deadlock.

In an operating system the usual solution is to avoid deadlocks by establishing an order on all locks and making sure that every thread acquires its locks in that order. In this example, the rule may be to lock the account with the lowest account ID first. Picking and obeying the order on *all* locks requires that modules make public their locking behavior, and requires them to know about other module's locking. This can be painful and error-prone. Here is the correct code which is both race-free and deadlock-free.

```
bool transfer(struct account *src, struct account *dst)
{
  if (src->account_id < dst->account_id) {
    acquire(&src->lock);
    acquire(&dst->lock);
  } else {
    acquire(&dst->lock);  //reverse the order
    acquire(&src->lock);
  }
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    release(&src->lock);
    release(&dst->lock);
    return true;
  }
  release(&src->lock);
  release(&dst->lock);
  return false;
}

int sum(void)
{
  int i, ret;
  int s[N];
  ret = 0

  //sort the accounts and store the sorted order in s[]
```

```
   s = sort_by_account_id(accounts);

   for (i = 0; i < N; i++) {
     acquire(&accounts[s[i]].lock);
   }
   for (i = 0; i < N; i++) {
     ret += accounts[i].money;
   }
   for (i = 0; i < N; i++) {
     release(&accounts[s[i]].lock);
   }
   return ret;
 }
```

Locks and modularity

When designing a system we desire clean abstractions and good modularity. We'd like a caller not to have to know how a callee implements a particular function. We'd like to hide locking behavior in this way -- for example, it would be nice if the lock on each file-system inode was the private business of methods on inodes.

It turns out it's hard to use locks in a modular way. Two problems come up that require non-modular global knowledge: multi-module invariants and deadlock.

First, multi-module invariants. You could imagine the code implementing directories exporting various operations like `dir_lookup(d, name)`, `dir_add(d, name, inumber)`, and `dir_del(d, name)`. These directory operations would each acquire the lock on `d`, do their work, and release the lock. Then higher-level code could implement operations like moving a file from one directory to another (one case of the UNIX `rename()` system call):

```
move(olddir, oldname, newdir, newname){
  inumber = dir_lookup(olddir, oldname)
  dir_del(olddir, oldname)
  dir_add(newdir, newname, inumber)
}
```

This code is pleasingly simple, but has a number of problems. One of them is that there's a period of time when the file is visible in neither directory. Fixing that, sadly, seems to require that the directory locks *not* be hidden inside the `dir_` operations, thus:

```
move(olddir, oldname, newdir, newname){
  acquire(olddir.lock)
  acquire(newdir.lock)
  inumber = dir_lookup(olddir, oldname)
  dir_del(olddir, oldname)
  dir_add(newdir, newname, inumber)
  release(newdir.lock)
  release(olddir.lock)
}
```

The above code is ugly in that it exposes the implementation of directories to `move()`, but (if all you have is locks) you have to do it this way.

The second big problem is deadlock. Notice that `move()` holds two locks. What would happen if one process called `move(dirA, ..., dir B, ...)` while another process called `move(dirB, ..., dir A, ...)`?

In an operating system the usual solution is to avoid deadlocks by establishing an order on all locks and making sure that every thread acquires its locks in that order. For the file system the rule might be to lock the directory with the lowest inode first. Picking and obeying the order on *all* locks requires that modules make public their locking behavior, and requires them to know about other module's locking. This can be painful and error-prone.

Fine-grained locks usually make both of these problems worse.

Some terms

Locks are a way to implement "mutual exclusion", which is the property that only one CPU at a time can be executing a certain piece of code. This is also called "isolation atomicity".

The code between an acquire() and a release() is sometimes called a "critical section."

The more general idea of ensuring one thread waits for another when neccessary for correctness is called "coordination." People have invented a huge number of coordination primitives; locks are just one example, which we use because they are simple. We'll see some more coordination primitives next.

Spin Lock Subtleties

The `xchg R,M` instruction is expensive because it usually involves a bus transaction, to ensure that the read and write both happen atomically. Here is a computer system diagram to explain this better:

```
                                 -------------
                                | Main Memory |  (containing l->locked)
                                 -------------
                                       |
                                       |
        --------|-----------------|----------------|------
        --------|-----------------|----------------|------   Bus
        --------|-----------------|----------------|------
                |                                  |
                |                                  |
              cache0                             cache1
             --------                           --------
             | CPU0 |                           | CPU1 |
             --------                           --------
```

Each CPU usually has a private L1 cache. Consistency between the caches is maintained using a "cache coherence protocol", whereby the system ensures that each memory location has only one valid value at any time.

Thus if CPU0 access location X, a 100 times (without any intervening access to X by CPU1), the first access will result in a bus transaction (whereby X will be fetched either from the main memory, or from CPU1's cache through the cache coherence protocol). All the other 99 accesses will be served locally from cache0, without incurring a bus transaction. In a multiprocessor system, the bus often becomes a performance bottleneck (especially if the number of CPUs is high), and so reducing the number of bus transactions is an important optimization.

In our implementation of `acquire()` and `release()`, we use the atomic `xchg` instruction that needs to perform two memory accesses (one read and one write) in an atomic fashion. This can only be achieved by a bus transaction, because all other processors need to be informed that the two memory accesses need to be done atomically. Hence, if a CPU is spin-waiting inside a lock acquire function, it is constantly making bus transactions, which can severely affect performance.

```
void acquire(struct lock *l) {
  Reg = 1;
  while (xchg(Reg, &l->locked) == 1);
}
void release(struct lock *l) {
  l->locked = 0;
}
```

Instead, it may be better to write the code like this:

```
void acquire(struct lock *l) {
  Reg = 1;
  while (xchg(Reg, &l->locked) == 1) {
    while (l->locked == 1);
  }
}
void release(struct lock *l) {
  l->locked = 0;
}
```

In this case, while waiting, we use a regular memory read instruction (in the inner loop) to test the current value of `l->locked`. For the time that `l->locked` is not changed by the other CPU, `l->locked` will get cached into cache0, and the accesses by a CPU will be only to its local cache (thus avoiding bus transactions). If the local cache reports that the value has changed (which means another CPU must have changed its value), this code again tries the `xchg` instruction, which may succeed this time. If it fails again, we again go into the inner waiting loop. This is a more efficient method of waiting as it reduces bus transactions significantly. Modern processors provide special instructions (e.g., `pause` on x86) to make this waiting operation even more efficient.

Effect of Compiler/Architecture Optimizations

Compiler optimizations typically do not work well with code that can be executed concurrently. Using locks, we have ensured that the only code that can execute concurrently are the `acquire()` and `release()` functions. Let's see what could happen if the compiler tries to optimize our `acquire()` and `release()` implementation.

Register allocation of variables

Register allocation of a variable involves reading the variable from memory into a register, and then performing the operations on the register locally, before writing the register back to memory. For example, in the following code, the variable `a` gets register allocated to register `R`.

```
a = 1;
b = a + 2;
a = a + 1;
```

may get compiled to

```
load  a, Reg     //load 'a' from memory to register
b = Reg + 2
Reg = Reg + 1
store Reg, a     //store current value of 'a' from register to memory
```

Intelligent register allocation reduces the number of memory accesses, and thus reduces memory accesses. But what would happen if the `l->locked` variable in our `acquire()` function gets register allocated. It will result in an infinite loop inside acquire, causing a deadlock.

Because compiler optimizations assume sequential code, they do not play well with code that could be executed concurrently. Languages typically provide ways to tell the compiler to not optimize accesses to certain variables (for example, the `volatile` keyword in C). In this case, we should either write the waiting loop in assembly, or use the volatile keyword appropriately to ensure that the `l->locked` variable does not get register allocated.

Reordering of Memory Accesses

Usually, compilers are free to reorder accesses to "independent" memory locations. For a compiler, two locations are independent, if they are different. For example, it is perfectly valid for a compiler to invert the order of accesses to variables `a` and `b` like this:

```
a = 2;
b = 3;
```

becomes

```
b = 3;
a = 2;
```

For a compiler which only looks at sequential semantics, this is a valid optimization. Similarly, even if the compiler does not reorder the instructions, the hardware is free to reorder the memory accesses at runtime. For example, if the access to variable `"a"` is a cache miss, while the access to variable `"b"` is a cache hit, the access to `"b"` will complete first. This is true for most modern processors, and this behaviour is called "out-of-order" execution.

In our lock implementation, if the access to a shared variable gets reordered with the access to the lock variable (`l->locked`), our semantics get violated. For example,

```
acquire(l);
hits++;
release(l);
```

may become equivalent to (either at compile-time or at runtime)

```
acquire(l);
release(l);
hits++;
```

To avoid this, it is possible to tell the compiler and the hardware to not reorder instructions. Many architectures that support out-of-order execution, also provide "`fence`" instructions, which can be used to instruct the hardware to not reorder memory accesses across the fence instruction. On the x86 architecture, the locked `xchg` instruction implicitly also acts as a fence, so our `acquire` implementation is correct even in the presence of memory access reordering. However, we need to explicitly insert a `fence` instruction in our `release()` implementation like this:

```
void acquire(struct lock *l) {
  Reg = 1;
  while (xchg(Reg, &l->locked) == 1) {
    while (l->locked == 1);
  }
}
void release(struct lock *l) {
  fence;   //assembly instruction
  l->locked = 0;
}
```

Spinlock implementations inside the kernel

For a uniprocessor kernel, spin locks are not needed. Instead, mutual exclusion can be ensured by clearing (and re-enabling) insterrupts around before (and after) the critical section.

For a multiprocessor kernel, spin locks as implemented above suffice for ensuring mutual exclusion among different CPUs. However, this still does not protect a CPU from its own interrupt handler. For example, if an interrupt can occur within a critical section, and the interrupt handler could access the shared data (which was also being accessed within the critical section), mutual exclusion would get violated. On xv6, this is true for the process table `ptable`, for example, where the timer handler may need to inspect the `ptable` and may find it in an inconsistent state, if the timer interrupt occurred in the middle of an access to the `ptable`. Worse, if the timer handler also tries to acquire the `ptable.lock`, it would result in a deadlock.

To deal with this, xv6's spinlock also disables the interrupts on a call to `acquire`, and reenables them on a call to `release`. To allow a thread to acquire multiple locks simultaneously, the interrupts are enabled and disabled in a recursive manner, as follows:

```
void acquire(struct lock *l) {
  Reg = 1;
  pushcli();
  while (xchg(Reg, &l->locked) == 1) {
    while (l->locked == 1);
  }
}
void release(struct lock *l) {
  fence;   //assembly instruction
  l->locked = 0;
  popcli();
}
```

The `pushcli()` function uses the `cli` instruction internally to disable interrupts. The `popcli()` function uses the `sti` instruction internally to re-enable interrupts. They also maintain a per-CPU counter `cpu->ncli` to correctly handle nested calls to `pushcli()` and `popcli()`.

```
void pushcli(void) {
  if (cpu->ncli == 0) {
    cli;
  }
  cpu->ncli++;
}
void popcli(void) {
  cpu->ncli--;
  if (cpu->ncli == 0) {
    sti;
  }
}
```

Spinlock implementations in the user-mode

In the usermode, spinlocks work exactly like they work in kernel-mode, except that they do not need to disable and enable interrupts. Kernel's interrupt handlers are not expected to touch user data, so disabling interrupts is not required. Notice that the `xchg` instruction is an unprivileged instruction, and has identical semantics in both user and kernel modes.

Blocking Locks

Blocking locks involve changing the status of the current thread from RUNNING to BLOCKED in the process table (or the list of PCBs, depending on how the processes are maintained). For xv6, the global `ptable` array is the process table structure. A kernel thread that needs to wait on a lock `acquire`, will set its status to "BLOCKED on lock l", and call the scheduler. Similarly, when a thread calls `release(l)`, it scans the `ptable` and changes the status of one of the blocked processes (if any), blocked on `l` from BLOCKED to READY.

Notice that the accesses to the `ptable` also need to be mutually exclusive. This is done using a spinlock (`ptable.lock` in the case of xv6). Hence, a blocking lock typically uses a spinlock internally. Of course, the spinlock is released, as soon as the ptable is updated.

Blocking locks at user-level

At the user-level, blocking locks need to suspend a thread. If the threads are kernel-level threads, this involves making a system call to tell the kernel to block on a lock acquire. Similarly, release involves making a system call to unblock one of the threads blocked on that lock.

If the threads are user-level threads however, all this can be done entirely in userspace, without any kernel involvement. For user-level threads, the `ptable` itself will be maintained (and manipulated) at the user-level.

