Linux Kernel Internals

Table of Contents

Linux Kernel Internals

Tigran Aivazian `tigran@veritas.com`

22 August 2000

Introduction to the Linux 2.4 kernel. The latest copy of this document can be always downloaded from: http://www.moses.uklinux.net/patches/lki.sgml This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. The author is working as senior Linux kernel engineer at VERITAS Software Ltd and wrote this book for the purpose of supporting the short training course/lectures he gave on this subject, internally at VERITAS.

3.Virtual Filesystem (VFS)

1.Booting

1.1 Building the Linux Kernel Image

This section explains the steps taken during compilation of the Linux kernel and the output produced at each stage. The build process depends on the architecture so I would like to emphasize that we only consider building a Linux/x86 kernel.

When the user types 'make zImage' or 'make bzImage' the resulting bootable kernel image is stored as `arch/i386/boot/zImage` or `arch/i386/boot/bzImage` respectively. Here is how the image is built:

1. C and assembly source files are compiled into ELF relocatable object format (.o) and some of them are grouped logically into archives (.a) using ar(1)
2. Using ld(1), the above .o and .a are linked into 'vmlinux' which is a statically linked, non−stripped ELF 32−bit LSB 80386 executable file
3. System.map is produced by 'nm vmlinux', irrelevant or uninteresting symbols are grepped out.
4. Enter directory arch/i386/boot
5. Bootsector asm code bootsect.S is preprocessed either with or without −D__BIG_KERNEL__, depending on whether the target is bzImage or zImage, into bbootsect.s or bootsect.s respectively
6. bbootsect.s is assembled and then converted into 'raw binary' form called bbootsect (or bootsect.s assembled and raw−converted into bootsect for zImage)
7. Setup code setup,S (setup.S includes video.S) is preprocessed into bsetup.s for bzImage or setup.s for zImage. In the same way as the bootsector code, the difference is marked by −D__BIG_KERNEL__ present for bzImage. The result is then converted into 'raw binary' form called bsetup
8. Enter directory arch/i386/boot/compressed and convert /usr/src/linux/vmlinux to $tmppiggy (tmp filename) in raw binary format, removing .note and .comment ELF sections
9. gzip −9 < $tmppiggy > $tmppiggy.gz
10. Link $tmppiggy.gz into ELF relocatable (ld −r) piggy.o
11. Compile compression routines head.S and misc.c (still in arch/i386/boot/compressed directory) into ELF objects head.o and misc.o

12. Link together head.o misc.o piggy.o into bvmlinux (or vmlinux for zImage, don't mistake this for /usr/src/linux/vmlinux!). Note the difference between −Ttext 0x1000 used for vmlinux and −Ttext 0x100000 for bvmlinux, i.e. for bzImage compression loader is high−loaded
13. Convert bvmlinux to 'raw binary' bvmlinux.out removing .note and .comment ELF sections
14. Go back to arch/i386/boot directory and using the program tools/build cat together bbootsect + bsetup + compressed/bvmlinux.out into bzImage (delete extra 'b' above for zImage). This writes important variables like setup_sects and root_dev at the end of the bootsector.

The size of the bootsector is always 512 bytes. The size of the setup must be greater than 4 sectors but is limited above by about 12K − the rule is:

0x4000 bytes >= 512 + setup_sects * 512 + room for stack while running bootsector/setup

We will see later where this limitation comes from.

The upper limit on the bzImage size produced at this step is about 2.5M for booting with LILO and 0xFFFF paragraphs (0xFFFF0 = 1048560 bytes) for booting raw image, e.g. from floppy disk or CD−ROM (El−Torito emulation mode).

Note, that tools/build validates the size of the boot sector, of the kernel image and lower bound on the size of setup but not the upper bound of setup so it is easy to build a broken kernel by adding some large ".space" at the end of setup.S.

1.2 Booting: Overview

The boot process details are architecture−specific so we shall focus our attention on the IBM PC/IA32 architecture. Due to old design and backward compatibility, the PC firmware boots the operating system in an old−fashioned manner. This process can be separated into the following six logical stages:

1. BIOS selects the boot device
2. BIOS loads the bootsector from the boot device
3. Bootsector loads setup, decompression routines and compressed kernel image
4. The kernel is uncompressed in protected mode
5. Low−level initialisation performed by asm code
6. High−level C initialisation

1.3 Booting: BIOS POST

1. The power supply starts the clock generator and asserts #POWERGOOD signal on the bus
2. CPU #RESET line is asserted (CPU now in real 8086 mode)
3. %ds=%es=%fs=%gs=%ss=0, %cs:%eip = 0xFFFF:0000 (ROM BIOS POST code)
4. All the checks performed by POST with interrupts disabled
5. IVT initialised at address 0

6. The BIOS Bootstrap Loader function is invoked via int 0x19 with %dl containing the boot device 'drive number'. This loads track 0, sector 1 at physical address 0x7C00 (0x07C0:0000).

1.4 Booting: bootsector and setup

The bootsector used to boot Linux kernel could be either:

- Linux bootsector, arch/i386/boot/bootsect.S
- LILO (or other bootloader's) bootsector
- No bootsector (loadlin etc)

We consider here the Linux bootsector in detail. The first few lines initialize the convenience macros to be used for segment values:

```
29 SETUPSECS = 4                /* default nr of setup-sectors */
30 BOOTSEG   = 0x07C0           /* original address of boot-sector */
31 INITSEG   = DEF_INITSEG      /* we move boot here - out of the way */
32 SETUPSEG  = DEF_SETUPSEG     /* setup starts here */
33 SYSSEG    = DEF_SYSSEG       /* system loaded at 0x10000 (65536) */
34 SYSSIZE   = DEF_SYSSIZE      /* system size: # of 16-byte clicks */
```

(the numbers on the left are the line numbers of bootsect.S file) The values of DEF_INITSEG, DEF_SETUPSEG, DEF_SYSSEG, DEF_SYSSIZE are taken from `include/asm/boot.h`:

```
/* Don't touch these, unless you really know what you're doing. */
#define DEF_INITSEG     0x9000
#define DEF_SYSSEG      0x1000
#define DEF_SETUPSEG    0x9020
#define DEF_SYSSIZE     0x7F00
```

Now, let us consider the actual code of bootsect.S:

```
    54          movw    $BOOTSEG, %ax
    55          movw    %ax, %ds
    56          movw    $INITSEG, %ax
    57          movw    %ax, %es
    58          movw    $256, %cx
    59          subw    %si, %si
    60          subw    %di, %di
```

```
61          cld
62          rep
63          movsw
64          ljmp    $INITSEG, $go

65  # bde − changed 0xff00 to 0x4000 to use debugger at 0x6400 up (bde).  We
66  # wouldn't have to worry about this if we checked the top of memory.  Also
67  # my BIOS can be configured to put the wini drive tables in high memory
68  # instead of in the vector table.  The old stack might have clobbered the
69  # drive table.

70  go:     movw    $0x4000−12, %di         # 0x4000 is an arbitrary value >=
71                                          # length of bootsect + length of
72                                          # setup + room for stack;
73                                          # 12 is disk parm size.
74          movw    %ax, %ds                # ax and es already contain INITSEG
75          movw    %ax, %ss
76          movw    %di, %sp                # put stack at INITSEG:0x4000−12.
```

The lines 54−63 move the bootsector code from address 0x7C00 to 0x90000. This is achieved by:

1. set %ds:%si to $BOOTSEG:0 (0x7C0:0 = 0x7C00)
2. set %es:%di to $INITSEG:0 (0x9000:0 = 0x90000)
3. set the number of 16bit words in %cx (256 words = 512 bytes = 1 sector)
4. clear DF (direction) flag in EFLAGS to auto−increment addresses (cld)
5. go ahead and copy 512 bytes (rep movsw)

The reason this code does not use "rep movsd" is intentional (hint − .code16).

