YehudaShapira /
xv6-explained

Code Issues Pull requests Actions Projects Security Insigh

master xv6-explained / **xv6 Code Explained.md**

YehudaShapira Fixed markup 7 years ago

1513 lines (790 loc) · 36.6 KB

Preview Code Blame Raw

Hello, world.

In this document, we'll attempt to explain the actual code of our beloved xv6.
Not all of it, but some of the interesting parts.

God have mercy on us.

1217 main(void)

The entry point of the kernel.
Sets up kernel stuff and starts running the first process.

1219: set up first bunch of pages, for kernel to work with (minimal, because old harware has little memory)

1220: set up all kernel pages

1223: set up segment tables and per-CPU data

1238: set up the rest of the pages for general use (because until now we had just minimal, because other CPUs might not handle high addresses)

1239: set up the First Process

1241: run the scheduler (and the First Process)

2764 struct run

Represents a memory page.
A `run` points to the next available page/ `run` (which is actually the *previous* page, because the first available is the last in the memory).

`2768 struct kmem`

Points to the head of a list of free (that is, available) pages of memory.

`2780 kinit1(void *vstart, void *vend)`

Frees a bunch of pages.
Also does some locking thing (I'll elaborate once we actually learn this stuff).
Used only when kernel starts up.

Called by `main` .

`2788 kinit2(void *vstart, void *vend)`

Frees a bunch of pages.
Also does some locking thing (I'll elaborate once we actually learn this stuff).
Used only when kernel starts up.

Called by `main` .

`2801 freerange(void *vstart, void *vend)`

Frees a bunch of pages.

2804: use PGROUNDUP because `kinit1` in called to start where the kernel finished (which is not likely to end *exactly* at a page end).

Called by:

- `kinit1`
- `kinit2`

`2815 kfree(char *v)`

Frees the (single!) page that `v` points at.

2819-2820: address validity checks

2823: fill page with 1s, to help in case of bugs

2827-2829: insert our page into the beginning of `kmem` (a linked list with all available pages)

Called by:

- `deallocuvm`
- `freevm`
- `fork`
- `wait`
- `freerange`
- `pipealloc`
- `pipeclose`

`2838 kalloc(void)`

Removes a page from `kmem` , and returns its (virtual!) address.

2844-2846: remove first free page from `kmem`

2849: return address

Called by:

- `startothers`
- `walkpgdir`
- `setupkvm`
- `inituvm`
- `allocuvm`
- `copyuvm`
- `allocproc`
- `pipealloc`

1757 `kvmalloc(void)`

Builds new page table and makes `%CR3` point to it.

1759: make table and get its address

1760: make `%CR3` point to returned address

Called by `main` .

1728 `kmap[]`

Contains data of how kernel pages should look.
Used by `setupkvm` for mapping.

Column 0: Virtual addresses
Column 1: Physical addresses start
Column 2: Physical addresses end
Column 3: Pages permissions

Note: The `data` variable (used in lines 1730-1731 is where the kernel's data start (*data* is plural, by the way).
We do not know during compliation where this will be.

1737 `setupkvm(void)`

Sets up kernel virtual pages.
Returns page table address if successful, 0 if not.

1742: create outer `pgdir` page table

1744: clear `pgdir`

1745-1746: make sure we didn't map illegal address

1747-1750: loop over `kmap` and map pages using `mappages`

Called by:

- `kvmalloc`
- `copyuvm`
- `userinit`
- `exec`

1679 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)

Creates translations from *va* (virtual address) to *pa* (physical address) in existing page table `pgdir`.
Returns 0 if successful, -1 if not.

1684: get starting address

1685: get ending address (which is starting address if `size`=1)

1686: for each page...

1687: get i1 row address (using `walkpgdir`)

1689: make sure i1 row not used already

1691: write *pa* in i1 and mark as valid, with required permissions

Called by:

- [setupkvm]
- [inituvm]
- `allocuvm`
- `copyuvm`

1654 walkpgdir(pde_t *pgdir, const void *va, int alloc)

Looks at virtual address *va*,
finds where where it should be mapped to according to page table `pgdir`,
and returns the **virtual** address of the the *index* i1.
Returns address if successful, 0 if not.