Locking Variations

Recursive locks

Our current lock abstraction does not allow a code path which acquires the same lock twice --- execution of this code path will cause a deadlock. For example, as we discussed previously, if an interrupt occurs in the middle of a critical section, and the interrupt handler tries to acquire an already-held lock, a deadlock would result.

One proposal to solve this issue is to allow the same thread to acquire a lock multiple times, but still disallow different threads to acquire the lock at the same time. Here is an example implementation of recursive locks, which uses regular locks internally.

```
void recursive_acquire(struct rlock *rl) {
  if (rl->owner == cur_thread) {
    rl->count++;
  } else {
    acquire(&rl->lock);
    rl->count = 0;
    rl->owner = cur_thread;
  }
}
void recursive_release(struct rlock *rl) {
  rl->count--;
  if (rl->count == 0) {
    release(&rl->lock);
    rl->owner = -1;
  }
}
```

Usually, recursive locks are a bad idea. Most programmers maintain the invariant that a critical section always begins in a consistent state, i.e., immediately after successfully returning from a lock acquire, the state of the system is consistent. Similarly, they maintain the invariant that the state of the system is left in a consistent state on a lock release, i.e., the state is consistent just before a call to `release()`. (By consistency, we mean that certain invariants hold about the program state. For example, in our bank account program, consistency may mean that the total money in the bank is constant).

Recursive locks encourage the programmer to allow a critical section to begin in an inconsistent state. Consider a function `foo()` which acquires a lock `l`, and then calls another function `bar()`. The call to the function `bar()` may be synchronous (i.e., the programmer has a call chain in her program that eventually leads to `bar` from `foo`), or asynchronous (e.g., if `bar()` is called within an interrupt handler, and the interrupt is received within `foo()`). In either case, `bar()` may be assuming that the state is consistent when it begins its critical section. However, because the function was called in the middle of the critical section of `foo()`, `bar` may unexpectedly see inconsistent state.

```
foo() {
  acquire();
  ....
  bar();   //at this point, state is inconsistent
  ....
  release();
}

bar() {
  acquire();
  ....      // assumes that if acquire succeeds, the state here is consistent.
  ....
  release();
}
```

Because recursive locks encourage such bugs, they are not considered very useful.

Try locks

Another locking variation is a try lock, where the `acquire()` function returns a SUCCESS or a FAILURE value, depending on whether the acquisition was successful or not. Here is a sample implementation of a trylock.

```
bool try_acquire(struct trylock *tl) {
  Reg = 1;
  xchg(Reg, &tl->locked);
  if (Reg == 1) {
    return false;  //FAILURE
  } else {
    return true;   //SUCCESS
  }
}
void try_release(struct trylock *tl) {
  fence;
  tl->locked = 0;
}
```

The caller of `try_acquire` may decide its action depending on the return value. A regular acquire can be implemented trivially by calling `try_acquire` in a loop till it succeeds. However, `try_acquire` gives the flexibility to the caller to do something else, if the lock is not currently available.

An Example of a Webserver

Consider a webserver that receives an HTTP request (e.g., fetch URL), and replies with the contents of the corresponding HTML file. Also, let's say that the HTML files are stored in the server's disk. Multiple clients can connect to the webserver simultaneously.

Let us first understand the typical hardware characteristics. The webserver is possibly running on a modern processor executing at 1-2 GHz frequency, i.e., each instruction executed locally on the CPU (e.g., register access, L1 cache hit, etc.) takes around 1-2 nanoseconds to complete. Instructions that miss the cache, and thus need to access the main memory take 10-100 nanoseconds. In contrast, a disk access takes roughly 1-10 milliseconds to complete, i.e., around one million instructions can execute in the time it takes to complete one disk access!

Why is a disk access so expensive? As we will also see later in our discussion on file systems, a magnetic disk is a mechanical device organized as a cylindrical spindle, with multiple circular platters stacked one above another. Each platter in the cylinder has concentric tracks, which hold the data as magnetic impressions. To read this data, a magnetic disk has a *disk head* associated with each platter that can move radially and position itself on a particular track. The cylindrical spindle rotates around its central axis so the head can read data off the track, as the spindle rotates.

Notice that reading data involves mechanical radial movement of the disk head arm, and also requires the head to be positioned at the right point within a track, which involves a rotational latency. The time taken for the radial movement of the head to reach the desired track, is called the *seek time*. Typically, average seek times range between 4 milliseconds (for enterprise drives) to 15ms (for mobile drives). Typical disks rotate at 5000 RPM (for mobile drives) to 15000 RPM (for enterprise drives), and so the average rotational latencies range between 2-5 milliseconds. Overall a disk access can take 5-20 milliseconds to complete.

```
                 <----->
                 moves radially
         |   v------|
     ---------      |
         |          |
 platter |   v------|
     ---------      |
         |          |disk head controller
         |   v------|
     ---------      |
         |          |
         |   v------|
     ---------      |
         |          |
       spindle
       rotates
```

Let's look at the pseudo-code of a webserver:

```
while (1) {
  read message from incoming network queue;
  URL = parse message;
  read URL file from disk;
  write reply on outgoing network queue;
}
```

Here we assume that there is an incoming network queue, where incoming packets are buffered, and read in FIFO order. If multiple clients are accessing the webserver, their requests will get buffered in the incoming queue in arrival order. This webserver picks and serves one request at a time. Because, each request needs to go to the disk device, and assuming each disk request takes 10 milliseconds, we can serve 100 requests per second. Modern webservers serve tens of thousands of requests per second. We can make this system faster by introducing a cache of disk files in memory.

```
while (1) {
  read message from incoming network queue;
  URL = parse message;
  if (requested file is in cache) {
    data = read URL file from disk;
    new.name = URL;      //new is the current cache pointer.
    new.data = data;
    new = new + 1;       //mark the current cache pointer as used.
  }
  write reply on outgoing network queue;
}
```

This server is much better, as we have optimized for the common case where there is locality of access, i.e., the same set of pages are accessed close in time, and most of these accesses can be served from memory.

However, still, if even one request results in a disk access (cache miss), then all the following requests in the network queue will have to wait for a long time (even though those requests could have been served much faster from memory). So, let's use multiple threads, each serving one request.

```
for (i = 0; i < 10; i++) {
  fork(server);
}

server() {
  while (1) {
    read message from incoming network queue;
    URL = parse message;
    if (requested file is in cache) {                 // A
      data = read URL file from disk;
      //new is the current cache pointer.
      new.name = URL;                                 // B
      new.data = data;                                // C
      //mark the current cache pointer as used.
      new = new + 1;                                  // D
    }
    write reply on outgoing network queue;
  }
}
```

Each thread executes the server function, but now we have concurrency problems. Firstly, assuming there is only one network queue, multiple server threads will be reading from the same input queue, and need correct synchronization for that. We will be discussing this later. Similarly, the output network queue is shared by multiple server threads. Let's however first focus our attention to the shared cache, accesses to which are made in statements A, B, C, and D.

Many race conditions exist in this code and many bad things can happen. We can have a cache with an empty entry, or a bad entry, or worse still, a page belonging to URL1, cached under URL2.

One way to solve this problem is to use a global cache lock, and instrument the entire critical section with it (statements A, B, C, and D). However, that brings us back to the same problem (which we wanted to solve using multiple threads), which is that even if one thread misses the cache and accesses the disk, all other concurrent threads will have to wait (on the lock acquire statement) for a long time. (Notice that we have just changed the waiting point from the input queue to the lock acquisition). So, a full solution will require fine-grained locking (e.g., lock per file/page, a lock for the disk, etc.), and will need careful lock acquisition for correctness.

Let's now look at the shared network queues. On the incoming queue, data (or packets) is *produced* by a "network thread" (or multiple network threads) and *consumed* by the "server threads". The network thread reads the incoming packets from the network wire and appends it to the queue's buffer (which is consumed in FIFO order by the server thread). Similarly, on the outgoing queue, data is produced by the server threads, and consumed by another set of outgoing network threads. This is a common situation where there is a shared queue and there are producer and consumer threads accessing this queue. This is also commonly known as the *producer-consumer* problem.

```
network() {
  packet = read from wire;
  queue.head = packet;
  queue.head++;
}

server() {
  packet = queue.tail;
  queue.tail++;
  read packet and process request;
}
```

Queues are often implemented as circular buffers, with `head` and `tail` pointers, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global

void produce(char data) {
  if ((head + 1) % MAX  ==  tail) {
    error("producing to full queue!");
    return;
  }
  queue[head] = data;
  head = (head + 1) % MAX;
}

char consume(void) {
  if (tail == head) {
    error("consuming from empty queue!");
    return -1;
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  return e;
}
```

The code above implements `produce()` and `consume()` functions for a queue implemented as a circular buffer. Additions to full queues, and consumption from empty queues are not allowed, in this case.

In a multi-threaded scenario, one of the threads may be consuming from the queue, while another thread may be producing to the queue. Also, if a consumer tries to consume from an empty queue, it may want to wait till the queue becomes non-empty, and then consume an element. Similarly, if a producer wants to produce to a full queue, it may want to wait till the queue becomes not-full, and then produce the element, in the following way:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global

void produce(char data) {
  if ((head + 1) % MAX  ==  tail) {
    wait for queue to have some empty space;
  }
  queue[head] = data;
  head = (head + 1) % MAX;
}

char consume(void) {
  if (tail == head) {
    wait for queue to have some elements;
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  return e;
}
```

How can a thread wait for a condition to become true? For example, how does a thread wait for the queue to have some empty space? This requires some coordination between the producer and the consumer threads. Such coordination is not possible using locks, as locks only implement mutual exclusion. This is done using a special construct, called *condition variables*.

Just like locks, the condition variables are also variables, usually shared by multiple threads, and they provide functions called `wait()` and `notify`.

```
struct cv;   //condition variable type

void wait(struct cv *, struct lock *);

void notify(struct cv *);
```

(Let's ignore the second argument of the `wait` function, of type `lock`, for some time). The `wait` function causes the calling thread to get suspended (or blocked). It is woken up, only if another thread subsequently calls `notify` on the same condition variable.

In our producer-consumer example, the producer thread may wait on a condition variable, if it finds that the buffer is full. The consumer thread, after consuming an element, may call `notify` to wakeup any sleeping producer thread (as the queue may now be not-full), in the following way:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;

void produce(char data) {
  if ((head + 1) % MAX  ==  tail) {
    wait(&not_full, ...);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_empty);
}

char consume(void) {
  if (tail == head) {
    wait(&not_empty, ...);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_full);
  return e;
}
```

Here, we declared two condition variables, `not_full` and `not_empty`. If the queue is full, the producer thread waits on `not_full`. The consumer thread calls `notify(not_full)` if it consumes an element. The effect of `notify` is to wakeup all threads waiting on that condition variable. If no threads are currently waiting on that condition variable, `notify` does not have any effect. Also, notice that we have left the second argument of the `wait` calls blank for now.

Firstly, this code is incorrect because the threads are concurrently reading shared variables, `queue`, `head`, and `tail`, and so bad interleavings can cause incorrect results. We need to use locks to fix this. Secondly, we do not need to call `notify` on every produce and consume operation. Instead, notify only needs to be called if the queue transitioned from full to not-full in the case of the consumer (or empty to not-empty in the case of the producer). Let's try fixing this code, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  if ((head + 1) % MAX  ==  tail) {
    wait(&not_full, ...);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  if (head == (tail + 1) % MAX) {   // the queue just became not-empty
    notify(&not_empty);
  }
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  if (tail == head) {
    wait(&not_empty, ...);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  if ((head + 2) % MAX == tail) {    // the queue just became not-full
    notify(&not_full);
  }
  release(&qlock);
  return e;
}
```

Notice that now, if a producer starts waiting on the condition variable, the system will deadlock, as it has acquired the lock and so no other thread will be able to access the queue and so the producer will never get notified. A better solution may perhaps be to release the lock before calling wait, and reacquire it afterwards, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  if ((head + 1) % MAX  ==  tail) {
    release(&qlock);
    wait(&not_full, ...);
    acquire(&qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_full);
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  if (tail == head) {
    release(&qlock);
    wait(&not_empty, ...);
    acquire(&qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_empty);
  release(&qlock);
  return e;
}
```

This code is also incorrect. Let's see what can happen: it is possible that the producer thread checks the queue, finds it full, and so releases the lock, and is about to call `wait()`. However, before the producer calls `wait()`, the consumer thread, can acquire the lock, consume an element, notify `not_full`, and release the lock. In this case, the notify call will be lost, as the producer has not called wait yet. When the producer calls wait after this, it will keep sleeping forever, as the consumer will not call notify again, resulting in a deadlock. This is also called the *lost-notify* problem.

The lost-notify problem occurred because the release of the lock and the act of going to sleep were not atomic. Hence, the `wait()` function takes an additional argument, `lock`. Internally, `wait()` releases the lock before going to sleep (in an atomic manner), and tries to reacquire the lock after being notified. The wait function ensures that no other thread can call notify between the release of the lock and the act of going to sleep. The (almost) correct code looks like the following:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  if ((head + 1) % MAX  ==  tail) {
    wait(&not_full, &qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_full);
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  if (tail == head) {
    wait(&not_empty, &qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_empty);
  release(&qlock);
  return e;
}
```

Notice that, now we do not need the calls to `acquire()` and `release()` around the calls to `wait()`, as they are called internally by `wait` itself. In this code, if a producer finds the buffer full, it goes to sleep, releasing the lock atomically. A consumer which consumes an element can then call notify to wakeup the producer. The woken-up producer is now ready to run (though it may not immediately get the CPU). The first thing that the producer does is to try to reacquire the lock (inside wait). Because the lock may be already held by the consumer (notice that the call to `release(&qlock)` is after the call to `notify`), and so it may have to wait again. As soon as the lock is released, the producer may get the lock and can continue to consume the next element.