The line 64 jumps to the label "go:" in the newly made copy of the bootsector, i.e. in the segment 0x9000. This and the following three instructions (lines 64−76) prepare the stack at $INITSEG:0x4000−12, i.e. %ss = $INITSEG (0x9000) and %sp = 0x3FEE (0x4000−12). This is where the limit on setup size comes from that we mentioned earlier (see Building the Linux Kernel Image).

The lines 77−103 patch the disk parameter table for the first disk to allow multi−sector reads:

```
77  # Many BIOS's default disk parameter tables will not recognize
78  # multi−sector reads beyond the maximum sector number specified
79  # in the default diskette parameter tables − this may mean 7
80  # sectors in some cases.
81  #
82  # Since single sector reads are slow and out of the question,
83  # we must take care of this by creating new parameter tables
84  # (for the first disk) in RAM.  We will set the maximum sector
85  # count to 36 − the most we will encounter on an ED 2.88.
86  #
87  # High doesn't hurt.  Low does.
88  #
89  # Segments are as follows: ds = es = ss = cs − INITSEG, fs = 0,
90  # and gs is unused.

91          movw    %cx, %fs                # set fs to 0
```

```
 92         movw    $0x78, %bx              # fs:bx is parameter table address
 93         pushw   %ds
 94         ldsw    %fs:(%bx), %si          # ds:si is source
 95         movb    $6, %cl                 # copy 12 bytes
 96         pushw   %di                     # di = 0x4000−12.
 97         rep                             # don't need cld −> done on line 66
 98         movsw
 99         popw    %di
100         popw    %ds
101         movb    $36, 0x4(%di)           # patch sector count
102         movw    %di, %fs:(%bx)
103         movw    %es, %fs:2(%bx)
```

The floppy disk controller is reset using BIOS service int 0x13 function 0 "reset FDC" and setup sectors are loaded immediately after the bootsector, i.e. at physical address 0x90200 ($INITSEG:0x200), again using BIOS service int 0x13, function 2 "read sector(s)". This happens during lines 107−124:

```
107  load_setup:
108         xorb    %ah, %ah                # reset FDC
109         xorb    %dl, %dl
110         int     $0x13
111         xorw    %dx, %dx                # drive 0, head 0
112         movb    $0x02, %cl              # sector 2, track 0
113         movw    $0x0200, %bx            # address = 512, in INITSEG
114         movb    $0x02, %ah              # service 2, "read sector(s)"
115         movb    setup_sects, %al        # (assume all on head 0, track 0)
116         int     $0x13                   # read it
117         jnc     ok_load_setup           # ok − continue

118         pushw   %ax                     # dump error code
119         call    print_nl
120         movw    %sp, %bp
121         call    print_hex
122         popw    %ax
123         jmp     load_setup

124  ok_load_setup:
```

If loading failed for some reason (bad floppy or someone pulled the diskette out during the operation) then we dump error code and retry in an endless loop. The only way to get out of it is to reboot the machine, unless retry succeeds but usually it doesn't (if something is wrong it will only get worse).

If loading setup_sects sectors of setup code succeeded we jump to label "ok_load_setup:"

Then we proceed to load the compressed kernel image at physical address 0x10000. This is done to preserve the firmware data areas in low memory (0−64K). After the kernel is loaded we jump to $SETUPSEG:0 (arch/i386/boot/setup.S). Once the data is no longer needed (e.g. no more calls to BIOS) it is overwritten by moving the entire (compressed) kernel image from 0x10000 to 0x1000 (physical addresses, of course). This is done by setup.S which sets things up for protected mode and jumps to 0x1000 which is the head of the compressed kernel, i.e. arch/386/boot/compressed/{head.S,misc.c}. This sets up stack and calls decompress_kernel() which uncompresses the kernel to address 0x100000 and jumps to it.

Note that the old bootloaders (old versions of LILO) could only load the first 4 sectors of setup so there is code in setup to load the rest of itself if needed. Also, the code in setup has to take care of various combinations of loader type/version vs zImage/bzImage and is therefore highly complex.

Let us examine the kludge in the bootsector code that allows to load a big kernel, known also as "bzImage". The setup sectors are loaded as usual at 0x90200 but the kernel is loaded 64K chunk at a time using a special helper routine that calls BIOS to move data from low to high memory. This helper routine is referred to by bootsect_kludge in bootsect.S and is defined as bootsect_helper in setup.S. The bootsect_kludge label in setup.S contains the value of setup segment and the offset of bootsect_helper code in it so that bootsector can use lcall instruction to jump to it (inter−segment jump). The reason why it is in setup.S is simply because there is no more space left in bootsect.S (which is strictly not true − there are approx 4 spare bytes and at least 1 spare byte in bootsect.S but that is not enough, obviously). This routine uses BIOS service int 0x15 (ax=0x8700) to move to high memory and resets %es to always point to 0x10000 so that the code in bootsect.S doesn't run out of low memory when copying data from disk.

1.5 Using LILO as a bootloader

There are several advantages in using a specialized bootloader (LILO) over a bare bones Linux bootsector:

1. Ability to choose between multiple Linux kernels or even multiple OSes.
2. Ability to pass kernel command line parameters (there is a patch called BCP that adds this ability to bare−bones bootsector+setup)
3. Ability to load much larger bzImage kernels − up to 2.5M vs 1M

Old versions of LILO (v17 and earlier) could not load bzImage kernels. The newer versions (as of a couple of years ago or earlier) use the same technique as bootsect+setup of moving data from low into high memory by means of BIOS services. Some people (Peter Anvin notably) argue that zImage support should be removed. The main reason (according to Alan Cox) it stays is that there are apparently some broken BIOSes that make it impossible to boot bzImage kernels while loading zImage ones fine.

The last thing LILO does is to jump to setup.S and things proceed as normal.

1.6 High level initialisation

By "high−level initialisation" we consider anything which is not directly related to bootstrap, even though parts of the code to perform this are written in asm, namely arch/i386/kernel/head.S which is the head of the uncompressed kernel. The following steps are performed:

1. initialises segment values (%ds=%es=%fs=%gs=__KERNEL_DS= 0x18)
2. initialises page tables
3. enables paging by setting PG bit in %cr0
4. zero−cleans BSS (on SMP, only first CPU does this)
5. copies the first 2k of bootup parameters (kernel commandline)
6. checks CPU type using EFLAGS and, if possible, cpuid, able to detect 386 and higher

7. the first CPU calls start_kernel(), all others call arch/i386/kernel/smpboot.c:initialize_secondary() if ready=1, which just reloads esp/eip and doesn't return.