If there is no mapping, then:
if `alloc`=1, mapping is created (and address is returned);
if `alloc`=0, return 0

Some constants and macros used here:
PDX - zeroes offset bits
PTX - uh... some more bit manipluations
PTE_P - "valid" bit
PTE_W - "can write" bit
PTE_U - "available in usermode" bit

1659: get i0 index address

1660: check if i0 is valid

1661: put address of subtable in `pgtab`

1663: (i0 not valid) create new subtable, `pgtab`

1666: (i0 not valid) clear `pgtab` rows

1670: (i0 not valid) point i0 to `pgtab` and mark **i0** as valid, writable, usermodeable.

1672: return address of appropriate row in `pgtab`

Called by:

- `mappages`
- `loaduvm`
- `deallocuvm`
- `clearpteu`
- `copyuvm`
- `uva2ka`

`1616 seginit(void)`

Sets segmentation table so that it doesn't get in the way, for each CPU. Adds extra row to each segementation table in order to guard CPU-specific data, makes `%gs` resgister point to it, and makes `proc` and `cpu` actually point to `%gs` .

1624-1628: set up regular rows in segmentation table

1631-1634: set up special row and `%gs` register

1637-1638: set up inital `proc` and `cpu` data

Called by `main` .

2252 `userinit(void)`

Creates and sets up The First Process.

2255: data and size of First Process code. Filled by script during compilation.

2257: allocate `proc` structure and set up data on kernel stack

2258: save proc in `initproc`, so we'll always remember who the First Process is

2259-2260: create page table with kernel addresses mapped

2261: allocate free page, copy process code to page, map user addresses in page table

2262-2275: fix data on kernel-stack, *as if* it were stored there because of an interrupt

- **2264**: make sure we'll be in usermode when process starts
- **2270**: make sure process will start in address 0

Called by `main`.

2205 `allocproc(void)`

Allocates `proc` structure and sets up data on kernel stack.
Returns proc if succeeds, 0 if not.

2211-2213: find first unused `proc` structure

2217-2219: set to EMBRYO state and give `pid`

2223-2226: allocate and assign kernel stack for process

2227: set stack pointer to bottom of stack (stack bottom is highest address in stack)

2230-2231: make room on stack for `trapframe`

2235-2239: make room on stack for `context`

2240-2241: set `context`, setting `context.eip` to function `forkret`

Called by:

- `userinit`
- `fork`

`1803 inituvm(pde_t *pgdir, char *init, uint sz)`

Allocates and maps single page (4KB), and fills it with with program code.

1807-1808: make sure entire program code fits in single page

1809: allocate page

1810: clear page

1811: map pages in page table `pgdir` (using `v2p`, because we know what memory this is, because this is the First Process, which the kernel always creates on startup)

1812: copy the code

Called by `userinit`.

`2458 scheduler(void)`

Loops over all processes (in each CPU), finds a runnable process, and runs it.
Loops for ever and ever.

2467: lock process table, to prevent multiple CPUs from grabbing same process

2468-2470: find first available process

2475: set *per-CPU* variable `proc` to point to current running process

2476: set up process's kernel stack, and switch to its page table

2477: mark process as running

2478: save current registers (including where to continue on scheduler) and load process's registers, handing the stage over to the process
(NOTE: the running process is responsible to release the process table lock (to enable interrupts) and later re-lock it.)

2479: now that process is switching back to scheduler, switch back to kernel registers and Page Table

2483: set per-CPU variable `proc` back to 0

2485: release process table, allowing other CPUs to grab processes (just in case

Called by `mpmain`.

`1773 switchuvm(struct proc *p)`

Perpares kernel-stack of process (that is, makes `%tr` register indirectly point to it), and loads process's Page Table to `%cr3` .