Semantics of wait() and notify()

`wait(cv, lock)` releases the `lock` and goes to sleep on `cv`. `notify(cv)` wakes up *all* threads that are currently waiting on that cv. After waking up, the `wait()` function tries to reacquire the `lock` before returning.

Multiple producers and consumers

Interestingly, the producer-consumer code presented previously works if there is only one producer and one consumer thread. However, if there are multiple producers or multiple consumers, this code fails. For example, if two producers try to produce to a full queue, both of them may start waiting on `not_full`. If a consumer consumes one element, and calls `notify`, both producers will wakeup. Each of them will, in turn, try to acquire `qlock` and produce an element. However, the consumer only consumed one element, and the two producers will produce two elements, causing a queue overflow!

The problem is that the producer threads, do not check the condition after waking up from a `wait()`, and simply assume that the condition is true. This problem can be fixed by requiring the threads to check the condition again after returning from `wait`. This can be done by replacing the `if` condition with a `while` condition, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  while ((head + 1) % MAX  ==  tail) {
    wait(&not_full, &qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_full);
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  while (tail == head) {
    wait(&not_empty, &qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_empty);
  release(&qlock);
  return e;
}
```

A thread that has just woken up from `wait()` will again check the condition (because of the `while` construct). If the condition is true, it will go back to sleep (by another call to `wait()`). If the condition is false, it will fall through and perform its operation. This code is correct, even for multiple producers and multiple consumers.

In general, usage of condition variables has the following pattern:

```
struct lock mutex;
struct cv condition_variable;

acquire(&mutex);
while (!condition) {
  wait(&condition_variable, &mutex);
}
...
release(&mutex);
```

Here, the `condition_variable` is associated with the `condition` becoming true. The mutex variable is a lock used for mutual exclusion.

Semaphores

A common pattern in programming, as also seen in the producer-consumer example, is counting of resources. To simplify such patterns, Dijkstra proposed an abstraction called semaphores in 1965.

So far, we have studied locks, which are used for mutual exclusion, and condition variables, that are associated with conditions. Semaphores are usually used for "counting". Here are the semantics of a semaphore.

Semantics

A semaphore is defined by a stateful type (`struct sema;`) and three functions:

```
void sema_init(struct sema *, int val);
void P(struct sema *);   // also called wait()
void V(struct sema *);   // also called signal()
```

The names 'P' and 'V' stand for the dutch names of these operations. The semaphore type maintains an integer state, called `count`, which is a non-negative counter.

The `sema_init` function initializes the counter to `val`, which must be a non-negative integer.

The `P` function decrements the counter by one, if it is strictly positive. If the counter is 0 (before the decrement operation), then the `P` function waits for the counter to become positive (greater than zero), before decrementing it.

The `V` function increments the counter by one. Also, it wakes up one process that is waiting on the semaphore to become positive.

Most importantly, all these operations, `sema_init`, `P`, and `V` are atomic with respect to each other.

Here is a sample implementation of a semaphore using locks and condition variables:

```
struct sema {
  int count;
  struct lock lock;
  struct cv cv;
};

void sema_init(struct sema *sema, int val) {
  sema->count = val;
  lock_init(&sema->lock);
  cv_init(&sema->cv);
}

void P(struct sema *sema) {
  acquire(&sema->lock);
  while (sema->count == 0) {
    wait(&sema->cv, &sema->lock);
  }
  sema->count--;
  release(&sema->lock);
}

void V(struct sema *sema) {
  acquire(&sema->lock);
  sema->count++;
  if (sema->count == 1) {
    notify(&sema->cv);
  }
  release(&sema->lock);
}
```

Notice that a semaphore maintains state. This is unlike a condition variable which maintains no state. For condition variables, the effect of a `notify()` operation is to wakeup any currently waiting threads. If a thread has not yet gone to sleep, `notify()` will have no effect. On the other hand, the `V()` operation (that is somewhat analogous to `notify()`) will increment the counter, even if no thread is currently waiting inside the `P()` operation (which is somewhat analogous to `wait()`). Let's see the implications of this using concrete examples.

Producer-consumer example with semaphores

The producer consumer problem involves ensuring that a producer does not produce to a full queue, and a consumer does not consume from an empty queue. This can be done by using two counters, one for the number of filled slots in the queue (`nchars`), and another for the number of empty slots in the queue (`nholes`). We use one semaphore for each counter, as follows.

```
char q[MAX];
int head = 0, tail = 0;
struct sema nchars, nholes;
struct lock mutex;

sema_init(&nchars, 0);
sema_init(&nholes, MAX);

void produce(char c) {
  P(&nholes);
  acquire(&mutex);

  //produce an element.

  release(&mutex);
  V(&nchars);
}

char consume(void) {
  P(&nchars);
  acquire(&mutex);

  //consume an element.

  release(&mutex);
  V(&nholes);
}
```

Notice that this code is much simpler, as it subsumes much of the counting and waiting logic within `P()` and `V()` functions. Because this pattern is common, semaphores are a convenient abstraction.

Resource allocation with semaphores

Semaphores are also a convenient abstraction for performing resource allocation. For example, if you have multiple threads that want to access a resource, and you only have N copies of that resource (e.g., N printers), then you could use a semaphore to restrict the number of concurrent accesses to N.

Here is an example where multiple threads may call the `print()` function, but you want to restrict the maximum number of concurrent prints to N.

```
sema_init(&nprinters, N);

void print(void) {
  P(&nprinters);
  //print command
  V(&nprinters);
}
```

This will ensure that the system has at most N simultaneous print requests. Also, as print requests get completed, they allow other print requests to be admitted.

Locks as semaphores

Locks can be trivially implemented as semaphores by initializing it to 1.

```
struct lock {
  struct sema sema;
};

void lock_init(struct lock *l) {
  sema_init(&l->mutex, 1);
}

void acquire(struct lock *l) {
  P(&l->sema);
}

void release(struct lock *l) {
  V(&l->sema);
}
```

Scheduling with semaphores

Semaphores are also used for enforcing thread schedules. Consider the following example, where one thread computes `x`, a second thread computes `y`, a third thread computes `z = f(x, y)`, and finally a fourth thread prints `z`. You may want to enforce an execution order (schedule) such that the computation of `z` happens only after the computation of `x` and `y`. Similarly, the print statement should execute only after the computation of `z`. This can be achieved by using semaphores `sx`, `sy`, and `sz`, initialized to zero. A one value in these semaphore counters will be used to indicate that the corresponding value has been computed.

```
thread1() {
  ....
  x = ...;
  V(sx);
}

thread2() {
  ....
  y = ...;
  V(sy);
}

thread3() {
  ....
  P(sx);
  P(sy);
  z = f(x, y);
  V(sz);
}

thread4() {
  P(sz);
  print z;
}
```

Monitors

All our abstractions for synchronization so far (locks, condition variables, semaphores) are implemented as types and functions on them. It is the responsibility of the programmer to ensure that the type objects are correctly used --- for example, a programmer must ensure that a lock is properly initialized before it is used; she should also ensure that a lock acquisition is always appropriately bracketed with a lock release. Notice that this is completely independent of the language in which the program is written.

Because concurrent programs are quite common, some languages provide intrinsic support for handling concurrency. One example of such support is a *monitor*. A monitor is defined as a type construct (e.g., class), which may declare private shared-memory objects and routines that access those objects. The monitor enforces mutual exclusion between the monitor routines, by using an "implicit lock". Here is an example of the producer-consumer queue implemented as a monitor:

```
monitor Q {
        char buf[MAX];
  int head = 0, tail = 0;

  void produce(char c) {
              while ((head + 1) % MAX == 0) {
      wait(&not_full);
    }
              q[head] = c;
    head = (head + 1) % MAX;
    notify(&not_empty);
  }

        char consume(void) {
    ...
  }
}
```

Notice that because the monitor guarantees mutual exclusion between its routines (only one thread can be inside a monitor routine at any time), we did not have to use a mutex ourselves. Instead, the compiler will implicitly generate a mutex for us. One way to implement monitors, is for the compiler to instrument each function with `acquire()` and `release()` calls on the implicit lock, before and after the routine respectively. Also, notice that the `wait()` function does not need the second argument; instead the second argument is automatically understood to be the monitor's implicit lock, i.e., a thread calling wait() will internally release the monitor's implicit lock before going to sleep.

Monitors make it simpler to write concurrent programs, and avoid common bugs like unbracketed acquires or releases. In Java, the "synchronized" keyword can be used to specify a monitor.

Transactions

Transactions are another way of providing atomicity, and are significantly different from locks in their handling of concurrency. Recall that for atomicity, we need to ensure that two threads do not interleave accesses to shared data within a critical section. Locks ensure this by requiring the programmer to apriori announce her intention of accessing shared data, and the semantics of a lock prevent concurrent execution within a critical section. This is a *pessimistic* method of dealing with concurrency, because you always need to acquire the lock, irrespective of whether another thread actually tried to enter the critical section or not. This is analogous to saying that whenever you enter a room where you need privacy, you must always first lock it from inside, before you start using the room.

A more optimistic method may be to just enter the room (without locking). If another thread tries to enter the same room, it will likely notice that somebody else is already using the room and will rollback its execution (or return from the room in the physical analogy). This is more optimistic because you assume that in the common case, it is unlikely that somebody will try to enter the room at the same time. Hence, you have avoided the overhead of having to acquire a lock. In software, this is done by enclosing the act of accessing the shared resource in a *transaction*. All accesses to the shared resource within a transaction are *tentative*. If during the accesses, the thread notices that the same resource is being concurrently accessed by another thread (thus violating atomicity), the transaction is *rolled back*. Let's look at some examples.

Bank account example with transactions

Recall the bank account example where the `transfer()` function was used to transfer a unit of money from one account to another. Let's recall the code which used fine-grained locking.

```
void transfer(account *a, account *b) {
  if (a->account_id < b->account_id) {
    acquire(&a->mutex);
    acquire(&b->mutex);
  } else {
    acquire(&b->mutex);
    acquire(&a->mutex);
  }
  if (a->money > 0) {
    a->money--;
    b->money++;
  }
  release(&a->mutex);
  release(&b->mutex);
}
```

In this code which uses fine-grained locks, whenever we access a shared account, we always acquire the corresponding lock (mutex) before the access. However, in the common case, it is unlikely (though possible) that another thread will be accessing the same accounts at the same time. But we pay the overhead of lock acquisition each time!

It would be nice if we could structure our code as a transaction that can be rolled back if required, as follows:

```
void transfer(account *a, account *b) {
  int am, bm;

  do {
    tx_begin();
    am = tx_read(a->money);
    bm = tx_read(b->money);
    if (am > 0) {
      am--;
      bm++;
    }
    success = tx_commit(am, &a->money, bm, &b->money);
  } while (!success);
}
```

Here `tx_begin` indicates the start of a transaction, and `tx_commit` indicates the end of a transaction. At the end, the commit operation *atomically* tries to update the values of shared variables `a->money` (with value stored in local variable `am`), and `b->money` (with value stored in local variable `bm`). The commit succeeds, if no concurrent transaction had read these shared variables during the time the transaction was executing. It fails otherwise.

Notice that to implement an atomic commit (and rollback if needed), the runtime system needs to track, which locations were read and written. Also notice that this mechanism of a transaction is only useful if we expect that the probability of a commit failure is small. Because if there are likely to be a lot of rollbacks, then the performance of this system may actually be worse than that of coarse-grained locks.

The probability of a commit failure depends on the size of the transaction and on the expected concurrency of this code region.

Transactions are very useful for databases, where the read and commit operations are performed on disk blocks. Because disk accesses are anyways slow, it is relatively cheap to maintain this information about which disk blocks were read or written (in memory). However, in the case of operating systems, maintaining such information for memory bytes is much more expensive. Recently, new processors (e.g., Intel's Haswell) are providing hardware support for implementing transactions for memory accesses (also called *transactional memory*).

Compare and Swap (List insert example, hits example)

While most code today uses locks for atomicity, there is a special class of operations that have been using transactions for a long time. These are operations that can be written as atomic updates on a single memory location. The common primitive for single memory location transactions is a *compare-and-swap* instruction.