The init/main.c:start_kernel() is written in C and does the following:

1. takes a global kernel lock (it is needed so that only one CPU goes through initialisation)
2. performs arch−specific setup (memory layout analysis, copying boot command line again, etc.)
3. prints Linux kernel "banner" containing the version, compiler used to build it etc. to the kernel ring buffer for messages. This is taken from the variable linux_banner defined in init/version.c and is the same string as displayed by "cat /proc/version".
4. initialises traps
5. initialises irqs
6. initialises data required for scheduler
7. initialises time keeping data
8. initialises softirq subsystem
9. parses boot commandline options
10. initialises console
11. if module support was compiled into the kernel, initialises dynamical module loading facility
12. if "profile=" command line was supplied initialises profiling buffers
13. kmem_cache_init(), initialises most of slab allocator
14. enables interrupts
15. calculates BogoMips value for this CPU
16. calls mem_init() which calculates max_mapnr, totalram_pages and high_memory and prints out the "Memory: ..." line
17. kmem_cache_sizes_init(), finishes slab allocator initialisation
18. initialises data structures used by procfs
19. fork_init(), creates uid_cache, initialises max_threads based on the amount of memory available and configures RLIMIT_NPROC for init_task to be max_threads/2
20. creates various slab caches needed for VFS, VM, buffer cache etc
21. if System V IPC support is compiled in, initialises IPC subsystem. Note, that for System V shm this includes mounting an internal (in−kernel) instance of shmfs filesystem
22. if quota support is compiled into the kernel, create and initialise a special slab cache for it
23. performs arch−specific "check for bugs" and, whenever possible, activates workaround for processor/bus/etc bugs. Comparing various architectures reveals that "ia64 has no bugs" and "ia32 has quite a few bugs", good example is "f00f bug" which is only checked if kernel is compiled for less than 686 and worked around accordingly
24. sets a flag to indicate that a schedule should be invoked at "next opportunity" and creates a kernel thread init() which execs execute_command if supplied via "init=" boot parameter or tries to exec /sbin/init,/etc/init,/bin/init,/bin/sh in this order and if all these fail, panics with suggestion to use "init=" parameter.
25. goes into the idle loop, this is an idle thread with pid=0

Important thing to note here that the init() kernel thread calls do_basic_setup() which in turn calls do_initcalls() which goes through the list of functions registered by means of __initcall or module_init() macros and invokes them. These functions either do not depend on each other or their dependencies have been manually fixed by the link order in the Makefiles. This means that depending on the position of directories in the trees and the structure of the Makefiles the order initialisation functions are invoked can change. Sometimes, this is important because you can imagine two subsystems A and B with B depending on some initialisation done by A. If A is compiled statically and B is a module then B's entry point is guaranteed to be invoked after A prepared all the necessary environment. If A is a module, then B is also necessarily a module so there are no problems. But what if both A and B are statically linked into the kernel? The order they are invoked depends on the relative entry point offsets in the ".initcall.init" ELF section of the kernel image. Rogier Wolff proposed to introduce the hierarchical "priority" infrastructure whereby modules could let the linker know in what (relative) order they should be linked but so far there are no patches available that implement this in a sufficiently elegant manner to be acceptable into the kernel. Therefore − make sure your link order is correct, if, in the example above, A and B work fine when compiled statically once they will work always provided they are listed sequentially in the same Makefile. If they don't work change the order their object files are listed.

Another thing worthy of note is Linux's ability to execute "alternative init program" by means of passing "init=" boot commandline. This is useful for recovering from accidentally overwritten "/sbin/init" or debugging the initialisation (rc) scripts and /etc/inittab by hand, executing them one at a time.

1.7 SMP Bootup on x86

On SMP, the BP goes through the normal sequence of bootsector, setup etc until it reaches the start_kernel() and then on to smp_init() and especially src/i386/kernel/smpboot.c:smp_boot_cpus(). The smp_boot_cpus() goes in a loop for each apicid (until NR_CPUS) and calls do_boot_cpu() on it. What do_boot_cpu() does is create (i.e. fork_by_hand) an idle task for the target cpu and writes in well−known locations defined by the Intel MP spec (0x467/0x469) the eip of trampoline code found in trampoline.S. Then it generates STARTUP IPI to the target cpu which makes this AP execute the code in trampoline.S.

The boot CPU creates a copy of trampoline code for each CPU in the low memory. The AP code writes a magic number in its own code which is verified by the BP to make sure that AP is executing the trampoline code. The requirement that trampoline code must be in low memory is enforced by the Intel MP specification.

The trampoline code simply sets %bx register to 1, enters protected mode and jumps to startup_32 which is the main entry to arch/i386/kernel/head.S.

Now, the AP starts executing head.S and discovering that it is not a BP, it skips the code that clears BSS and then enters initialise_secondary() which just enters the idle task for this CPU − recall that init_tasks[cpu] was already initialised by BP executing do_boot_cpu(cpu).

Note, that init_task can be shared but each idle thread must have its own TSS so init_tss[NR_CPUS] is an array.

1.8 Freeing initialisation data and code

When the operating system initialises itself most of the code and data structures are never needed again. Most operating systems (BSD, FreeBSD etc.) cannot dispose of this unneeded information thus wasting the precious physical kernel memory. The excuse they use (see McKusick's 4.4BSD book) is that "the relevant code is spread around various subsystems and so it is not feasible to free it". Linux, of course, cannot use such excuses because under Linux "if something is possible in principle, then it is already implemented or somebody is working on it".

So, as I said earlier, Linux kernel can only be compiled as ELF binary and now we find out the reason (or one of the reasons) for that. The reason related to throwing away initialisation code/data is that Linux provides two macros to be used:

- __init − for initialisation code
- __initdata − for data

These evaluate to gcc attribute specificators (also known as "gcc magic") as defined in include/linux/init.h:

```
#ifndef MODULE
#define __init        __attribute__ ((__section__ (".text.init")))
#define __initdata    __attribute__ ((__section__ (".data.init")))
#else
#define __init
#define __initdata
#endif
```

What this means is that if the code is compiled statically into the kernel (i.e. MODULE is not defined) then it is placed in the special ELF section ".text.init" which is declared in the linker map in arch/i386/vmlinux.lds. Otherwise (i.e. if it is a module) the macros evaluate to nothing.

What happens during boot is that the "init" kernel thread (function init/main.c:init()) calls the arch−specific function free_initmem() which frees all the pages between addresses __init_begin and __init_end.

On a typical system (my workstation), this results in freeing about 260K of memory.

The functions registered via module_init() are placed in ".initcall.init" which is also freed in the static case. The current trend in Linux, when designing a subsystem (not necessarily a module) is to provide init/exit entry points from the early stages of design so that in the future the subsystem in question can be modularized if needed. Example of this is pipefs, see fs/pipe.c. Even if subsystem will never become a module, e.g. bdflush (see fs/buffer.c) it is still nice and tidy to use module_init() macro against its initialisation function, provided it does not matter when exactly is the function called.

There are two more macros which which work very similar, called __exit and __exitdata but they are more directly connected to the module support and therefore will be explained in a later section.

1.9 Processing kernel command line

Let us recall what happens to the commandline passed to kernel during boot.

1. LILO (or BCP) accepts the commandline using BIOS keyboard services and stores it at a well−known location in physical memory, as well as a signature saying that there is a valid commandline there

2. arch/i386/kernel/head.S copies the first 2k of it out to the zeropage. Note that current version (21) of LILO chops the commandline to 79 bytes. This is a nontrivial bug in LILO (when large EBDA support is enabled) and Werner promised to fix it sometime soon. If you really need to pass commandlines longer than 79 bytes then you can either use BCP or hardcode your commandline in arch/i386/kernel/setup.c:parse_mem_cmdline() function
3. arch/i386/kernel/setup.c:parse_mem_cmdline() (called by setup_arch() called by start_kernel()) copies 256 bytes from zeropage into saved_command_line which is displayed by /proc/cmdline. This same routine processes "mem=" portion and makes appropriate adjustments to VM parameters
4. we return to commandline in parse_options() (called by start_kernel()) which processes some "in−kernel" parameters (currently "init=" and environment/arguments for init) and passes each word to checksetup()
5. checksetup() goes through the code in ELF section ".setup.init" and invokes each function passing it the word if it matches. Note that using the return value of 0 from the function registered via __setup() it is possible to pass the same "variable=value" to more than one function with "value" invalid to one and valid to another. Jeff Garzik commented: "hackers who do that get spanked :)" Why? Because this is clearly ld−order specific, i.e. kernel linked in one order will have functionA invoked before functionB and another will have it in reversed order with the result depending on the order

So, how do we write code that processes boot commandline? We use __setup() macro defined in include/linux/init.h:

```
/*
 * Used for kernel command line parameter setup
 */
struct kernel_param {
        const char *str;
        int (*setup_func)(char *);
};

extern struct kernel_param __setup_start, __setup_end;

#ifndef MODULE
#define __setup(str, fn) \
   static char __setup_str_##fn[] __initdata = str; \
   static struct kernel_param __setup_##fn __initsetup = \
   { __setup_str_##fn, fn }

#else
#define __setup(str,func) /* nothing */
endif
```

So, you would typically use it in your code like this (taken from code of real driver, BusLogic HBA drivers/scsi/BusLogic.c):

```
static int __init
BusLogic_Setup(char *str)
{
        int ints[3];
        (void)get_options(str, ARRAY_SIZE(ints), ints);

        if (ints[0] != 0) {
                BusLogic_Error("BusLogic: Obsolete Command Line Entry "
                                 "Format Ignored\n", NULL);
                return 0;
        }
        if (str == NULL || *str == '\0')
                return 0;
        return BusLogic_ParseDriverOptions(str);
}

__setup("BusLogic=", BusLogic_Setup);
```

Note, that for modules __setup() does nothing so the code that wishes to process boot commandline and can be either a module or statically linked must invoke its parsing function manually in the module initialisation routine "manually". This also means that it is possible to write code that processes parameters when compiled as a module but not when it is static or vice versa.