1776-1779: set up `%tr` register and `SEG_TSS` section in GDT end up magically (don't ask how) referring us to top of process's kernel stack

Called by:

- `growproc`
- `scheduler`
- `exec`

`2708 swtch(struct context **old, struct context *new)`

Saves current register context in `old` , then loads the register context from `new` . Basically gives control to new process.

2709: set `%eax` to contain address of `old` context

2710: set `%edx` to contain `new` context

2713-2716: push `%ebp` , `%ebx` , `%esi` , `%edi` onto current stack (which happens to be `old` stack)

2719: copy value of `%esp` to address held in `%eax` , which is the `old` stack address (see line **2709**)

2720: set current stack pointer (`%esp`) to value of `%edx` , which is the `new` stack address (see line **2710**)

2723-2726: pop `%edi` , `%esi` , `%ebx` , `%ebp` from `new` stack onto the actual stack

Called by:

- `scheduler`
- `sched`

`2503 sched(void)`

Switches back `scheduler` to return from a process that had enough running.

Called by:

- `exit`
- `yield`
- `sleep`

`2522 yield(void)`

Gives up the CPU from a running process.

2524: re-lock the process table for scheduler

2525: make self as not running

2526: switch back to scheduler

2527: after scheduler re-ran process, re-ealease process table to enable interrupts

Called by `trap`.

`2553 sleep(void *chan, struct spinlock *lk)`

Makes process sleep until `chan` event occurs.

2568: lock process table in order to set sleeping state safely

2569: now that process table is locked, release `lk`

2573-2574: set up sleeping state (and alarm clock)

2575: return to scheduler until the event manager marks process as runnable

2578: clean up

2582: release process table

2583: lock `lk` once again

`1555 pushcli(void)`

Saves state of `%eflags` register's `IF` bit (that is, the current state of "listen to interrupts?" bit),
increments the "how many times did we choose to ignore interrupts" counter,
and clears the "listen to interrups" bit.

1561-1562: save initial state of bit in `interna` var (FL_IF is the location of our bit)

Called by:

- `acquire`
- `switchuvm`

`1566 popcli(void)`

Decrements the "how many times did we choose to ignore interrupts" counter,
and if it reaches 0 then sets the "listen to interrups" bit to whatever it was before the very first `pushcli` was ever called.

1572-1573: only set our bit if `interna` (initial bit value) was set

Called by:

- `release`
- `switchuvm`

`1474 acquire(struct spinlock *lk)`

Loops over spinlock until lock is acquired (exclusively by current CPU).

1476: disable interrupts (which will be enabled in `release`)

1483-1484: loop over lock until it has a zero (and write "1" in it)

`1502 release(struct spinlock *lk)`

Releases spinlock from being held by current CPU.

1519: write "0" in lock

1521: re-enable interrups (which were disabled in `acquire`)

`3004 alltraps`

Catches and prepares all interrupts for `trap`.
Pushes register data on stack, calls `trap` with the `stack` as a `trapframe` argument, pops register data from stack, and finally calls `iret`.

3005-3010: store registers and build `trapframe`

3013-3018: set up data and per-CPU segments (?)

3021-3023: call `trap`, using stack as argument

- **3023**: skip over top of frame (`%esp` address) without popping it into anything

3027-3034: pop registers and call `iret`

- **3033**: skip over data and per-CPU segments (without popping it into anything)

`3101 trap(struct trapframe *tf)`

Handles all interrups.

3103-3111: handle system call and return

- **3106**: save `tf` to `proc->tf`, so that we don't need to start passing it around during `syscall`

3113-3143: handle controller interrupts (keyboard, timer, etc.)

3150-3163: handle unexpected interrupt

- **3151**: check if there is no current process (i.e. during `scheduler`) or if we were in kernel-mode during interrupt
- **3158-3162**: print error and kill buggy process

3168-3178: finish up non-system calls

- **3168-3169**: if process is user-process, and killed, and is not in the middle of a system-call, exit (and don't return to `alltraps`)
- **3173-3174**: if process is running, and we had a timer-interrupt, and the process ran for long enough already, yield CPU back to `scheduler`
- **3177-3178**: after previous yield, if process was killed (and not in middle of system-call), exit

Called by `alltraps`

`3067 tvinit(void)`

Initializes the IDT table.