```
int cmpxchg(int *addr, int old, int new) {
  int was = *addr;
  if (was == old) {
    *addr = new;
  }
  return was;
}
```

In the instruction form, this can be written as:

```
cmpxchg Rold, Rnew, Mem
```

This instruction compares the value at location `Mem` with register `Rold`; if the values are equal, it replaces the value at `Mem` with `Rnew`. Else it does nothing. It returns the old value at location `Mem` in register `Rold`. This instruction is atomic if prefixed with the `lock` prefix..

For example, if we want to increment a shared variable `hits` atomically using `cmpxchg`, we can write the code as:

```
int l_hits, l_new_hits;
retry:
l_hits = hits;
l_new_hits = l_hits + 1;
if (cmpxchg(&hits, l_hits, new_l_hits) != l_hits) {
  goto retry;
}
```

Here, we first read the shared `hits` variable into a local variable `l_hits`. We perform some computation on `l_hits` to obtain a new value `l_new_hits`. Now, we wish to write the new value `l_new_hits` to the shared variable `hits`, but only if the `hits` variable has not been modified in between (i.e., between the first read to `hits` and this point). To do this, we can use the `cmpxchg` instruction on the memory address of shared variable `hits` with the old value as `l_hits` and the new value as `l_new_hits`. If the atomic `cmpxchg` instruction returns that the current read value of `hits` is still equal to its previous read value (`l_hits`), the increment operation was performed atomically. On the other hand, if the current read value of `hits` differs from the previous read value, it indicates a concurrent access to the `hits` variable, and so the operation needs to be retried (or rolled-back).

Simlarly, our list insert example can be written using compare-and-swap (or *CAS*) as follows:

```
void insert(struct list *l, int data) {
  struct list_elem *e;
  e = new list_elem;
  e->data = data;
retry:
  e->next = l->head;
  if (cmpxchg(&l->head, e->next, e) != e->next) {
    goto retry;
  }
}
```

Once again, if two threads try to insert into a list concurrently, one of them will execute the atomic `cmpxchg` instruction first. Whichever thread executes `cmpxchg` first, will succeed (as it will see the same current value of `l->head` as the one that it previously read). The successful thread will also atomically update the value of `l->head` to its new value (with the inserted element). The thread which executes `cmpxchg` second, will notice that the value of `l->head` has changed from its last read value, and this will cause a rollback.

Reader-Writer Locks

Finally, often there is a situation where some parts of the code *only* read shared data, while other parts of the code read and write to it. The threads that only read the data do not need to be serialized with respect to each other. In other words, it is okay to allow multiple reader threads to execute concurrently. But it is still incorrect to allow a writer thread to execute concurrently with a reader thread. It is also incorrect to allow a writer thread to execute concurrently with another writer thread.

To capture this programming pattern, an abstraction called *read-write locks* can be used. Read-write locks are defined as follows:

```
struct rwlock rwlock;

void read_acquire(struct rwlock *);
void write_acquire(struct rwlock *);
void release(struct rwlock *);
```

The `read_acquire` function acquires the `rwlock` in read mode, while the `write_acquire` function acquires the `rwlock` in read/write mode. The semantics are that an `rwlock` can be acquired in read mode multiple times simultaneously. However, it can only be acquired in write mode, if it is not currently held in either read or write mode by any other thread.

Locking in xv6

xv6 runs on a multiprocessor and allows multiple CPUs to execute concurrently inside the kernel. These usually correspond to system calls made by processes running on different CPUs. There might also be interrupt handlers running at the same time as other kernel code. An interrupt can occur on any of the CPUs; both user-level processes and processes inside the kernel can be interrupted. xv6 uses spin-locks to coordinate how these concurrent activities share data structures.

How do xv6 locks work? Let's look at acquire() in spinlock.c. [1474]

- what does pushcli() do? disables interrupts, increments per-CPU variable called ncli. release() calls popcli() which decrements counter and enables interrupts if counter equals zero.
- why does pushcli()/popcli() count nested calls? To ensure that interrupts are disabled only when all locks have been released.
- so why disable interrupts on the current processor while holding the lock? To guarantee atomicity with respect to the interrupt handlers.
- but won't the interrupt handlers also acquire a lock before accessing shared data? yes, they will but that's even worse because if a thread is already holding a lock and an interrupt occurs that tries to acquire the same lock, it will result in a deadlock.
- holding() is true if the current CPU already holds the lock. why is it bad for a CPU to re-acquire a lock it already has? because the lock implementation is not recursive.
- Wouldn't recursive locks be better? No, because that encourages bugs. For example, if interrupts were kept enabled within the critical section and recursive locks were allowed, an interrupt handler would be able to acquire the lock even if the interrupt occurred in the middle of the critical section --- bad!

And release():

- 1519: why not just lock->locked = 0? We need to ensure that instructions do not get reordered (either by compiler or by the hardware at runtime). The xchg instruction acts as an implicit barrier, preventing instructions from getting reordered across it.

Sleep/Wakeup

Sleep and wakeup are xv6's incarnation of condition variables. Here are the signatures of the `sleep()` and `wakeup()` functions:

```
void sleep(void *channel, struct lock *mutex);
void wakeup(void *channel);
```

The `sleep()` function is similar to the `wait()` function and the `wakeup()` function is similar to the `notify` function (recall condition variables). The difference is that instead of declaring a condition variable explicity (recall `struct cv`), we simply use any number (or void pointer) as our "channel" to wait on. Recall that because a condition variable is stateless, we do not really need to declare it, and can simply use any program variable's address to act as the "condition variable" (or channel, as called in xv6). The channel is just a number, so callers of sleep and wakeup must agree on a convention so they correctly synchronize between themselves.

The semantics of sleep()/wakeup() are identical to those of condition variables. The sleep() function goes to sleep on the channel releasing the mutex atomically, and the wakeup() function wakes up all threads sleeping on the channel. Below we describe the semantics of sleep and wakeup using code (assuming xv6 process table structure):

```
void sleep(void *chan, struct lock *lk):
  //begin{atomic}

  curproc->chan = chan       //curproc points to the PCB of current process
  curproc->state = SLEEPING
  release(lk);
  sched()                     //call the scheduler, which will context switch
                                to another process
  //end{atomic}

wakeup(chan):
  //begin{atomic}

  foreach p in ptable[]:      //ptable is an array containing all PCBs
    if p->chan == chan and p->state == SLEEPING:
      p->state = RUNNABLE

  //end{atomic}
```

The sleep() and wakeup() functions need to be atomic with respect to each other (also indicated by the begin-atomic and end-atomic annotations in the pseudo-code) and also with respect to other accesses to the process table. Recall that xv6 uses the `ptable.lock` to implement mutual-exclusion for accesses to the `ptable`.

```
void sleep(void *chan, struct lock *lk):
  acquire(&ptable.lock);
  curproc->chan = chan       //curproc points to the PCB of current process
  curproc->state = SLEEPING
  release(lk);
  sched()                    //recall that the scheduler (or the next process)
                             //releases ptable.lock

void wakeup(void *chan):
  acquire(&ptable.lock);
  foreach p in ptable[]:     //ptable is an array containing all PCBs
    if p->chan == chan and p->state == SLEEPING:
      p->state = RUNNABLE
  release(&ptable.lock);
```

Notice that it is also okay to release the lock `lk` before accessing `curproc` (as `ptable.lock` is now protecting accesses to `curproc`).

However, we need to ensure that our implementation cannot result in a deadlock (given that a thread can hold two locks at the same time). Firstly, we should check if `lk` is the same as `ptable.lock` --- if so, we do not need to reacquire it again. Secondly, and more importantly, we need to ensure that there is a global ordering on the lock acquisition. Notice that `ptable.lock` is acquired after the acquisition of `lk` (which must have been acquired by the caller of sleep()). The invariant followed by xv6 is that the `ptable.lock` will always be the inner-most lock, i.e., no other lock acquisition will be attempted while `ptable.lock` is held. This ensures that the `ptable.lock` is always the least priority, and so no deadlocks can result from cycles involving `ptable.lock`. Here is xv6's (correct) implementation of sleep and wakeup:

```
void sleep(void *chan, struct lock *lk):
  if (lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  curproc->chan = chan       //curproc points to the PCB of current process
  curproc->state = SLEEPING
  sched()                    //recall that the scheduler (or the next process)
                             //releases ptable.lock