2.Process and Interrupt Management

2.1 Task Structure and Process Table

Every process under Linux is dynamically allocated a 'struct task_struct' structure. The maximum number of processes that can be created on the Linux system is limited only by the amount of physical memory present, and is equal to (see kernel/fork.c:fork_init()):

```
        /*
         * The default maximum number of threads is set to a safe
         * value: the thread structures can take up at most half
         * of memory.
         */
        max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;
```

which on IA32 architecture basically means 'num_physpages/4' so, for example on 512M machine you can create 32k threads which is a considerable improvement over the 4k−epsilon limit for older (2.2 and earlier) kernels. Moreover, this can be changed at runtime using KERN_MAX_THREADS sysctl(2) or simply using procfs interface to kernel tunables:

```
# cat /proc/sys/kernel/threads−max
32764
# echo 100000 > /proc/sys/kernel/threads−max
# cat /proc/sys/kernel/threads−max
100000
# gdb −q vmlinux /proc/kcore
Core was generated by `BOOT_IMAGE=240ac18 ro root=306 video=matrox:vesa:0x118'.
#0  0x0 in ?? ()
(gdb) p max_threads
$1 = 100000
```

The set of processes on the Linux system is represented as a collection of 'struct task_struct' structures which are linked in two ways:

1. as a hashtable, hashed by pid
2. as a circular, doubly−linked list using p−>next_task and p−>prev_task pointers

The hashtable is called pidhash[] and is defined in include/linux/sched.h:

```
/* PID hashing. (shouldnt this be dynamic?) */
#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];

#define pid_hashfn(x)   ((((x) >> 8) ^ (x)) & (PIDHASH_SZ − 1))
```

The tasks are hashed by their pid value and the above hashing function is supposed to distribute the elements uniformly in their domain (0 to PID_MAX−1). The hashtable is used to quickly find a task by given pid, using find_task_pid() inline from include/linux/sched.h:

```
static inline struct task_struct *find_task_by_pid(int pid)
{
        struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];

        for(p = *htable; p && p−>pid != pid; p = p−>pidhash_next)
                ;

        return p;
}
```

The tasks on each hashlist (i.e. hashed to the same value) are linked by p−>pidhash_next/pidhash_pprev which are used by hash_pid() and unhash_pid() to insert and remove a given process into the hashtable. These are done under protection of the rw spinlock called 'tasklist_lock' taken for WRITE.

The circular doubly−linked list that uses p−>next_task/prev_task is maintained so that one could go through all tasks on the system easily. This is achieved by for_each_task() macro from include/linux/sched.h:

```
#define for_each_task(p) \
        for (p = &init_task ; (p = p−>next_task) != &init_task ; )
```

The users of for_each_task() should take tasklist_lock for READ. Note that for_each_task() is using init_task to mark the beginning (and end) of the list − this is safe because the idle task (pid 0) never exits.

The modifiers of the process hashtable or/and the process table links, notably fork, exit and ptrace must take the tasklist_lock for WRITE. What is more interesting is that the writers must also disable interrupts on the local cpu. The reason for this is not trivial. The send_sigio() walks the task list and thus takes tasklist_lock for READ and it is called from kill_fasync() in the interrupt context. This is why writers must disable the interrupts while the readers don't need to.

Now that we understand how the task_struct structures are linked together, let us examine the members of task_struct. They loosely corresponds to the members of UNIX 'struct proc' and 'struct user' combined together.

The other versions of UNIX separated the task state information into part which should be kept memory−resident at all times (called 'proc structure' which includes process state, scheduling information etc.) and part which is only needed when the process is running (called 'u area' which includes file descriptor table, disk quota information etc.). The only reason for such ugly design was that memory was a very scarce resource. Modern operating systems (well, only Linux at the moment but others, e.g. FreeBSD seem to improve in this direction towards Linux) do not need such separation and therefore maintain process state in a kernel memory−resident data structure at all times.

The task_struct structure is declared in include/linux/sched.h and is currently 1680 bytes in size.

The state field is declared as:

```
volatile long state;    /* −1 unrunnable, 0 runnable, >0 stopped */

#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
#define TASK_EXCLUSIVE          32
```

Why is TASK_EXCLUSIVE defined as 32 and not 16? Because 16 was used up by TASK_SWAPPING and I forgot to shift TASK_EXCLUSIVE up when I removed all references to TASK_SWAPPING (sometime in 2.3.x).

The volatile in p−>state declaration means it can be modified asynchronously (from interrupt handler):

1. **TASK_RUNNING** means the task is "supposed to be" on the run queue. The reason it may not yet be on the runqueue is that marking task as TASK_RUNNING and placing it on the runqueue is not atomic, however if you look at the queue under protection of runqueue_lock then every TASK_RUNNING is on the runqueue. The converse is not true. Namely, drivers can mark themselves (or rather the process context they run in) as TASK_INTERRUPTIBLE (or UNINTERRUPTIBLE) and then call schedule() which removes it from the runqueue (unless there is a pending signal, in which case it is left on the runqueue). speaking not true because setting state=TASK_RUNNING and placing task on the runq by wake_up_process() is not atomic so you can see (very briefly) TASK_RUNNING tasks not yet on the runq. **TASK_INTERRUPTIBLE** means the task is sleeping but can be woken up by a signal or by expiry of a timer. **TASK_UNINTERRUPTIBLE** same as TASK_INTERRUPTIBLE, except it cannot be woken up. **TASK_ZOMBIE** task has terminated but has not had its status collected (wait()−ed for) by the parent (natural or by adoption). **TASK_STOPPED** task was stopped either due to job control signals or due to ptrace(2). **TASK_EXCLUSIVE** this is not a separate state but can be OR−ed to either one of the TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE. This means that when this task is sleeping on a wait queue with many other tasks, it will be woken up alone instead of causing "thundering herd" problem by waking up all the waiters.

Task flags contain information about the process states which are not mutually exclusive:

```
unsigned long flags;    /* per process flags, defined below */
/*
 * Per process flags
 */
#define PF_ALIGNWARN    0x00000001      /* Print alignment warning msgs */
                                        /* Not implemented yet, only for 486*/
#define PF_STARTING     0x00000002      /* being created */
#define PF_EXITING      0x00000004      /* getting shut down */
#define PF_FORKNOEXEC   0x00000040      /* forked but didn't exec */
#define PF_SUPERPRIV    0x00000100      /* used super−user privileges */
#define PF_DUMPCORE     0x00000200      /* dumped core */
#define PF_SIGNALED     0x00000400      /* killed by a signal */
#define PF_MEMALLOC     0x00000800      /* Allocating memory */
#define PF_VFORK        0x00001000      /* Wake up parent in mm_release */
#define PF_USEDFPU      0x00100000      /* task used FPU this quantum (SMP) */
```

The fields p−>has_cpu,p−>processor, p−>counter, p−>priority, p−>policy and p−>rt_priority are related to the scheduler and will be looked at later.