Called by `main` .

`3079 idtinit(void)`

Makes `%IDTR` point at existing IDT table.

Called by `mpmain`

`3375 syscall(void)`

Handles system-calls from user-code.

3379: get system-call number

3380: make sure number is valid

3381: execute system-call and store return value in process's trapframe's `eax` field (which will afterward be popped to `%eax`)

3382-3385: if bad system-call number, print error and store -1 (error) in trapframe's `eax` field

Called by `trap` .

`3465 sys_sleep(void)`

System call for sleeping a certain amount of ticks.

3470: get number of required ticks (and validate that the user supplied a valid address as an argument)

3472: lock tickslock

3473: store current (initial) value of `ticks` in `ticks0`

3474-3480: while required number of ticks didn't pass, loop

- **3479**: wait for event # `&ticks` (we don't really need the lock in this case, but usually in `sleep` call we need to lock because other cases we `sleep` for disc or something else where we don't want two process's to grab the resource simultaneously)

Can be called by user code.

`2614 wakeup(void *chan)`

Locks process table, finds all sleeping processes that are waiting for `chan`, makes them runnable, and unlocks process table.

Called by a lot of different functions.

`2603 wakeup1(void *chan)`

Finds all sleeping processes that are waiting for `chan`, and makes them runnable.

Called by:

- `exit`
- `wakeup`

`3295 argint(int n, int *ip)`

Gets the `n` th *integer* argument pushed onto the user-stack by user code before user asked for system-call.

`3267 fetchint(uint addr, int *ip)`

Gets the integer argument in address `addr`, and sets it in `ip`.
Returns 0 if successful, -1 otherwise.

3267: Validates that neither "edge" of integer-containing address space goes beyond valid proc memory.

Called by:

- `argint`
- `sys_exec`

`2304 fork(void)`

Creates new process, copying lots from its parent, and set stack as if returning from a system-call.

2310: allocate new proc. Proc now contains kernel-stack, context (with trapret address), trapframe and pid

2314-2319: copy memory

- **2314**: copy memory
- **2315-2318**: if error, free kernel stack

2320: copy `sz`

2321: set parent

2322: *copy* trapframe struct to new kernel-stack

2325: clear `%eax` so `fork` will return 0 for child process

2327-2330: do file stuff

2333: make new proc RUNNABLE (at this point, `scheduler` can grab child process before parent)

`1953 copyuvm(pde_t *pgdir, uint sz)`

Creates copy of parent memory for child process.
Returns address of new page table.

1960: set up kernel virtual pages

1962: loop over all pages:

- **1963**: get address+flags of parent process's page
- **1965**: make sure page is actually present
- **1967**: get physical address of parent process's page
- **1968**: allocate new page
- **1970**: copy memory from old physial page to new physical page
- **1971**: add-n-map new page to new page table

1974: if no errors, return address of new page table

1977-1978: if there were any errors, release all memory and return 0

Called by `fork`

`1910 freevm(pde_t *pgdir)`

Frees a page table and all the physical memory pages (in its user part).

1916: free user-mode pages

1917-1921: free internal page tables

1923: free external page table

Called by:

- `copyuvm`
- `wait`
- `exec`

`1882 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)`

Deallocates user pages to bring the process size from `oldsz` to `newsz`.

1891: loop over extra pages we want to deallocate:

- **1892**: get virtual address of internal table entry that points to current page-to-remove
- **1896**: get physical address of page
- **1899**: get virtual address of page
- **1900**: free page
- **1901**: mark internal page table entry as "pointing at no page"

Called by:

- `allocuvm`
- `freevm`
- `growproc`

`1853 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)`

Allocate page tables and physical memory to grow process from `oldsz` to `newsz` . returns `newsz` if succeeded, 0 otherwise.

Called by:

- `growproc`
- `exec`

`5910 exec(char *path, char **argv)`

Replaces current process with new one.