void wakeup(void *chan):
  acquire(&ptable.lock);
  foreach p in ptable[]:     //ptable is an array containing all PCBs
    if p->chan == chan and p->state == SLEEPING:
      p->state = RUNNABLE
  release(&ptable.lock);
```

Notice that it is the programmer's responsibility to ensure that `wakeup()` is not called with `ptable.lock` held.

Use example: exit and wait

Recall API: Suppose process P1 has forked child process P2. If P1 calls wait(), it should return after P2 calls exit().

wait() in proc.c looks for any of its children that have already exited; if any exist, clean them up and return. If none, it calls sleep().

exit() calls wakeup(), marks itself as dead (ZOMBIE), and calls sched().

What lock should protect wait() and exit() from missing each other? `ptable.lock` as these functions manipulate the ptable.

What should be the channel convention? Using the process-ID of the waiting process seems like a good one. The waiting process will wait on its own process-ID. The exiting processes will wakeup processes waiting on their *parent's* process-ID.

What if we used a common channel for all processes? e.g., let's say we always used channel `(void *)1` (recall that a channel is just a number). Then exiting children of other processes may cause a waiting process to wakeup from sleep (inside wait). This may not be a correctness problem if the woken-up process again checks if it has any zombie children, and goes to sleep otherwise. However this causes poor performance, as all waiting processes will be woken up on every exit, and all but one of them will go back to sleep. Using the parent's process ID as the channel avoids this extra work.

What if some unrelated part of the kernel decides to sleep and wakeup on the same channel(s)? This will not be a problem from a correctness standpoint (irrelevant sleeping processes will wakeup only to go back to sleep). However this is not desirable from a performance standpoint.

Let's look at the xv6 code for wait and exit. Why does wait() free the child's memory (e.g., pgdir, kstack), and not exit()? Because the exit() function is currently using the process's pgdir and kstack, and so cannot deallocate them. Instead the process's parent can deallocate these structures inside wait.

Things to think about (relates to semantics and typical usage pattern of sleep/wakeup):

What if exit is called before wait?

What if wait is called just after exit calls wakeup?

xv6 kill

(sheet 26)

sets p->killed for the pid after taking `ptable.lock`.

What does setting p->killed do? Nothing, for now. But the process must be in the middle of something when this was done --- e.g., running in user mode, running in kernel mode, sleeping, etc. Whenever the process reaches a *safe* boundary, `p->killed` will be checked and `exit()` would be called. Notice that if the process was sleeping, kill() marks it RUNNABLE.

Where is killed checked? At syscall entry and exit (see trap function at lines 3104 and 3108). At other places where the process may have woken up from a sleep, and it is no longer correct to go back to sleep again.

Why not destroy the process right away? It is in the middle of something, destroying it will leave the kernel in an inconsistent state.

Why is it safe to mark the process RUNNABLE if it was SLEEPING? The programmer has ensured that the waiting loops that call sleep() within them, also check `p->killed` wherever necessary. e.g., the `wait()` functions waiting loop checks `proc->killed`.

Wouldn't it be okay to let the sleeping process keep sleeping, even if it was killed? Won't it anyways exit whenever it wakes up? This is true, and if you expect the process to wakeup in some bounded amount of time, then this may be okay. However, if the process could wait for an unbounded amount of time (e.g., parent waiting on child to exit), then this will not be acceptable. In general, all sleeping processes are woken up. Assuming that all calls to `sleep` are nested inside a `while` loop that checks the condition, it is possible that some of these processes go back to sleep again (where the sleep time is bounded anyways).

For example, the sleep loop inside `wait()` checks `p->killed` (because this sleep loop can wait for an unbounded time). However, the sleep loop inside the IDE device driver `iderw` does not check `p->killed`, which means that if it is woken up by `kill()`, it will go back to sleep again till the IDE request is not completed (see following discussion for more clarity on this).

ide device driver

(sheet 39)

The `iderw` function is called with an argument of type `struct buf *`, which represents a buffer in memory that will store the contents of the disk block. This function is used to read/write from/to the disk which has an IDE interface. The API for `iderw` specifies that if the B_DIRTY flag is set in `buf`, then the buffer needs to be written to the disk. Similarly, if the `B_VALID` flag is not set in `buf`, then the buffer needs to be read from the disk.

Multiple processes may be trying to access the disk through system calls (e.g., read/write). Also, disk requests could be made due to virtual memory's demand paging logic. All these accesses to the disk need to be mutually exclusive. So, xv6 uses a lock, called `idelock`.

Also, as we know, disk accesses are likely to be very slow. So, it is quite likely that multiple processes are simultaneously waiting for the disk. The right thing to do is to make the processes sleep, while they are waiting for the disk. However, before they go to sleep, they must register their request. These outstanding requests are maintained in a FIFO list, called `idequeue`.

The disk driver processes one request at a time, starting from the head of the list.

Let's look at how the IDE driver uses sleep and wakeup. ide_rw() starts a disk operation and calls sleep(). ide_intr() calls wakeup() when disk interrupts to say that it's done.

Why does it work? What if the disk finishes and interrupts just before ide_rw() calls sleep()? What if the disk interrupts during the execution of the interrupt handler?

What's the sleep channel convention? Why does it make sense?

Let's look at the IDE disk driver as an example of xv6's use of locks. Sheet 39. The file system kernel code calls ide_rw() to read or write a block of data from/to the disk. ide_rw() should return when the read or write has completed. There may be multiple calls to ide_rw() outstanding because multiple processes may be reading files at the same time. The disk can only perform one operation at a time, so the IDE driver keeps a queue of requests that are waiting their turn.

The IDE driver has just one lock (ide_lock). This one lock helps enforce multiple invariants required for correct operation:

- Only one thread should be inserting or deleting from ide_queue at a time.
- Only one thread should be commanding the IDE hardware (via inb/outb instructions) at a time.
- The disk hardware can only execute one read or write at a time.

The software (and hardware) for these activities was all designed for one-at-a-time use, which is why it needs locking.

What happens if two processes call ide_rw() at the same time from different CPUs? [3954]

ide_intr() deletes a request from ide_queue (line 3902). What happens if an IDE interrupt occurs when a process is adding a block to the queue?

Why does idestart() want the caller to hold the lock? Why doesn't it just acquire the lock itself?

recursive locks are a bad idea
ideintr should certainly not use that instead of disabling interrupts!

lock ordering
iderw: sleep acquires ptable.lock
never acquire ptable.lock and then ide_lock
Also never acquire ide_lock and then some other lock
So, `ptable.lock` has least priority (will always be the inner-most lock), ide_lock will have higher priority than `ptable.lock` but lower priority than all other locks in xv6.

Why not check p->killed in the ide_rw() loop? As also discussed earlier, this sleep loop is guaranteed to finish in bounded time. Also, the caller of `iderw` may be in an inconsistent state and may not expect `iderw` to return without actually reading the buffer. In this case, it is okay for some function higher in the stack to check `p->killed` and exit.

Demand Paging

Now, let's move back to our discussion on virtual memory. Recall that paging divides the address space into page-sized segments and allows a more flexible mapping between VA space and PA space. Also, the TLB acts as a cache for page table mappings and usually has high hit rates, that allow paging to be used without excessive overhead.

Further, previously we said that while loading a process, the executable file in `a.out` format is parsed, and all its memory contents are loaded into physical memory (from disk) and corresponding mappings created in the virtual address space (through the page table). In general, a loaded process may not necessarily access all its code/data and so, many disk reads can be avoided if the code/data pages are loaded on-demand.

Here, at program load time, you could load some pages in physical memory and create corresponding mappings in the page table. For others, you may mark them not-present, but also store somewhere that these pages are present at a location on the disk (along with the details of the location). Notice that now, some mappings in the page table are marked not-present, even if the program believes that it has loaded them. In other words, the OS is playing tricks with the program under the carpet, without the program's knowledge.

Previously, if a program tried to access a virtual address that is not currently mapped (i.e., the corresponding present bit is zero in the page table), an exception would get generated. This exception is also called a *page fault*. The corresponding OS exception handler (also called `page fault handler`) would get to execute and would likely kill the process. With demand paging however, the page fault handler will additionally check if this address is logically present (but physically not present due to its demand paging optimization), and if so, will load it from the disk to the physical memory on-demand.

After the OS loads the page from disk to physical memory, it creates a mapping in the page table, such that a page fault is not generated again on that address, and restarts the faulting instruction. Notice that in doing so, the OS assumes that it is safe to restart a faulting instruction. On the x86 platform, the hardware guarantees that if an instruction causes an exception, the machine state is left unmodified (as though the instruction never executed) before transferring control to the exception handler. This property of the hardware is called *precise exceptions*. If the hardware guarantees precise exceptions, it is safe for the OS to return control to the faulting instruction, thus causing it to execute again without changing the semantics of the process logic.

Demand Paging

A processes' page table at any time has two types of mappings, one to the physical memory and the other to the disk. The physical memory mappings are translated by the hardware. On access to a disk mapping, a page fault is generated, and handled by the operating system. The OS brings the page into the physical memory, and creates a physical memory mapping for the faulting address in the page table.

Physical memory is roughly 100 times costlier than magnetic disk storage, i.e., one byte of physical memory costs 100 times more than one byte of magnetic disk storage. Also, magnetic disks can be very large (100GBs-TBs); physical memories are smaller (10-100GBs). On the other hand, disk access is much slower than physical memory access (milliseconds vs. nanoseconds).

Another way to think is to consider the physical memory as a cache to the disk which maintains all the virtual memory contents. A cache hit costs around 10-100 nanoseconds, while a cache miss costs around 10 milliseconds (around million times slower). Thus, having a high hit-rate in the cache is crucial to the success of this demand paging system. As an example, if the hit rate was 90\%, then the average memory access time would be:

```
10% * 10ms + 90% * 100ns = 1ms
```

In other words, on average, every memory access would take 1ms, which is 10,000 times slower than the physical memory access time of 100ns. Thus, this hit rate is unacceptable. A hit rate of around 99.99% would be more realistic.

Fortunately, because most workloads exhibit a significant degree of spatial and temporal locality in their memory access patterns, high hit rates are usually possible. Thus through demand paging, the OS is able to provide disk-sized address space, that is as fast as physical memory.

The 90/10 rule: Spatial and temporal locality can also be articulated using the 90/10 rule, i.e., 10% of the memory addresses get 90% of the memory references. Given this, it would suffice if the OS ensures that the 10% *hot* memory addresses are usually present in physical memory, while the 90% *cold* memory addresses may be present on disk.

Figure with memory address on the X axis and number of references in the Y axis.

Cache characteristics:

- block size: one page. A very large block size may result in unnecessary data to be read in, and thus cause cache pollution. A very small block size may not be able to exploit spatial locality. More importantly, a small block size will not be able to amortize the cost of a disk seek/rotation over the amount of data read.
- organization or associativity: direct mapped/set-associative/fully-associative? Given that hit-rate is of prime importance, and that miss costs are very high, fully associative seems best. This is because fully-associative caches are likely to have the best possible hit-rates (no conflict misses). The drawback of implementing a fully-associative cache is that identifying the best candidate to replace is costlier (in direct-mapped there is only one choice). However, this cost at replacement time is relatively small as it only involves memory accesses --- recall that a disk access is necessary during replacement, and that dominates the replacement time overwhelmingly.
- hit or miss decision? check the present bit in page table entry in hardware. <1ns if TLB hit, 10-100ns if TLB miss (page table walk)
- hit path? if L1 cache hit, 1-2ns, else 10-100ns to access L2 cache/main memory (both in hardware).
- miss path? page fault handler, disk access (in software)
- replacement policy? sort-of LRU, needs more discussion.
- what happens on write? definitely write-back! (write-through would require a disk access on every write). How does the OS know at replacement time if the page needs to be written back to disk:
    - Option 1: always write back to disk on replacement. con: unnecessary disk accesses
    - Option 2: have a *dirty bit* in the page table entry. This bit is set by the hardware on a write access (notice that this needs to be set by hardware as it is on the hit path and needs to be fast). When

this page is first loaded by the OS, the corresponding dirty bit is set to zero, indicating that the page is clean, i.e., its contents are identical to the corresponding disk contents. At replacement time, the page needs to be written back to disk only if its dirty bit is set. The x86 architecture supports the dirty bit, and this mechanism is used for implementing write-back.

What to fetch initially and on-demand? Some options:

- Ask the user: not reliable
- Load some initial pages (based on common-case), e.g., first instruction, stack, and then do demand paging. Can do read-ahead (e.g., if fetch X, then also fetch X+1) or other forms of prefetching (e.g., if fetch X, then also fetch Y based on previous observations). Why is read-ahead useful? spatial locality and that the cost to read two pages from disk is not very different from the cost to read one page (dominated by seek time and rotational latency).

What to eject and when? Cache replacement policy

- Random: pro- easiest/fastest to implement, con- may replace a hot page
- FIFO: throw out oldest page (the page that was brought into the cache earliest).

  pro- easy/cheap to implement (just use an in-memory queue and insert/replace in page-fault handler), fair.

  con- ignores usage. A page that is being used often may get replaced just because it was brought in earlier, while a cold page that was brought-in later may keep occupying cache space.

- What could be the optimal policy? MIN: throw out page not used for longest time in future. A page that is not used for longest time in future is the best candidate for ejection, as all other pages will be used before this page. The problem is of course, that we do not know anything about the future. Hence, this is impractical, but nonetheless a good yardstick.
- Least Recently Used (LRU): throw out page not used for longest time in past. This is just an approximation of MIN, where we assume that past can be used to predict the future (a common theme in several optimizations). If past == future, then LRU=MIN. If past is somewhat equal to future, LRU has roughly the same performance as MIN

Implementing perfect LRU requires that each page is timestamped on every access. This timestamp needs to be saved on every access, and is thus on the hit path and has to be done in hardware. Saving the timestamp has both time and space overheads. Instead, the hardware implements a 1-bit approximation to a timestamp, namely an *accessed bit* in the page table entry. Thus, we approximate LRU using a 1-bit timestamp, namely the accessed-bit.

The accessed-bit in the pagetable entry is set by the hardware on a memory access. This bit distinguishes between pages that have been accessed recently and those that have not been accessed recently. accessed=1 implies "accessed recently", and accessed=0 implies "not accessed recently". The operating system periodically checks and clears the accessed bit, i.e., sets it to zero. If in a consecutive check, the OS finds that the accessed bit is set, it indicates that the page was accessed in the last periodic interval. Otherwise, it was not accessed. The OS only replaces a page that was not accessed. Also, among all the not-accessed pages, the pages are replaced in FIFO order (recall that FIFO orders the pages by their first access time, so it has some merit at-least, if not precise LRU). This 1-bit approximation to LRU is also called the CLOCK algorithm and is commonly used in operating systems.

One way to implement CLOCK is to arrange the pages in a circular list (clock) and have a pointer iterate over this circular list (clock hand). Each page has its accessed bit set to either zero or one. At replacement time, the clock hand looks at the page at which it is pointing, and uses the following logic:

```
if (page->accessed== 1) {
         page->accessed = 0;
   skip this page, i.e., advance the clock hand to next page
} else {
   evict this page, write to disk if needed
   read new page in this position, and set its accessed bit to zero
   advance the clock hand by one
}
```

Notice that because the clock hand is advanced by one after the new page is added, the newly added pages will be examined in FIFO order. If all pages have the same value for accessed bit, then this algorithm is equivalent to FIFO. On the other hand, if the page was accessed during the time period it takes to complete one revolution of the clock-hand, then it will not be replaced (i.e., it will be given priority over pages that were not accessed). However, its accessed bit will be set to zero for the next round.

What does it mean for the clock hand to move too fast? e.g., if it has to make one full revolution on every page fault. This means that all the pages are being accessed frequently, and this indicates that your physical memory is too small to fit the hot pages of the running program (also called its *working set*). This indicates that perhaps you are incurring many capacity misses, and thus need to buy more memory. On the other hand, if the clock hand moves too slow, it shows that most of the pages in physical memory are cold, i.e., they are not being accessed, and the working set of the program is much smaller than the size of the physical memory.

Two-hand clock: In the clock algorithm discussed so far, the notion of "recency" is defined by one clock revolution, i.e., if a page is accessed within one clock revolution, it is considered to be accessed "recently". If the size of the physical memory is too large, this interval of one clock revolution may become too coarse-grained. To deal with this, a variation is a clock with two hands, separated by a constant angle `theta`. The leading hand clears the accessed bit. The trailing hand looks for the victim page with accessed bit still cleared, i.e., it evicts the page if it finds that the accessed bit is still zero. Thus for a page to be considered recent, it needs to be accessed within the angle formed by the two hands.

If theta=0, it means that the accessed bit information is completely ignored, and it defaults to FIFO. If theta=360, it becomes identical to a single-hand clock.

An example of clock algorithm statistics recorded on real machines/OS:

```
bigmachine$ vmstat -s       # on SunOS
pages scanned by clock/second
- 200K pages examined
- 6 revolutions
- 120K pages freed

smallmachine$ vmstat -s
- 15K revolutions
```

The clock hand in the small machine is moving too fast, indicating the need to buy more memory.

Clock Extensions

- **Replace multiple pages at once**: expensive to run replacement algorithm and to write single block to disk. Find multiple victims each time.
- **Nth chance**: The clock algorithm is also called the "second-chance" algorithm, as the pages with the accessed bit equal to one, are given a second chance. A straight-forward extension to this is the Nth chance algorithm, wherein a page is given N chances before evicting it. Here is the Nth chance algorithm:
  - With each page, OS maintains a counter to indicate the number of sweeps that page has gone through.
  - On page fault, OS checks accessed bit:
    - If 1, then clear it, and also clear the counter.
    - If 0, then increment the counter; if count == N, replace page.

  Large N implies better approximation to LRU, e.g., N = 1000 is a very good LRU approximation. However, a large N implies more work by the OS before a page can be replaced.

  N = 1 implies the default clock algorithm.
- **Treat dirty pages preferentially**: Dirty pages require more work for eviction, and so given a choice, it is cheaper to replace clean pages. A common approach is to give dirty pages one more chance over clean pages, e.g., N=1 for clean pages, N=2 for dirty pages (and write back to disk in a batched manner when N=1).

LRU analysis

In general, LRU works well, but there are certain workloads where LRU performs very poorly. One well-known workload is a "scan" wherein a large memory region (larger than physical memory) is accessed in quick succession. e.g.,

```
for (i = 0; i < MAX_PHYSICAL_MEM_PAGES + 1; i++) {
  access(page i);
}
```

Consider an example where there are 5 pages in physical memory, and 6 pages of the address are repeatedly scanned: ABCDEFABCDEFABCDEFA...
What happens with LRU? All accesses are misses. In this case, the page that is replaced happens to be the one that is likeliest to be accessed closest in future. In other words, the future is opposite of the past. Any other algorithm (e.g., Random) would perform better than LRU in this case.

Such scans are often quite common, as some thread may want to go through all its data. And so, plain LRU (or CLOCK) does not work as well in practice. Instead a variant of LRU that considers both recency and frequency of access to a page, is typically used in an OS (e.g., 2Q, CLOCK with Adaptive Replacement, etc.).

A related idea called LRU-K tracks the K-th reference to a page in the past, and replaces the page with oldest K-th reference (or one that does not have K-th reference). This algorithm is resistant to scans, i.e., a scan will not pollute the cache with cold pages. This is expensive to implement for an OS, but LRU-2 is often used in databases.

Global or Local Replacement

Per-process or global replacement?

In global replacement, all pages from all processes are grouped into a single replacement pool. Each process competes with all the other processes for page frames.

In per-process replacement, each process has a separate pool of pages. A page fault in one process can only replace one of that process's frames. This avoids interference with other processes. But, how to decide how many page frames to allocate to each process? Bad decision implies inefficient memory usage.

One of two common approaches used in practice:

1. Use global replacement
2. Use per-process replacement but "slowly" change the number of frames allocated (quota) to each process depending on usage. The rate of change of the quota is much slower than the cache replacement rate, but adjusts for the case where one process is using less memory while the other process is running out of its allocated quota.

Thrashing and Working Sets

Normally, if a thread takes a page fault and must wait for the page to be read from disk, the OS runs a different thread while the I/O is occurring.

But what happens if memory gets overcommitted? Suppose the pages being actively used by multiple processes do not all fit in the physical memory. Each page fault causes one of the active pages to be moved to disk, so another page fault is expected soon. The system will spend most of its time reading and writing pages instead of executing useful instructions.

This is a situation where the demand paging illusion breaks down. The OS wanted to provide the size of disk and the access time of main memory. However, at this point, it is providing the access time of disk. This situation is called *thrashing*; it was a serious problem in early demand paging systems.

Dealing with thrashing:

- If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash.
- If the problem arises because of the sum of many processes:
  - Figure out how much memory each process needs.
  - Change scheduling priorities to run processes in groups that fit comfortably in memory: must shed load.

Working Sets

- Informally, a working set of a process is the collection of pages that a process is using actively, and which must thus be memory resident for the process to make progress (and avoid thrashing).
- If the sum of all working sets of all runnable threads exceeds the size of memory, then stop running some of the threads for a while.
- Divide processes into two groups: active and inactive:
  - When a process is active, its entire working set must always be in memory: never execute a thread whose working set is not resident.
  - When a process becomes inactive, its working set can migrate to disk
  - Threads from inactive processes are never scheduled for execution.
  - The collection of active processes is called the *balance set*.
  - Gradually move processes in and out of the balance set.
  - As working sets change, the balance set must be adjusted.

How to compute working sets?

- Denning proposed a parameter T; all pages referenced in the last T seconds comprise the working set. Note that recency of access (and not frequency of access) is used to determine the page's value (as in LRU).
- Computation of working set can be done by maintaining an *idle time* with each page.
- Clock algorithm extended to also increment the idle time, each time the hand passes through the page and the page is found "not-accessed". If the page is found accessed, the idle time is reset to zero.
- Pages with idle times less than T are in the working set.

Difficult questions for the working set approach:

- How long should T be? typically tens of seconds to minutes.
- Need to handle changes in working sets, and manage balance set.
- How to account for memory shared between processes? e.g., let processes that share memory be swapped in and out together.

Page Fault Frequency: another approach to prevent thrashing

- Use per-process replacement. Monitor the rate at which page faults occur in each process.
- If the rate gets too high for a process, assume that its memory is overcommitted; increase the size of its memory pool.
- If the rate gets too low, assume that its memory pool can be reduced in size.
- If the sum of all memory pools don't fit in memory, deactivate some processes.

Thrashing was more of a problem with shared computers (e.g., mainframes). With personal computers, users can either buy more memory or manage the balance set by hand (e.g., terminate some processes manually). Also, memory has become cheaper and plentiful, and so thrashing is less of a problem in general today.

Memory-mapped files

Modern operating systems allow file I/O to be performed using the VM system.

- Memory-mapped files (e.g., mmap())
- Programmer's perspective: file lives in memory, access through pointer dereferences
- Advantages: elegant, no overhead for system calls (can be especially significant for fast storage devices), use madvise() for prefetching limit.

Storage Devices

- Typical Characteristics of Magnetic Disk:
    - 1-5 platters, magnetically coated on each surface.
    - Actuator arm positions heads radially.
    - Roughly 200,000 tracks per radial inch (today's technology)
    - Overall disk package: 1-8 inches in height.
    - Typical capacities today: 100GB-2TB.
    - Typical rotational speeds: 5000-15000RPM. Larger disks tend to be slower.
    - Typical seek time: 2-10ms
    - Average rotational latency (for half a revolution): 4ms (@7500RPM)
    - Transfer: read or write data as it passes under the head. Typical transfer rates: 100-150 MBytes/second
    - Disk API:
        - read(startSector, sectorCount, buffer)
        - write(startSector, sectorCount, buffer)
    - Direct Memory Access (DMA): device can copy data to and from memory, without help from the CPU. This is the common case.
- Flash Memory:
    - Solid state (semiconductor) storage, but non-volatile
    - No moving parts, hence more shock-resistant
    - Faster access than disk
    - 5-10x more expensive than disk
    - Two methods of implementing: NAND and NOR. NAND more popular today.
        - Each device divided into blocks, which are subdivided into pages.
        - Typical block size: 16-256KB
        - Typical page size: 512-4096B. Page is the unit of reading/writing
        - Total capacity: up to 16GB per chip
        - Two significant quirks:
            - Before rewriting a page, its entire block must be *erased*; this is a separate operation. This makes random-access updates expensive.
            - Wear-out: once a block has been erased about 100,000 times, it no longer stores information reliably.
    - Ideal Usage of flash:
        - Write entire blocks (do not update individual pages). Otherwise, writes too slow, wears out blocks too quickly.
        - Wear leveling: manage device so all blocks get erased at roughly the same rate. Avoid hotspots.
- Older method: Magnetic tape
    - 9 tracks acrosss tape (one byte plus parity)
    - 0.5 inch wide by 2400 feet long
    - Typical bandwidth: few Mbytes/sec

Disk Scheduling

- If there are several disk I/Os waiting to be executed, what is the best order in which to execute them? The goal is to minimize seek time. Some options:
    - First come first served (FCFS, aka FIFO): simple, but nothing to optimize seeks.
    - Shortest seek time first (SSTF):
        - Choose next request that is as close as possible to the previous one.
        - Good for minimizing seeks, but can result in starvation of some requests.
    - Scan or elevator algorithm:
        - Choose a direction (inwards or outwards) based on FIFO order
        - Keep moving in the chosen direction, reading the blocks on the way, till you reach the last block. In other words, use SSTF for all blocks that are accessible in the chosen direction.
        - If no more blocks in the current chosen direction, change direction.
        - Similar to how an elevator works typically (replace inwards/outwards with up/down).

UNIX Filesystem Interface

We will focus on magnetic disks, given their popularity. The interface design and implementation heavily depends on the characteristics of the underlying storage device. Most OSes today, have been optimized for magnetic disks.

The API for a minimal file system consists of: open, read, write, seek, close, and stat. Dup duplicates a file descriptor. For example:

```
fd = open("x/y", O_RDWR);
read (fd, buf, 100);
write (fd, buf, 512);
close (fd)
```

Notice that this interface assumes that the file is a stream of bytes. Alternatively, the data in a file could have been organized in a structured way. The structured variant is often called a database. Any particular structure is likely to be useful to only a small class of applications, and other applications will have to work hard to fit their data into one of the pre-defined structures. Besides, if you want structure, you can easily write a user-mode library program that imposes that format on any file (with performance penalties however because of layering of different abstractions one above another). (Databases have special requirements and support an important class of applications, and thus have a specialized plan.)

To allow users to remember where they stored a file, they can assign a symbolic name to a file, which appears in a directory. There's a couple of different things going on here.

- Names: how do we get from the name "x/y" to the file we're actually talking about.
- Directories: provide a way of assigning user-meaningful names to files. This is a part of the kernel name interface.
- In Unix (and almost all other filesystems) there's some notion of an on-disk file that's independent of the file's name, called an "inode". Seen by the user (e.g. stat) but cannot access a given inode.
- Disk blocks: the inode translates offsets in our conceptual file into concrete locations on disk. Not seen by the user.
- File descriptors: in Unix, the FD binds to the inode, rather than the pathname, so the pathname lookup is done only when the file is first opened. Application handle for an inode in the Unix API.

Maintaining the file offset behind the read/write interface is an interesting design decision. The alternative is that the state of a read operation should be maintained by the process doing the reading (i.e., that the pointer should be passed as an argument to read). This argument is compelling in view of the UNIX fork() semantics, which clones a process which shares the file descriptors of its parent.

With offsets in the file descriptor, a read by the parent of a shared file descriptor (e.g., stdin) changes the read pointer seen by the child. This isn't always desirable: for example, consider a data file, in which the program seeks around and reads various records. If we fork(), the child and parent might interfere. On the other hand, the alternative (no offset in FD) would make it difficult to get "(echo one; echo two) > x" right. Easy to implement separate-offsets if kernel provides shared-offsets (re-open file, mostly), but not the other way around.

The file API turned out to be quite a useful abstraction. Unix uses it for many things that aren't just files on local disk, e.g. pipes, devices in /dev, network storage, etc. Plan9 took this further, and a few of those ideas came back to Linux, like the /proc filesystem.

Unix API doesn't specify that the effects of write are immediately on the disk before a write returns. It is up to the implementation of the file system within certain bounds. Choices include (that aren't non-exclusive):

- Before the write returns [most conservative, but bad for performance];
- Before close returns [AFS];
- At some point in the future, if the system stays up (e.g., after 30 seconds) [highest performance];
- Application specified (e.g., before fsync returns) [requires careful application coding];
- Before external things (screen, network) indicate the write returned [will read paper about this later].

Some examples of file systems:

- Contiguous Allocation: `file a = (base=10, len=12)`
  - Pro: Fast sequential access, easy random access
  - Con: External Fragmentation/hard to grow files
- Linked-list Allocation: `file = linked list of blocks`
  - Pro: Easy to grow file. Free list can be implemented as another file.
  - Con: Bad sequential access performance! Unreliable: Loose block, loose rest of file
  - Serious Con: Bad random access

File systems

Overview

- DOS FAT: Cute modification to linked list. Links reside in fixed size File Allocation Table (FAT). Still need to do pointer chasing but the entire FAT can fit in memory so chasing cheap.
    - Discussion (Entry size = 16 bits for FAT16, 28 bits for FAT32):
        - What's the maximum size of the FAT? (2^{16} for FAT16, 2^{28} for FAT32)
        - Each entry descibes a "block" that may be made up of multiple contiguous sectors.
        - Given a 2KByte block, what's the maximum size of FS? (2^{16}*2KB=128MB for FAT16, 2^{28}*2KB=512GB for FAT32)
        - Option: go to bigger blocks. FAT allows 2-32KB block sizes. Block size fixed for a disk partition at format time (called "Allocation Unit Size"). Pro? Bigger disk, better spatial locality exploitation. Con? Internal fragmentation, more time for data transfer than strictly needed, buffer cache pollution.
    - Space Overhead of FAT16: 4 bytes/2Kbyte block = approx .2%
    - Reliability: create duplicate copies of FAT to protect against errors
    - Bootstrapping: Where is root directory? Fixed location on disk / have a table in the bootsector (sector 0)
- Indexed Files: Each file has an array holding all it's block pointers. Fill in the block pointers as file grows/shrinks.
    - Pros: Fast random access, easy growth
    - Cons: Slow sequential access, growth beyond index size clumsy. Index size? (filesize/blocksize)*4 (assuming 4 bytes per block pointer).
- Multi-level Indexed Files (BSD/Unix/xv6): Direct blocks, Indirect blocks, Double-indirect blocks. Caters to common case of small files. Handles large files. Example: 4.3 BSD Unix
    - Inode contains 14 4-byte block pointers (initally 0, indicating no block)
    - First 12 point to data blocks
    - Next entry points to an indirect block containing 1024 4-byte block pointers
    - Last entry points to a doubly-indirect block.
    - Maximum file length is fixed, but large.
    - Indirect blocks are not allocated until needed.
    - Common case: small files require only one extra disk access for metadata lookup (inode lookup). Larger files require 2 or 3 metadata lookups.

A design issue is the semantics of a file system operation that requires multiple disk writes. In particular, what happens if the logical update requires writing multiple disks blocks and the power fails during the update? For example, to create a new file, requires allocating an inode (which requires updating the list of free inodes on disk), writing a directory entry to record the allocated i-node under the name of the new file (which may require allocating a new block and updating the directory inode). If the power fails during the operation, the list of free inodes and blocks may be inconsistent with the blocks and inodes in use. Again this is up to implementation of the file system to keep on disk data structures consistent:

- Don't worry about it much, but use a recovery program to bring file system back into a consistent state. Linux ext2 and FAT are such examples. xv6 is almost in this category, except that it has no recovery program.
- Journaling all file system state (kernel-managed structure as well as user-managed data). As we will discuss later, journaling provides atomicity over multiple disk operations. This tends to be quite expensive, in terms of performance. Linux's ext3 has a full data journaling mode that behaves in this fashion.
- Journaling only state that maintains kernel invariants. Never let the file system metadata get into an inconsistent state. In some sense, this resembles how the kernel deals with killing user processes: it cleans up kernel structures, so kernel invariants are upheld, but ignores application invariants. NTFS does this, as does Linux ext3 by default.

Another design issue is the semantics are of concurrent writes to the same data item. What is the order of two updates that happen at the same time? For example, two processes open the same file and write to it. Modern Unix operating systems allow the application to lock a file to get exclusive access. If file locking is not used and if the file descriptor is shared, then the bytes of the two writes will get into the file in some order (this happens often for log files). If the file descriptor is not shared, the end result is not defined. For example, one write may overwrite the other one (e.g., if they are writing to the same part of the file.) Lots of other examples with directories, names, etc.

An implementation issue is performance, because writing to magnetic disk is relatively expensive compared to computing. Three primary ways to improve performance are: careful file system layout and data structure design that induces few seeks (locality, btrees, logging, etc), an in-memory cache of frequently-accessed blocks (or even prefetching), and overlap I/O with computation so that file operations don't have to wait until their completion and so that that the disk driver has more data to write, which allows disk scheduling. (We will talk about performance in detail later.)

xv6 code examples

xv6 implements a minimal Unix file system interface. xv6 doesn't pay attention to file system layout. It overlaps computation and I/O, but doesn't do any disk scheduling. Its cache is write-through, which simplifies keeping on disk data structures consistent, but is bad for performance.

What is xv6's disk layout? Who determines how many inodes, blocks, etc. (mkfs.c). How does xv6 keep track of free blocks and inodes? See balloc()/bfree() and ialloc()/ifree(). Is this layout a good one for performance? What are other options?

- Block 0 is unused
- Block 1 is super block
- Inodes start at block 2
- Free-block bitmap at block `ninodes/IPB + 3`
- Free blocks after that

Free blocks are tracked using bitmap. Free inodes are tracked by linearly scanning the inode array to find one that is free. Other options (bitmap for inodes, better locality among inodes and blocks)

On disk, files are represented by an inode (struct dinode in fs.h), and blocks. Small files have up to 12 block addresses in their inode; large files use the last address in the inode as a disk address for a block with 128 disk addresses (512/4). The size of a file is thus limited to 12 * 512 + 128*512 bytes. What would you change to support larger files? (Ans: e.g., double indirect blocks.)

Directories are files with a bit of structure to them. The file contains of records of the type struct dirent. The entry contains the name for a file (or directory) and its corresponding inode number.

Directory contents: A directory is much like a file, but user can't directly write or read from it. It can only do so using system calls like `readdir()`. Also, the kernel reads the contents to do path resolution (e.g., converting "x/y" to inode number). The content of a "directory file" is an array of dirents (notice that the contents are structured this time).
dirent:

- inum
- 14-byte file name

dirent is free if inum is zero
How many files can appear in a directory? (max file size / sizeof(struct dirent))

xv6's on-disk inode (64 bytes):

- type (free, file, directory, device)
- nlink

- size
- addrs[12+1]
  direct and indirect blocks

Each i-node has an i-number. To turn i-number into inode's block, use 2 + inum/IPB. (IPB = inodes per block). Blocksize=512 inodesize=64, IPB=8

Can view FS as an on-disk data structure. Draw tree: dirs, inodes, blocks. There are two allocation pools: inodes and blocks.

Q: how does xv6 create a file?:

```
$ echo > a
log_write 4 ialloc (44, from create 54)
# allocating an inode for a

log_write 4 iupdate (44, from create 54)
# updating the inode of a

log_write 29 writei (47, from dirlink 48, from create 54)
# adding entry for a in the contents to the current working directory

log_write 2 iupdate
# updating the inode of the current working directory.
```

Filesystem layout

Draw diagram of disk:

- Boot sector (sector 0)
- Superblock (sector 1)
- Inodes (sector 2 to ninodes/IPB+3)
- Free-block bitmap (one bit per data block indicating free/used)
- Blocks

File Creation

```
$ echo > a

1. allocate an inode (write to inode region).
2. update the size/status of the inode (write to inode region).
3. add a directory-entry record to the parent directory's data blocks. (write
to data block region)
4. update parent directory's inode (size)
```

File Write/Append

```
$ echo x > a

1. allocate a block (write to block bitmap region)
2. zero out block contents (write to data block region)
3. write "x" to block (write to data block region)
4. update "a"'s inode (size, addrs)
```

File Unlink

```
$ rm a

1. write zero to "a"'s directory-entry record in parent directory's
        data blocks (write to data block region)
2. update parent directory's inode (size)
3. update "a"'s inode to mark it free
4. update free block bitmap to mark data blocks free
```

Notice that each operation involves 4-8 disk writes and these filesystem accesses are to different regions of disk, i.e., they involve multiple seeks/rotations. Notice that we are assuming a write-through cache. Also, the number of outstanding I/O's (in-flight I/O's for the disk controller) is relatively small (one?) so this is not very efficient usage of the disk.

Synchronization for FS Accesses

What happens if concurrent accesses to the filesystem are performed by multiple threads? For example, bad things can happen if two threads try to create files within the same directory concurrently.

Solution 1: Use a coarse-grained lock for the entire filesystem.
This is bad because it serializes concurrent accesses to different files. Given that file accesses are anyways very slow, this can be a huge performance problem. Moreover, it restricts the number of outstanding I/Os to a very small number (typically, one), as only one thread could be executing within the filesystem at any time.

Question: does the coarse-grained global filesystem lock need to be stored on-disk or does an in-memory lock suffice? Answer: In-memory lock suffices, as all threads need to access the disk through the OS interfaces which can synchronize using the in-memory lock.

Solution 2: Use fine-grained locks --- in particular, use a lock per block. Use the buffer cache to synchronize. Mandate that a block can only be accessed by first bringing it into the buffer cache. Then use a lock per buffer in the buffer cache. (recall that a buffer corresponds to a disk block). The locks need to be blocking locks, as critical sections are likely to be very large (disk accesses).

xv6 implements per-buffer locks by using a BUSY bit per buffer. Accesses to the BUSY bit are protected by a global buffer cache spinlock, `bcache.lock`. Thus, to access a block:

1. First the global `bcache.lock` is acquired.
2. The block being accessed is searched in the buffer cache.
3. If found and !BUSY,
    - Mark it busy
    - Release `bcache.lock`
    - Return buffer contents
4. If found and BUSY,
    - `sleep(buffer address, &bcache.lock)`. This will release `bcache.lock`. A thread that is currently working on this buffer (and has thus marked it BUSY) should call wakeup after clearing the BUSY bit.
    - On waking up, the thread should rescan/re-search the buffer cache for the desired block at step 2 (as it may have been replaced after the `bcache.lock` was released).
5. If not found
    - Find a replacement buffer that is not currently BUSY (this ensures that a busy buffer cannot be evicted).
    - Mark it busy
    - Release `bcache.lock`. (notice that I now hold a fine-grained lock on the desired buffer, and so do not need `bcache.lock`)
    - Write replacement buffer to disk if dirty
    - Read desired block into this buffer from disk
    - Return buffer contents

   Notice that all disk operations are done without holding `bcache.lock`. Instead disk operations are protected per-buffer locks (BUSY bits).

Notice that the global buffer cache lock is used only to manipulate the busy bits of the individual buffers. The busy bits act as fine-grained locks.

But doesn't the busy bit protect only against concurrent accesses to a single block. What about operations that require accesses to multiple blocks. How are they made atomic?

Simple: mark all the blocks (on which an atomic operation needs to be performed) busy. (Just like taking multiple locks). But . . . we know that fine-grained locks have deadlocks. So, need an order on the setting of busy bits for buffers. Here are some global invariants followed by xv6 (check on your own):

- Always mark the superblock busy first (if it needs to be involved in the atomic operation).
- Always mark inode blocks before data blocks
- Always mark parent (dir)'s inode before child (dir)'s inode
- Never mark any other block if one of the bitmap blocks is marked

xv6 implements LRU for buffer cache replacement.

- Maintain the buffers in a doubly-linked list.
- Each time you are done with accessing a buffer (at the time of clearing the busy bit), move the buffer to the front of the buffer cache list (see `brelse`).
- Start replacement at the last entry of the list.

Notice that because xv6 uses a write-through buffer cache, many consistency problems become easy. We will next look at the consistency issues that can arise due to a write-back buffer cache.

Recall that concurrent accesses to the disk are synchronized by `idelock`. Also, recall that the xv6 IDE device driver allows only one outstanding disk request at any time (the front of the idequeue). Clearly, this can be improved so that there are multiple outstanding disk requests and the disk device can schedule them efficiently (recall elevator algorithm).

How fast can an xv6 application read big files? If the file has contiguous blocks? xv6 has no prefetching, it will wait for the first block to return before it issues command for second block. In doing so, we waste a full rotation of the disk! A better approach would be to prefetch, or to allow multiple outstanding requests to the disk device.

Q: Why does it make sense to have a double copy of I/O? First we read data from disk to buffer cache. Then, from buffer cache to user space. Can we do better? e.g., pass user-space buffer to the disk device driver? Few issues:

- Need to lock that page in memory (can't be swapped out)
- What if the user process accesses it in the middle. May be okay, actually.
- But, the buffer will not be available to other processes -- no caching! may be okay for large scans, but not okay for workloads exhibiting locality across processes.

Q: how much RAM should we dedicate to disk buffers? Tradeoff: if too big buffer cache, less space for virtual memory pages; and vice-versa. Can be adapted at runtime based on usage patterns.

Crash Recovery

A filesystem operation requires multiple disk accesses. What happens if power fails in the middle of a filesystem operation. It can leave the filesystem in an inconsistent state.

Example: file creation involves two main steps (in terms of disk writes)

1. Initialization/allocation of inode
2. Creating entry for it in the parent directory

If at power failure time, the entry for the new inode has been created in the parent directory, but the inode has not been allocated, then we have a case of a *dangling pointer* in our filesystem tree. This can be a major problem, as the uninitialized inode may contain junk data (or worse private data of another user) which may confuse future accesses to this part of the filesystem.

On the other hand, if at power failure time, the opposite is true, i.e., the inode has been allocated/initialized but its entry has not been created in the parent directory, then there is no problem of a dangling pointer in the filesystem tree. In this case, we have a *space leak* because an inode has been marked allocated but it is not pointed-to by the filesystem tree, and so it will never be used.

A space-leak is more acceptable than a dangling pointer. Using this observation, a filesystem can enforce an order on the disk writes. The order should ensure that there will never be any dangling pointers in the filesystem tree. In this file creation example, this means that an inode should be initialized before an entry is created for it in its parent directory. Similarly at file unlink time, the entry must be removed from the parent directory before deallocating the inode (if done in the opposite order, dangling pointers can result).

Crash Recovery

Ordering Disk Writes

Order disk writes to avoid dangling pointers on crash. Allow disk block leaks. On reboot after crash, optionally perform a global filesystem check (fsck) to identify inconsistencies (like space leaks).

- e.g., create/append: initialize child block/file before adding pointer inside parent inode/dir
- e.g., unlink/truncate: remove parent inode/dir's pointer before deallocating child block/file

The fsck program can assume that inconsistencies will be limited to certain types (e.g., space leaks are possible but dangling pointers are not). If it finds an inconsistency (e.g., space leak), it will fix it. But, can we always avoid filesystem inconsistencies? Consider the following example:

```
$ mv a/foo b/foo
```

In this case, the pointer to `foo`'s inode needs to be removed from directory `a`, and added to directory `b`, involving two disk block writes. If we remove the pointer from directory `a` first, we risk losing the file `foo` on a crash. On the other hand, if we add the pointer to directory `b` first, we risk an inconsistency, where the same file `foo` appears in two directories (this may not be allowed depending on the filesystem semantics).

In this case, on a reboot after crash, the `fsck` program will notice the inconsistency, but will not know how to resolve it, i.e., should it remove the pointer from directory `a` or from directory `b`? At these points, the `fsck` program can ask the user to resolve this inconsistency by suggesting a choice.

Let's see how long does `fsck` take. A random read on disk takes about 10 milliseconds. Descending the directory hierarchy might involve a random read per inode. So maybe (n-inodes / 100) seconds? This can be made faster if all inodes (and dir blocks) are read sequentially (in one go), and then descend hierarchy in memory.

fsck takes around 10 minutes per 70GB disk w/ 2 million inodes. Clearly this performance is possible only by reading many inodes sequentially. The time taken by fsck is roughly linear in the size of the disk. As disks grew in size, the cost of running fsck started becoming impractical. We will next study a more modern method of ensuring crash recovery, called logging.

But before that, ordering of disk writes, as we have discussed it so far, would involve synchronous writes to disk. A write-back cache could potentially reorder the disk writes (i.e., writes to buffers in the cache may be in a different order from the actual writes to the disk). This may not be acceptable from a crash-recovery standpoint. On the other hand, a write-through buffer cache is unacceptable from a performance standpoint. (Recall that creating a file and writing a few bytes takes 8 writes, probably 80 ms, so can create only about a dozen small files per second. think about un-tar or rm *)

One solution is to maintain a dependency graph in memory for the buffer cache. The dependency graph indicates an ordering dependency between flushes of different blocks to disk. For example, for the command, `mv a/foo b/foo`, a dependency edge should be drawn from directory `b`'s block to directory `a`'s block, indicating that `b` should be flushed before `a`.

However, consider the following example, where the dependency graph can contain a cycle:

```
$ mv a/foo b/foo
$ mv b/bar a/bar
```

The first command draws a dependency edge from `b` to `a`, while the second command draws a dependency edge from `a` to `b`. At this point, no matter which block is flushed first, there is a chance that one file may get lost (either foo or bar). The solution to this problem is to identify potential cycles in the dependency graph (at the time of making in-memory updates), and avoid them by flushing existing dirty buffers to disk. In this example, before the second command is executed, the OS should flush the blocks for directories `a` and `b` (thus eliminating the dependency edge between them) so that the addition of a new dependency edge does not cause a cycle.

Logging

Goals:

- Atomic system calls w.r.t. crashes
- Fast recovery (no hour-long fsck)
- Speed of write-back cache for normal operations

The basic idea behind logging:

- You want atomicity: all of a system call's writes, or none. Let's call an atomic operation a "transaction".
- Record all writes the sys call *will* do in the log
- then record "done"
- then do the writes
- on crash+recovery:
    - if "done" in log, replay all writes in log
    - if no "done", ignore log

  this is a WRITE-AHEAD LOG

Simple logging (as in xv6):
[diagram: buffer cache, FS tree on disk, log on disk]

- FS has a log on disk
- syscall:

```
begin_trans()
  bp = bread()
  bp->data[] = ...
  log_write(bp)
  more writes ...
commit_trans()
```

- begin_trans:
    - need to indicate which group of writes must be atomic!
    - lock -- xv6 allows only one transaction at a time
- log_write:
    - record sector #
    - append buffer content to log
    - leave modified block in buffer cache (but do not write)
- commit_trans():
    - record "done" and sector #s in log
    - do the writes
    - erase "done" from log
- recovery:
  if log says "done":
    - copy blocks from log to real locations on disk

What's wrong with xv6's logging?

- only one transaction at a time
    - two system calls might be modifying different parts of the FS
- log traffic will be huge: every operation is many records
- logs whole blocks even if only a few bytes written
- eager write to log -- slow. As discussed, the log blocks and the commit record needs to be written at `commit_trans()` time. This allows only 4-5 disk writes to get batched at a time. It would be much better if hundreds of disk writes get batched together.
- eager write to real location -- slow. This can be easily fixed by asynchronously applying the logged writes to the FS tree.

- Every block written twice

Fixing xv6 Logging (as done in Linux ext3):

- Basic idea: Batch many (atomic) disk operations into a single transaction. Each operation can involve multiple disk writes. Replace `begin_trans()` and `commit_trans()` with `begin_atomic_op()` and `commit_atomic_op()`. The commit of an atomic operation does not cause a commit of the whole transaction.
- Have one transaction *open* at any time.
- Add all new disk operations to the currently open transaction.
- *Close* the transaction at periodic intervals (e.g., 30 seconds).

Closing a transaction:

- Open a new transaction: all future disk operations that start will get added to the new transaction.
- Mark this transaction as closed. No new disk operations will be admitted to it.
- Wait for ongoing disk operations to complete, i.e., wait for them to call `commit_atomic_op()`.
- Write descriptor and data blocks to the log
- Write the commit record
- Allow the blocks in log to be applied to the FS tree (but not forced). Typically, done asynchronously with many in-flight I/Os to take best advantage of the disk scheduler.

In doing this, we alleviate many of the problems with xv6's logging:

- Many disk operations can simultaneously execute
- Writing the log will involve one (or two) large sequential writes, every few seconds. So log traffic is significantly reduced.
- Even though whole blocks are logged, multiple writes to the same block are absorbed in the buffer cache. For example, only the final value of the block (at the end of the transaction) needs to be logged to the disk.
- The write to the log is still eager, but only at a coarse periodic interval (e.g., few seconds). Also, the write can be completed in one sequential write benefitting from batching many different disk operations.
- The logged blocks are applied to the FS tree asynchronously and lazily, so the disk can efficient schedule these writes for maximum disk throughput.
- Even though every block is still written twice, we no longer pay the price of a disk seek and rotational latency for each write. These writes are fast as they are either a part of a large sequential write (in case of log) or can be efficiently scheduled with many other in-flight I/Os (in case of asynchronous writes to the FS tree).

Logging

ext3 structures:

- in-memory write-back block cache
- in-memory list of blocks to be logged, per-transaction
- on-disk FS
- on-disk circular log file

what's in the ext3 log?

- superblock: starting offset and starting seq #
- descriptor blocks: magic, seq, block #s
- data blocks (as described by descriptor)
- commit blocks: magic, seq

sys call:

- h = start()
- get(h, block #)
    - warn logging system we'll modify cached block.
      added to list of blocks to be logged
    - prevent writing block to disk until after xaction commits
- modify the blocks in the cache
- stop(h)
- guarantee: all or none
- stop() does *not* cause a commit
- notice that it's pretty easy to add log calls to existing code

is log correct if concurrent syscalls?

- e.g. create of "a" and "b" in same directory
- inode lock prevents race when updating directory
- other stuff can be truly concurrent (touches different blocks in cache)
- transaction combines updates of both system calls

what if a crash?

- crash may interrupt writing last xaction to log on disk
- so disk may have a bunch of full xactions, then maybe one partial
- may also have written some of block cache to disk
    - but only for fully committed xactions, not partial last one

how does recovery work?

1. find the start and end of the log
    - log "superblock" at start of log file
    - log superblock has start offset and seq# of first transaction. found start.
    - scan until bad record or not the expected seq #
    - go back to last commit record. found end.
2. Replay all blocks through last complete xaction, in log order

what if block after last valid log block looks like a log descriptor?

- perhaps left over from previous use of log? It will have the wrong sequence number
- perhaps some file data happens to look like a descriptor? It will not have the magic number.

when can ext3 free a transaction's log space?

- after cached blocks have been written to FS on disk
- free == advance log superblock's start pointer/seq

what if not enough free space in log for a syscall?

- suppose we start adding syscall's blocks to T2
- half way through, realize T2 won't fit on disk
- we cannot commit T2, since syscall not done
- but can free T1 to free up log space.
- Log space should be bigger than expected size of a single transaction

ext3 not as immediately durable as xv6

- creat() returns -> maybe data is not on disk! crash will undo it.
- need fsync(fd) to force commit of current transaction, and wait
- would ext3 have good performance if commit after every sys call?
    - would log many more blocks, no absorption
    - 10 ms per syscall, rather than 0 ms

what if syscall B reads uncommited result of syscall A?

```
A: echo hi > x &
B: ls > y &
```

could B commit before A, so that crash would reveal anomaly?

- case 1: both in same xaction -- ok, both or neither
- case 2: A in T1, B in T2 -- ok, A will commit first (assuming T1 < T2)
- case 3: B in T1, A in T2
    - could B see A's modification?
        - This is possible if B called `start()` before A, but called `stop()` after A. In this case, B could see A's modification, and yet be in an earlier transaction.
    - Solution: ext3 must wait for all ops in prev xaction to finish before letting any in next start, so that ops in old xaction don't read modifications of next xaction. Notice that ops in new xaction can start even if the old xaction has not yet committed.

T2 starts while T1 is committing to log on disk

- what if syscall in T2 wants to write block in prev xaction?
- can't be allowed to write buffer that T1 is writing to disk
    - then new syscall's write would be part of T1
    - crash after T1 commit, before T2, would expose update
- T2 gets a separate copy of the block to modify
    - T1 holds onto old copy to write to log

performance?
create 100 small files in a directory

- would take xv6 over 10 seconds (many disk writes per syscall)
- repeated mods to same direntry, inode, bitmap blocks in cache
    - write absorbtion...
- then one commit of a few metadata blocks plus 100 file blocks
- how long to do a commit?
    - seq write of 100*4096 at 50 MB/sec: 10 ms
    - wait for disk to say writes are on disk
    - then write the commit record
    - that wastes one revolution, another 10 ms
    - modern disk interfaces can avoid wasted revolution