The fields p−>mm and p−>active_mm point to the process' address space described by mm_struct structure and to the active address space if the process doesn't have a real one (e.g. kernel threads) − this is to minimize TLB flushes on switching address spaces when the task is scheduled out. So, if we are scheduling−in the kernel thread (which has no p−>mm) then its next−>active_mm will be set to the prev−>active_mm of the task that was scheduled−out which will be the same as prev−>mm if prev−>mm != NULL. The address space can be shared between threads if CLONE_VM flag is passed to the clone(2) system call or by means of vfork(2) system call.

The fields p−>exec_domain and p−>personality related to the personality of the task, i.e. to the way certain system calls behave in order to emulate "personality" of foreign flavours of UNIX.

The field p−>fs contains filesystem information, which under Linux means three pieces of information:

1. root directory's dentry and mountpoint
2. alternate root directory's dentry and mountpoint
3. current working directory's dentry and mountpoint

Also, this structure includes a reference count because it can be shared between cloned tasks when CLONE_FS flags are passed to the clone(2) system call.

The field p−>files contains the file descriptor table. This also can be shared between tasks if CLONE_FILES is specified with clone(2) system call.

The field p−>sig contains signal handlers and can be shared between cloned tasks by means of CLONE_SIGHAND flag passed to the clone(2) system call.

2.2 Creation and termination of tasks and kernel threads

Different books on operating systems define a "process" in different ways, starting from "instance of a program in execution" and ending with "that which is produced by clone(2) or fork(2) system calls". Under Linux, there are three kinds of processes:

- Idle Thread
- Kernel Threads
- User Tasks

The idle thread is created at compile time for the first CPU and then it is "manually" created for each CPU by means of arch−specific fork_by_hand() in arch/i386/kernel/smpboot.c which unrolls fork system call by hand (on some archs). Idle tasks share one init_task structure but have a private TSS structure in per−CPU array init_tss. Idle tasks all have pid = 0 and no other task can share pid, i.e. use CLONE_PID flag to clone(2).

Kernel threads are created using kernel_thread() function which invokes the clone system call in kernel mode. Kernel threads usually have no user address space, i.e. p−>mm = NULL because they explicitly do exit_mm(), e.g. via daemonize() function. Kernel threads can always access kernel address space directly. They are allocated pid numbers in the low range. Running at processor's ring 0 implies that the kernel threads enjoy all the io privileges and cannot be pre−empted by the scheduler.

User tasks are created by means of clone(2) or fork(2) system calls, both of which internally invoke kernel/fork.c:do_fork().

Let us understand what happens when a user process makes a fork(2) system call. Although the fork(2) system call is architecture−dependent due to the different ways of passing user stack and registers, the actual underlying function do_fork() that does the job is portable and is located at kernel/fork.c.

The following steps are done:

1. Local variable retval is set to −ENOMEM as it is the value errno is set to if fork(2) fails to allocate a new task structure
2. if CLONE_PID is set in clone_flags then return an error (−EPERM) unless the caller is the idle thread (during boot only). So, normal user threads cannot pass CLONE_PID to clone(2) and expect it to succeed. For fork(2) it is irrelevant as clone_flags is set to SIFCHLD − this is only relevant when do_fork() is invoked from sys_clone() which passes the clone_flags from the value requested from userspace
3. current−>vfork_sem is initialised (it is later cleared in the child). This is used by sys_vfork() (vfork(2) system call, corresponds to clone_flags = CLONE_VFORK|CLONE_VM|SIGCHLD) to make the parent sleep until the child does mm_release() for example as a result of execing another program or exit(2)−ing
4. A new task structure is allocated using arch−dependent alloc_task_struct() macro, on x86 it is just a gfp at GFP_KERNEL priority. This is the first reason why fork(2) system call may sleep. If this allocation fails we return −ENOMEM
5. All the values from current process' task structure are copied into the new one, using structure assignment *p = *current. Perhaps this should be replaced by a memset? Later on, the fields that should not be inherited by the child are set to the correct values
6. Big kernel lock is taken as the rest of the code would otherwise be non−reentrant
7. If the parent has user resources (a concept of UID, Linux is flexible enough to make it a question rather than a fact), then verify if the user exceeded RLIMIT_NPROC soft limit − if so, fail with −EAGAIN, if not, increment the count of processes by given uid p−>user−>count
8. If the system−wide number of tasks exceeds the value of the tunable max_threads, fail with −EAGAIN
9. If the binary being executed belongs to a modularised execution domain, increment the corresponding module's reference count
10. If the binary being executed belongs to a modularised binary format, increment the corresponding module's reference count
11. The child is marked as 'has not execed' p−>did_exec = 0
12. The child is marked as 'not−swappable' p−>swappable = 0
13. The child is put into 'uninterruptible sleep' state p−>state = TASK_UNINTERRUPTIBLE (TODO: why is this done? I think it's not needed − get rid of it, Linus confirms it is not needed)
14. The child's p−>flags are set according to the value of clone_flags, for the plain fork(2) it is p−>flags = PF_FORKNOEXEC.
15. The childs pid p−>pid is set using the fast algorithm in kernel/fork.c:get_pid() (TODO: lastpid_lock spinlock can be made redundant since get_pid() is always called under big kernel lock from do_fork(), also remove flags argument of get_pid, patch sent to Alan on 20/06/2000 − followup later).
16. The rest of the code in do_fork() initialises the rest of child's task structure. At the very end, the child's task structure is hashed into pidhash hashtable and the child is woken up (TODO: wake_up_process(p) sets p−>state = TASK_RUNNING and adds the process to the runq, therefore we probably didn't need to set p−>state to TASK_RUNNING earlier on in do_fork()). The interesting part is setting p−>exit_signal to clone_flags & CSIGNAL which for fork(2) means just SIGCHLD and setting p−>pdeath_signal to 0. The pdeath_signal is used when a process 'forgets' the original parent (by dying) and can be set/get by means of PR_GET/SET_PDEATHSIG commands of prctl(2) system call (You might argue that the way the value of pdeath_signal is returned via userspace pointer argument in prctl(2) is a bit silly − mea culpa, after Andries Brouwer updated the manpage it was too late to fix ;)

Thus tasks are created. There are several ways for tasks to terminate:

1. By making exit(2) system call
2. By being delivered a signal with default disposition to die
3. By being forced to die under certain exceptions
4. By calling bdflush(2) with func == 1 (this is Linux−specific, for compatibility with old distributions that still had the 'update' line in /etc/inittab − nowadays the work of update is done by kernel thread kupdate

Functions implementing system calls under Linux are prefixed with 'sys_', but they are usually concerned only with argument checking or arch−specific ways to pass some information and the actual work is done by 'do_' functions. So it is with sys_exit() which calls do_exit() to do the work. Although, other parts of the kernel sometimes invoke sys_exit(), they should really call do_exit().

The function do_exit() is found in kernel/exit.c. The points to note about do_exit():

- Uses global kernel lock (locks but doesn't unlock)
- Calls schedule() at the end which never returns
- Sets the task state to TASK_ZOMBIE
- Notifies any child with current−>pdeath_signal, if not 0
- Notifies the parent with a current−>exit_signal, which is usually equal to SIGCHLD
- Releases resources allocated by fork, closes open files etc
- On architectures that use lazy FPU switching (ia64, mips, mips64, (TODO: remove 'flags' argument of sparc, sparc64) do whatever the hardware requires to pass the FPU ownership (if owned by current) to "none"

2.3 Linux Scheduler

The job of a scheduler is to arbitrate access to the current CPU between multiple processes. Scheduler is implemented in the 'main kernel file' kernel/sched.c. The corresponding header file include/linux/sched.h is included (either explicitly or indirectly) by virtually every kernel source file.