5920-5949: load da code

- **5920**: open file
- **5926**: read ELF header
- **5936-5947**: loop over sections in `proghdr` :
  - **5937**: read section from file
  - **5943**: allocate memory
  - **5945**: load code and data
- **5948**: close file

5952-5956: allocate user-stack and guard page

- **5952**: round up address in order to add stack and guard-page at new page
- **5953**: allocate two pages, for stack and guard page
- **5955**: remove user-mode bit from guard page
- **5956**: set stack pointer to point to stack page

5959-5967: push arguments to new process user-stack

- **5959**: loop over arguments:
  - **5962**: move new process stack pointer so there's room for current argument
  - **5963**: copy argument to actual new process user-stack

    - **5965**: make `argv` vector entry (which is still in temporary `ustack` variable!) point to current argument that we just pushed to stack

- **5967**: put 0 in last entry of `argv` vector (which is still in temporary `ustack` variable!), as is expected by convention

- **5969**: put -1 as return address in appropriate spot in temporary `ustack` variable

- **5970**: put `argc` in appropriate spot in temporary `ustack` variable

- **5971**: put address of where `argv` will be in new user-stack (but isn't there yet) in appropriate spot in temporary `ustack` variable

- **5974**: now that all arguments are copied to new user-stack, and we know where to place return address & `argc` & `argv`, copy `ustack` variable to new process user-stack

5978-5981: copy new process name

- **5978-5980**: get part of the name after all the slashes

5984-5991: switch address space and fix trapframe

- **5984**: save old page table for freeing later

- **5985**: give proc new page table!

- **5986**: give proc new size

- **5987**: set trapframe's `eip` to new process entry point

- **5988**: set trapframe's `sp` to new process user-stack

- **5989**: make `%CR3` point to new page table (which doesn't harm our current running!)

- **5990**: free old page table

- **5991**: return to syscall, which returns to alltraps, which returns to popall, which returns to iret, which pops a bunch of values to actual registers, which include `%eip`, which makes us actually continue with the new process

`1818 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)`

Loads a `sz` -sized program section from `ip` file (in `offset` offset) to address `addr` (which is already mapped in `pgdir`).
Returns 0 if successful, -1 otherwise.

1825-1835: loop page by page (because the pages mapped in `pgdir` are scattered (and we can't rely on the Paging Unit to handle this, because `pgdir` is not our current page table))

- **1826**: validate that page is already mapped
- **1828**: get physical address
- **1829-1832**: take care of case where what's left to read is less than page
- **1833**: copy code-n-data

Called by `exec`

`2002 uva2ka(pde_t *pgdir, char *uva)`

Returns the kernel virtual address of a user virtual address.
Only works for addresses of pages (and not for middle of page).

2006: get `pte` entry

2011: get offset, do `p2v` to it, and return result

`2018 copyout(pde_t *pgdir, uint va, void *p, uint len)`

Copies `len` bytes from `p` address to `pgdir->va` address.

2029: get `va` offset within its page

2032: copy data

`2354 exit(void)`

Exists current process.

2359-2360: make sure we're not the First Process

2363-2368: close all files opened by user-code

2370-2371: close current working-directory

2376: let parent know we're exiting (in case parent called `wait` for child to exit) (The `chan` the parent is waiting for is the parent's `proc` address)

2379-2385: pass abandoned children to the First Process

- **2382-2383**: if child did `exit`, let the First Process know

2388: become zombie

2389: awaken the `scheduler`

`5225 filealloc(void)`

Finds the first free slot in the global file table, and returns its *address*.
If there are none free, returns 0.

`5438 fdalloc(struct file *f)`

Finds the first free slot in the process's file table, points it to `f`, and returns the index (AKA the file descriptor).
If no room, returns -1.

`5252 filedup(struct file *f)`

Increments the reference count of `f`.
(Used as part of the file duplication process)

`5451 sys_dup(void)`

Duplicates proc's reference to file.

5458: actual duplication

5460: increment ref count

`5419 argfd(int n, int *pfd, struct file **pf)`

Gets the `n` th argument sent to the system call, as a file descriptor.
Returns descriptor and the struct file it points to.
(The only reason we need the file descriptor `pfd` is in case we're closing the file and need to make `ofile` point to null.)