The fields of task structure relevant to scheduler include:

- p−>need_resched, set if schedule() should be invoked at the 'next opportunity'
- p−>counter, number of clock ticks left to run in this scheduling slice, decremented by timer. When goes below or equal zero is reset to 0 and p−>need_resched set. This is also sometimes called 'dynamic priority' of a process because it can change by itself
- p−>priority, static priority, only changed through well−known system calls like nice(2), POSIX.1b sched_setparam(2) or 4.4BSD/SVR4 setpriority(2)
- p−>rt_priority, realtime priority
- p−>policy, scheduling policy, specifies which scheduling class the task belongs to. Tasks can change their scheduling class using sched_setscheduler(2) system call. The valid values are SCHED_OTHER (traditional UNIX process), SCHED_FIFO (POSIX.1b FIFO realtime process) and SCHED_RR (POSIX round−robin realtime process). One can also OR SCHED_YIELD to any of

these values to signify that the process decided to yield the CPU, for example by calling sched_yield(2) system call. FIFO realtime process runs until either a) it blocks on I/O b) explicitly yields the CPU or c) is preempted by another realtime process with a higher p−>rt_priority value. SCHED_RR is same as SCHED_FIFO except that when it's timeslice expires it goes back to the end of the runqueue

The scheduler's algorithm is simple, despite the great apparent complexity of the schedule() function. The function is complex because it implements three scheduling algorithms in one and also because of the subtle SMP−specifics.

The apparently 'useless' gotos in schedule() are there for a purpose − to generate the best optimized (for i386) code. Also, note that scheduler (like most of the kernel) was completely rewritten for 2.4 so the discussion below does not apply to 2.2 or to any other old kernels.

Let us look at the function in detail:

1. if current−>active_mm == NULL then something is wrong. Current process, even a kernel thread (current−>mm == NULL) must have a valid p−>active_mm at all times
2. if there is something to do on tq_scheduler task queue, process it now. Task queues provide a kernel mechanism to schedule execution of functions at a later time. We shall look at it in details elsewhere.
3. initialize local variables prev and this_cpu to current task and current CPU respectively
4. check if schedule() was invoked from interrupt handler (due to a bug) and panic if so
5. release the global kernel lock
6. if there is some work to do via softirq mechanism do it now
7. initialize local pointer 'struct schedule_data *sched_data' to point to per−CPU (cacheline−aligned to prevent cacheline ping−pong) scheduling data area containing TSC value of last_schedule and the pointer to last scheduled task structure (TODO: sched_data is used on SMP only but why does init_idle() initialises it on UP as well?)
8. runqueue_lock spinlock is taken. Note that we use spin_lock_irq() because in schedule() we guarantee that interrupts are enabled so when we unlock runqueue_lock we can just re−enable them instead of saving/restoring eflags (spin_lock_irqsave/restore variant)
9. task state machine: if the task is in TASK_RUNNING state it is left alone, if it is in TASK_INTERRUPTIBLE and a signal is pending then it is moved into TASK_RUNNING state. In all other cases it is deleted from the runqueue
10. next (best candidate to be scheduled) is set to the idle task of this cpu. However, the goodness of this candidate is set to a very low value of −1000 in hope that there is someone better than that.
11. if the prev (current) task is in TASK_RUNNING state, then the current goodness is set to its goodness and it is marked as a better candidate to be scheduled than the idle task
12. now the runqueue is examined and a goodness of each process that can be scheduled on this cpu is compared with current value and the process with highest goodness wins. Now the concept of "can be scheduled on this cpu" must be clarified − on UP every process on the runqueue is eligible to be scheduled, on SMP only process not already running on another cpu is eligible to be scheduled on this cpu. The goodness is calculated by a function called goodness() which treats realtime processes by making their goodness very high 1000 + p−>rt_priority, this being greater than 1000 guarantees that no SCHED_OTHER process can win so they only contend with other realtime processes that may have a greater p−>rt_priority. The goodness function returns 0 if the process' time slice (p−>counter) is over. For non−realtime processes the initial value of goodness is set to p−>counter − this way the process is less likely to get CPU if it already had it for a while, i.e. interactive processes are favoured more than cpu−bound number crunchers. The arch−specific constant PROC_CHANGE_PENALTY attempts to implement "cpu affinity" i.e. give advantage to a process

on the same cpu. It also gives slight advantage to processes with mm pointing to current active_mm or to processes with no (user) address space, i.e. kernel threads.

13. if the current value of goodness is 0 then the entire list of processes (not just runqueue!) is examined and their dynamic priorities are recalculated using simple algorithm:

    ```
    recalculate:
            {
                    struct task_struct *p;
                    spin_unlock_irq(&runqueue_lock);
                    read_lock(&tasklist_lock);
                    for_each_task(p)
                            p−>counter = (p−>counter >> 1) + p−>priority;
                    read_unlock(&tasklist_lock);
                    spin_lock_irq(&runqueue_lock);
            }
    ```

    Note that the we drop the runqueue_lock before we recalculate because we go through entire set of processes which can take a long time whilst the schedule() could be called on another cpu and select a process with goodness good enough for that cpu whilst we on this cpu were forced to recalculate. Ok, admittedly this is somewhat inconsistent because while we (on this cpu) are selecting a process with the best goodness, schedule() running on another cpu could be recalculating dynamic priorities
14. From this point on it is certain that 'next' points to the task to be scheduled so we initialise next−>has_cpu to 1 and next−>processor to this_cpu. The runqueue_lock can now be unlocked.
15. If we are switching back to the same task (next == prev) then we can simply reacquire the global kernel lock and return, i.e. skip all the hardware−level (registers, stack etc.) and VM−related (switch page directory, recalculate active_mm etc.) stuff
16. The macro switch_to() is architecture specific and (on i386) it is concerned with a) FPU handling b) LDT handling c) reloading segment registers d) TSS handling and e) reloading debug registers

2.4 Linux linked list implementation

Before we go on to examine implementation of wait queues we must acquaint ourselves with the Linux standard doubly−linked list implementation because wait queues (as well as everything else in Linux) makes heavy use of them and they are called in jargon "list.h implementation" because the most relevant file is include/linux/list.h.

The fundamental data structure here is 'struct list_head':

```
struct list_head {
        struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)
```

```
#define INIT_LIST_HEAD(ptr) do { \
        (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

#define list_entry(ptr, type, member) \
        ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))

#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)
```

The first three macros are for initialising an empty list by pointing both next and prev pointers to itself. It is obvious from C syntactical restrictions which ones should be used where − for example, LIST_HEAD_INIT() can be used for structure's element initialisation in declaration, the second can be used for static variable initialising declarations and the third can be used inside a function.

The macro list_entry() gives access to individual list element, for example: (from fs/file_table.c:fs_may_remount_ro())

```
struct super_block {
   ...
   struct list_head s_files;
   ...
} *sb = &some_super_block;

struct file {
   ...
   struct list_head f_list;
   ...
} *file;

struct list_head *p;

for (p = sb->s_files.next; p != &sb->s_files; p = p->next) {
     struct file *file = list_entry(p, struct file, f_list);
     do something to 'file'
}
```

A good example of the use of list_for_each() macro is in the scheduler where we walk the runqueue looking for the process with highest goodness:

```
static LIST_HEAD(runqueue_head);
struct list_head *tmp;
struct task_struct *p;

list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

Here p−>run_list is declared as 'struct list_head run_list' inside task_struct structure and serves as anchor to the list. Removing an element from the list and adding (to head or tail of the list) is done by list_del()/list_add()/list_add_tail() macros. The examples below are adding and removing a task from runqueue:

```
static inline void del_from_runqueue(struct task_struct * p)
{
        nr_running−−;
        list_del(&p−>run_list);
        p−>run_list.next = NULL;
}

static inline void add_to_runqueue(struct task_struct * p)
{
        list_add(&p−>run_list, &runqueue_head);
        nr_running++;
}

static inline void move_last_runqueue(struct task_struct * p)
{
        list_del(&p−>run_list);
        list_add_tail(&p−>run_list, &runqueue_head);
}

static inline void move_first_runqueue(struct task_struct * p)
{
        list_del(&p−>run_list);
        list_add(&p−>run_list, &runqueue_head);
}
```

2.5 Wait Queues

When a process requests the kernel to do something which is currently impossible but that may become possible later, the process is put to sleep and is woken up when the request is more likely to be satisfied. One of the kernel mechanisms used for this is called a 'wait queue'.

Linux implementation allows wake−on semantics using TASK_EXCLUSIVE flag. With waitqueues you can either use a well−known queue and then simply sleep_on/sleep_on_timeout/interruptible_sleep_on/interruptible_sleep_on_timeout or you can define your own waitqueue and use add/remove_wait_queue to add and remove yourself from it and also wake_up/wake_up_interruptible to wake up when needed.

An example of the first usage of waiteueus is interaction between page allocator mm/page_alloc.c:__alloc_pages() using the well−known queue kswapd_wait declared in mm/vmscan.c and on which kswap kernel daemon is sleeping in mm/vmscan.c:kswap() and is woken up when page allocator needs to free up some pages.

An example of autonomous waitqueue usage is interaction between user process requesting data via read(2) system call and kernel running in the interrupt context to supply the data. An interrupt handler might look like (simplified drivers/char/rtc_interrupt()):

```
static DECLARE_WAIT_QUEUE_HEAD(rtc_wait);

void rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        spin_lock(&rtc_lock);
        rtc_irq_data = CMOS_READ(RTC_INTR_FLAGS);
        spin_unlock(&rtc_lock);
        wake_up_interruptible(&rtc_wait);
}
```

so, the interrupt handler obtains the data by reading from some device−specific io port (CMOS_READ() macro turns into a couple outb/inb) and then wakes up whoever is sleeping on the rtc_wait wait queue.

Now, the read(2) system call could be implemented as:

```
ssize_t rtc_read(struct file file, char *buf, size_t count, loff_t *ppos)
{
        DECLARE_WAITQUEUE(wait, current);
        unsigned long data;
        ssize_t retval;

        add_wait_queue(&rtc_wait, &wait);
        current−>state = TASK_INTERRUPTIBLE;
        do {
                spin_lock_irq(&rtc_lock);
                data = rtc_irq_data;
                rtc_irq_data = 0;
                spin_unlock_irq(&rtc_lock);

                if (data != 0)
                        break;

                if (file−>f_flags & O_NONBLOCK) {
                        retval = −EAGAIN;
                        goto out;
                }
                if (signal_pending(current)) {
                        retval = −ERESTARTSYS;
                        goto out;
                }
                schedule();
        } while(1);
        retval = put_user(data, (unsigned long *)buf);
        if (!retval)
                retval = sizeof(unsigned long);

out:
        current->state = TASK_RUNNING;
        remove_wait_queue(&rtc_wait, &wait);
        return retval;
}
```

What happens in rtc_read() is this:

1. We declare a wait queue element pointing to current process context
2. We add this element to the rtc_wait waitqueue
3. We mark current context as TASK_INTERRUPTIBLE which means it will not be rescheduled after the next time it sleeps
4. We check if there is no data available, if there is we break out, copy data to user buffer, mark ourselves as TASK_RUNNING, remove from the wait queue and return
5. If there is no data yet we check if user specified non−blocking io and if so we fail with EAGAIN (which is the same as EWOULDBLOCK)
6. We also check if a signal is pending and if so inform the "higher layers" to restart the system call if necessary. By "if necessary" I meant the details of signal disposition as specified in sigaction(2) system call
7. Then we "switch out", i.e. fall asleep. until woken up by the interrupt handler. If we didn't mark ourselves as TASK_INTERRUPTIBLE then the scheduler could schedule as sooner than when the data is available and cause unneeded processing

It is also worth pointing out that using wait queues it is rather easy to implement poll(2) system call:

```
static unsigned int rtc_poll(struct file *file, poll_table *wait)
{
        unsigned long l;

        poll_wait(file, &rtc_wait, wait);

        spin_lock_irq(&rtc_lock);
        l = rtc_irq_data;
        spin_unlock_irq(&rtc_lock);

        if (l != 0)
                return POLLIN | POLLRDNORM;
        return 0;
}
```

All the work is done by device−independent function poll_wait() which does the necessary waitqueue manipulations all we need is to point it to the waitqueue which is woken up by our device−specific interrupt handler.

2.6 Kernel Timers

Now let us turn our attention to kernel timers. Kernel timers are used to dispatch execution of a particular function (called 'timer handler') at a specified time in the future. The main data structure is 'struct timer_list' declared in include/linux/timer.h:

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
        volatile int running;
};
```

The 'list' field is for linking into the internal list, protected by the timerlist_lock spinlock. The 'expires' field is the value of jiffies when the 'function' handler should be invoked with 'data' passed as a parameter. The 'running' field is used on SMP to test if the timer handler is currently running on another cpu.

The functions add_timer() and del_timer() add and remove a given timer to the list. When a timer expires it is removed automatically. Before a timer is used it must be initialised by means of init_timer() function. And before it is added, the fields 'function' and 'expires' must be set.

2.7 Bottom Halves

Sometimes it is reasonable to split the amount of work to be performed inside an interrupt handler into immediate (e.g. acknowledging the interrupt, updating the stats etc.) and that which can be postponed until later, when interrupts are enabled (e.g. to do some postprocessing on data, wake up processes waiting for this data etc.).

Bottom halves are the oldest mechanism for deferred execution of kernel tasks and have been available since Linux 1.x. In Linux2.0 a new mechanism was added called 'task queues' which will be the subject of next section.

Bottom halves are serialized by a global_bh_lock spinlock, i.e. there can only be one bottom half running on any cpu at a time. However, when attempting to execute the handler, if global_bh_lock is not available, the bottom half is marked (i.e. scheduled) for execution − so processing can continue, as opposed to a busy loop on global_bh_lock.

There can only be 32 bottom halves registered in total. The functions required to manipulate bottom halves are as follows (all exported to modules):

- void init_bh(int nr, void (*routine)(void)), installs a bottom half handler pointed to by 'routine'

argument into the slot 'nr'. The slot ought to be enumerated in include/linux/interrupt.h in the form XXXX_BH, e.g. TIMER_BH or TQUEUE_BH. Typically, subsystem's initialisation routine (init_module() for modules) installs the required bottom half using this function
- void remove_bh(int nr), does the opposite of init_bh(), i.e. de−installs bottom half installed at slot 'nr'. There is no error checking performed there, so, for example remove_bh(32) will panic/oops the system. Typically, subsystem's cleanup (cleanup_module() for modules) uses this function to free up the slot that can later be reused by some other subsystem. (TODO: wouldn't it be nice to have /proc/bottom_halves that lists all registered bottom halves on the system? That means global_bh_lock must be made read/write, obviously)
- void mark_bh(int nr), mark this bottom half for execution. Typically, an interrupt handler will mark its bottom half (hence the name!) for execution at a "safer time".

Bottom halves are globally locked tasklets so the question "when are bottom half handlers executed?" is really "when are tasklets executed?". And the answer is − in two places, a) on each schedule() and b) on each interrupt/syscall return path in entry.S. (TODO: therefore, the schedule() case is really boring − it like adding yet another very very slow interrupt, why not get rid of handle_softirq label from schedule() altogether?)