`5315 fileread(struct file *f, char *addr, int n)`

Reads from `f` to `addr` .

5319: make sure can read

5321-5322: handle case when file is pipe

5323-5329: handle case when file is inode:

- **5324**: lock the inode (because we must)
- **5325**: read
- **5326**: update offset
- **5327**: unlock the inode

Called by `sys_read` .

`5352 filewrite(struct file *f, char *addr, int n)`

Writes from `addr` to `f` .

5358-5359: handle case when file is pipe

5360-5386: handle case when file is inode:

- **5367-5372**: break up data to managable chunks (for transactions)
- **5374-5379**: write chunk:
    - **5374**: begin transaction
    - **5376-5377**: write
    - **5379**: end transaction

`5264 fileclose(struct file *f)`

Decrements file reference count.
When no references left, actually close file.

5271-5273: decrement ref count

5275: keep backup of struct file, because we're about to release the lock on the file table (and anything can happen after *that*)

5278: release file table because closing file on disk can take a long time

5280-5281: handle case when file is pipe (needs to happen once for `read` and once for `write` descriptors of pipe for it to actually close)

5282-5286: handle case when file is inode (using transaction because this can require writing on device)

`5851 sys_pipe(void)`

Allocates two files (read pipe and write pipe).
Expects a vector with two entries (from the input), in order to return the descriptors in.

5859: allocate pipe (creates 2 file structs)

5862: allocate slots in `ftable`, and point them to the pipe

5863-5867: remove files in case of failure

5869-5870: put descriptors in vector, for user code

`5701 sys_open(void)`

Opens or creates inode.

5710-5715: create inode (on disk!) - can only create file (not directory)

5716-5724: open inode

- **5720-5722**: if tried opening directory in *write* mode, close-n-error

5726: create struct for inode, add it to `ftable`

5734-5738: set file struct data

`5011 dirlookup(struct inode *dp, char *name, uint *poff)`

Finds an inode *under* `dp` with name that's equal to `name`.
(`poff` in an optional pointer to the offset of the found inode.)

5020: read single `struct dirent` in `dp`

5022: check whether `inum` of current `dirent` is active

5024: check whether name of current `dirent` equals `name`

If we found the droid we're looking for:

5026-5027: store offset in `poff` (if we supplied the optional pointer)

5028-5029: get actual inode

...And if we did not find it:

5033: return 0

`5115 skipelem(char *path, char *name)`

A helper function that helps us take apart "long/path/names".
Returns the value of `path` *without* the first part, and sets `name` to equal the chopped off head.
(Ignores the first instance of "/" in the path: "**/**a/b/c" acts the same as "a/b/c".)

`5189 namei(char *path)`

Returns the inode with the matching path.
(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

5192: ask `namex` to do the work

`5196 nameiparent(char *path, char *name)`

Returns the inode with the matching path *without the last part*.
(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

5198: ask `namex` to do the work

5154 namex(char *path, int nameiparent, char *name)

Returns the inode with the matching path.
(If `nameiparent` is 0, finds actual inode. Else, ignores last part (for case when we want to *create* a *new* inode).)

5158-5161: check if we need to look in root or current working directory

5163-5180: loop over parts in path:

- **5164**: `ilock` current inode, because we must
- **5165-5168**: make sure we're looking at a *directory*
- **5169-5173**: if we're looking for all but last (and found it), return it
- **5174-5177**: try to get the next part of the path under the current inode
- **5179**: set current inode to be the found path part inode

5181-5184: if looking for all but last (and haven't found id before), return 0

5185: happily return our found inode

5052 dirlink(struct inode *dp, char *name, uint inum)

Writes a new directory entry (name, inum) into the inode pointed at by `dp` .

5059-5062: make sure name doesn't already exist in inode

5065-5070: find empty slot in inode

5072: prepare to write name

5073: prepare to write inum

5074: actually write the data (if we reached the end of the inode, `writei` will increase its size)

5657 create(char *path, short type, short major, short minor)

Creates and returns inode supplied in `path` (expecting `path` to exist up to its last part, which is the inode we're creating).
If inode already exists, opens it.

5663: get parent of requested inode

5667-5674: check if inode exists (and is a file!)
If so, return it.
If exists but is not file, return 0.

5676: get an actual inode from the disk

5679: lock the new inode (because we must)

5680-5682: set some values. nlink is 1, because parent is linked to new inode

5683: updates new values in the disk

5689: add the two links every directory has:

- "." self
- ".." parent

5693: add the new inode to parent

4654 `iget(uint dev, uint inum)`

Opens (and returns) inode.

4661-4670: loop over `icache`, looking if inode was already loaded once

- **4663-4667**: check if the inode is already in the cache
- **4668-4669**: find first empty slot, in case we'll need to load inode to it

4673-4674: if cache is full (and inode not there), panic (although we could have also gone to sleep till there's room)

4676-4680: load some of the inode data to `icache`. Valid bit is set to 0, because we haven't yet read the data from the disk

4689 `idup(struct inode *ip)`

Increments reference count of inode (and returns it).

4691: lock **`icache`** before use, because we access inode field

4756 `iput(struct inode *ip)`

Decrements reference count of inode, and closes it on the disk if this it will no longer be referenced at all.

4759: check if:

1. We're about to close the inode's last reference
2. We're closing a valid inode
3. There are no more names of / links to the inode (so it must be destroyed)

If so:

- **4761**: mark inode as busy (since there's still one reference!), just in case
- **4765**: actually delete inode on disk
- **4767**: update the inode (???)
- **4770**: do something that isn't really needed

4772: finally decrement the ref count

`4603 ialloc(uint dev, short type)`

Allocates a new inode *on the disk*, and then *in the memory*.

4610: read super-block from disk.

4612: loop over all inodes on disk:

- **4613**: read block of current inode
- **4614**: calculate pointer to current inode (using casting so that `+ inum%IPB` will add the correct size)
- **4615**: check if inode is free. If so:
  - **4616**: clear inode data
  - **4617**: set type
  - **4618**: re-write entire block, marking new inode as used (because we set type)
  - **4619**: close current block
  - **4620**: allocate inode in memory and return the *memory* inode
- **4622**: close current block

4624: no more inodes on disk?! Panic!

4629 `iupdate(struct inode *ip)`

Updates inode-on-disk from inode-on-memory `ip`.

4634: read entire block from disk

4635: calculate pointer to inode-on-disk (using casting so that `+ inum%IPB` will add the correct size)

4636-4641: set inode data to copy of block that is on the memory now

4642: re-write entire block on disk

4643: close block

4703 `ilock(struct inode *ip)`

Locks inode, without spinning or preventing interrupts.

4711-4715: sleep over inode until it's not busy:

- **4712-4713**: while inode is busy, go back to sleep
- **4714**: mark inode as busy

4717-4730: make sure inode is still valid - affects only in-memory inode

- **4718**: read entire block
- **4719**: calculate pointer to inode
- **4720-4725**: set data (just in case it changed meanwhile
- **4726**: release block
- **4727**: mark as valid

4735 `iunlock(struct inode *ip)`

Unlocks inode.

4741: remove "busy" flag

4742: wakeup all procs waiting for inode lock.

5751 sys_mkdir(void)

Creates a new directory.

5513 sys_link(void)

Creates a new name (or *shortcut*) for a file.
(But not for a directory!)

5601 sys_unlink(void)

Destroys a name of a file or directory.

4902 readi(struct inode *ip, char *dst, uint off, uint n)

Actually reads data from the disk.

4907-4911: (weird stuff beyond the scope of this course)

4913-4914: offset validation

4915-4916: uh...

4918: for each block from offset till block where we want to finish reading:

- **4919**: read entire current block
- **4920**: figure out how much to read (entire block, or just part)
- **4921**: copy the how-much-to-read data to memory (buffer)
- **4922**: close current block

4952 writei(struct inode *ip, char *src, uint off, uint n)

Actually reads data from the disk.

4957-4961: (weird stuff beyond the scope of this course)

4963-4966: validation

4968: for each block from offset will block where we want to finish writing:

- **4969** read entire current block (we actually don't really need to do this when we're writing a whole block, but whatever)
- **4970**: figure out how much to write (entire block, or just part)
- **4971**: copy the how-much-to-write data to memory (buffer)
- **4972**: copy buffer to disk
- **4972**: close current block

4976-4979: if the file grew because of the write, update inode's *size*

`4856 itrunc(struct inode *ip)`

Destroys inode on disk!
(Must be called only when inode is no longer referenced or held open by anyone.)

4862-4867: free direct blocks (if they're allocated) and mark them as such on inode

4869: if there are also indirect blocks:

- **4870**: read the block with the indirect pointers
- **4871**: cast block to int vector, for convenience
- **4872-4875**: free indirect blocks (if they're allocated) - no need to mark them on indirect vector, because we'll destory him soon
- **4876**: close the block with the indirect pointers
- **4877**: destory the block with the (now freed) indirect pointers
- **4878**: mark the indirect pointer as free on the inode

4881-4882: update inode

`4810 bmap(struct inode *ip, uint bn)`

Returns the physical block number of `ip` 's `bn` th block.
If block doesn't exist, the block is allocated.

4815-4819: handle case when block is direct

- **4816**: try to get physical address

- **4817**: if no physical address, allocate one

If we reached here, then we know block is indirect

4824-4825: allocate indirect block if it doesn't exist yet

4826: read contents of indirect block

4827: cast the contents as a vector of numbers

4828-4831: try to get physical address

- **4829**: allocate new address if needed
- **4830**: write new address on the indrect block on the disk

4832: close indirect block

`4102 bread(uint dev, uint sector)`

Gets a block from the disk.

4106: try to get buffer from cache

4107-4108: if buffer is not valid, ask for the actual buffer from the driver (which is the layer that *really* really reads from the disk

`4114 bwrite(struct buf *b)`

Writes a block to the disk.

4116-4117: make sure buffer is marked as busy

4118: mark buffer as dirty (that's our way to tell the driver to *write*)

4119: ask driver layer to write to disk.

`4038 binit(void)`

Initialize `bcache` buffer cache.

`4066 bget(uint dev, uint sector)`

Gets a buffer from the cache.
If it's not there, allocate it there.

4070: lock buffer cache

4072-4084: search for buffer in cache

- **4075**: buffer found!
    - **4076-4080**: if buffer is not busy, unlock the buffer cache, mark buffer as busy, and return it
    - **4081-4082**: if buffer is busy, go to sleep till it's unlocked (but will need to search again for case buffer was changed)

Got here?
Buffer not found; allocate new buffer

4087: loop from end of list to beginning

- **4088**: check if current buff is not busy and not dirty
    - **4089-4093**: fill data, release lock, return buff

`4125 brelse(struct buf *b)`

Release buffer from being BUSY and move to head of linked list.

4130: lock cache

4132-4137: reposition buff to list head

4139: mark buff as not busy

4140: wake up anyone who might be waiting for buff

4142: release cache lock

`4454 balloc(uint dev)`

Allocates and zeroes a block on the disk.

4461 read super block

`bfree`

`idewait`

`idestart`

`3954 iderw(struct buf *b)`

Handles a queue of blocks to write
(because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk).

3968-3971: append `b` to end of `idequeue`

3974-3975: if our `b` is at the head of the list, read/write it to IDE

3978-3980: sleep until `b` is VALID and NOT DIRTY

`3902 ideintr(void)`

Handles the interrupt from the IDE (which means the disk finished reading/writing).
Manages `idequeue`.

3907-3913: get current head of `idequeue`, and move `idequeue` to next guy.

- **3908-3912**: handle false alarms (which happen sometimes)
- **3913**: "increment" queue

3916-3917: read data (if that's what current buff wanted)

3920-3922: clear buff flags and wakeup all those waiting for buff

3925-3926: tell IDE to start running on next guy in queue