2.8 Task Queues

Task queues can be though of as dynamic extension to old bottom halves. In fact, in the source code they are sometimes referred to as "new" bottom halves. More specifically, the old bottom halves discussed in previous section have these limitations:

1. There are only a fixed number (32) of them
2. Each bottom half can only be associated with one handler function
3. Bottom halves are consumed with a spinlock held so they cannot block

So, with task queues, arbitrary number of functions can be chained and processed one after another at a later time. One create a new task queue using DECLARE_TASK_QUEUE() macro and queues a task onto it using queue_task() function. The task queue then can be processed using run_task_queue() function. Instead of creating your own task queue (and having to consume it manually) you can use one of the Linux's predefined task queues which are consumed at well−known points:

1. **tq_timer** − timer task queue, run on each timer interrupt and when releasing tty device (closing or releasing a half−opened terminal device). Since the timer handler runs in the interrupt context the tq_timer tasks also run in interrupt context and thus cannot block
2. **tq_scheduler** − scheduler task, consumed by the scheduler (and also when closing tty devices, like tq_timer). Since the scheduler executed in the context of the process being re−scheduled, the tq_scheduler tasks can do anything they like, i.e. block, use process context data (but why would they want to) etc
3. **tq_immediate** − is really a bottom half IMMEDIATE_BH, so drivers can queue_task(task, &tq_immediate) and then mark_bh(IMMEDIATE_BH) to be consumed in the interrupt context
4. **tq_disk** − used by low level block device access (and RAID) to start the actual requests. This task queue is exported to modules but shouldn't be used except for special purpose it was designed for

Unless the driver uses its own task queues it does not need to call run_tasks_queues() to process the queue, except under circumstances explained below.

The reason tq_timer/tq_schduler task queues are consumed not only in the usual places but elsewhere (closing tty device is but one example) becomes clear if one remembers that the driver can schedule tasks on the queue that only makes sense while a particular instance of the device is still valid − which usually means until the application closes it. So, the driver may need to call run_task_queue() to flush the tasks it (and anyone else) has put on the queue, because allowing them to run at a later time may make no sense − i.e. the relevant data structures may have been freed/reused by a different instance. This is the reason you see run_task_queue() on tq_timer and tq_scheduler in places other than timer interrup and schedule() respectively.

2.9 Tasklets

Not yet, will be in future revision.

2.10 Softirqs

Not yet, will be in future revision.

2.11 How System Calls Are Implemented on i386 Architecture?

There are two mechanisms under Linux for implementing system calls:

- lcall7/lcall27 call gates
- int 0x80 software interrupt

Native Linux programs use int 0x80 whilst the binaries from foreign flavours of UNIX (Solaris, UnixWare 7 etc.) use lcall7 mechanism. The name 'lcall7' is historically misleading because it covers also lcall27 (e.g. Solaris/x86) but the handler function is called lcall7_func.

When the system boots the function arch/i386/kernel/traps.c:trap_init() is called which sets up the IDT to point vector 0x80 (of type 15, dpl 3) to the address of system_call entry from arch/i386/kernel/entry.S.

When application makes a system call, the arguments are passed via registers and the application executes 'int 0x80' instruction. This causes trap into kernel mode and processor jumps to system_call entry point in entry.S. What this does is:

1. Saves registers
2. Sets %ds and %es to KERNEL_DS, so that all data (and extra segment) references are made in kernel address space
3. If the value of %eax is greater than NR_syscalls (currently 256) then fail with ENOSYS error
4. If the task is being ptraced (tsk−>ptrace & PF_TRACESYS) do special processing. This is to support programs like strace (analogue of SVR4 truss(1)) or debuggers
5. Call sys_call_table+4*(syscall_number from %eax). This table is initialised in the same file (arch/i386/kernel/entry.S) to point to individual system call handlers which under Linux are (usually) prefixed with sys_, e.g. sys_open, sys_exit etc. These C system call handlers will find their arguments on the stack where SAVE_ALL stored them
6. Enter 'system call return path'. This is a separate label because it is used not only by int 0x80 but also by lcall7, lcall27. This is concerned with handling tasklets (including bottom halves), checking if a schedule() is needed (tsk−>need_resched != 0), checking if there are signals pending and if so handling them

Linux supports up to 6 arguments for system calls. They are passed in %ebx, %ecx, %edx, %esi, %edi (and %ebp used temporarily, see _syscall6() in asm−i386/unistd.h) and the system call number is passed via %eax.

2.12 Atomic Operations

There are two types of atomic operations − bitmaps and atomic_t. Bitmaps are very convenient for maintaining a concept of "allocated" or "free" units from some large collection where each unit is identified by some number, for example free inodes or free blocks. They are also widely use for simple locking for example to provide exclusive access to open a device, e.g. in arch/i386/kernel/microcode.c:

```
/*
 *  Bits in microcode_status. (31 bits of room for future expansion)
 */
#define MICROCODE_IS_OPEN       0       /* set if device is in use */

static unsigned long microcode_status;
```

There is no need to initialise microcode_status to 0 as BSS is zero−cleared under Linux explicitly.

```
/*
 * We enforce only one user at a time here with open/close.
 */
static int microcode_open(struct inode *inode, struct file *file)
{
        if (!capable(CAP_SYS_RAWIO))
                return −EPERM;

        /* one at a time, please */
        if (test_and_set_bit(MICROCODE_IS_OPEN, &microcode_status))
```

```
                return −EBUSY;

        MOD_INC_USE_COUNT;
        return 0;
}
```

The operations on bitmaps are:

- **void set_bit(int nr, volatilde void *addr)** − set bit 'nr' in the bitmap pointed to by 'addr'
- **void clear_bit(int nr, volatilde void *addr)** − clear bit 'nr' in the bitmap pointed to by 'addr'
- **void change_bit(int nr, volatile void *addr)** − toggle bit 'nr' (if set clear, if clear set) in the bitmap pointed to by 'addr'
- **int test_and_set_bit(int nr, volatile void *addr)** − atomically set the bit 'nr' and return the old bit value
- **int test_and_clear_bit(int nr, volatile void *addr)** − atomically clear the bit 'nr' and return the old bit value
- **int test_and_change_bit(int nr, volatile void *addr)** − atomically clear the bit 'nr' and return the old bit value

(TODO: why 'volatile' in the above declarations?)

These operations use LOCK_PREFIX which on SMP evaluates to bus lock instruction prefix and to nothing on UP. This guarantees atomicity of access in SMP environment.

Sometimes bit manipulations are not convenient but instead we need to perform arithmetic operations − add, subtract, increment decrement. The typical cases are reference counts (e.g. for inodes). This facility is provided by the atomic_t data type and the following operations:

- **atomic_read(&v)** − read the value of atomic_t variable v
- **atomic_set(&v, i)** − set the value of atomic_t variable v to integer i
- **void atomic_add(int i, volatile atomic_t *v)** − add integer 'i' to the value of atomic variable pointed to by 'v'
- **void atomic_sub(int i, volatile atomic_t *v)** − subtract integer 'i' from the value of atomic variable pointed to by 'v'
- **int atomic_sub_and_test(int i, volatile atomic_t *v)** − subtract integer 'i' from the value of atomic variable pointed to by 'v' and returns 1 if the new value is 0 and returns 0 in all other cases
- **void atomic_inc(volatile atomic_t *v)** − increment the value by 1
- **void atomic_dec(volatile atomic_t *v)** − decrement the value by 1
- **int atomic_dec_and_test(volatile atomic_t *v)** − decrement the value and return 1 if the new value is 0 and return 0 in all other cases
- **int atomic_inc_and_test(volatile atomic_t *v)** − increment the value and return 1 if the new value is 0 and return 0 in all other cases
- **int atomic_add_negative(int i, volatile atomic_t *v)** − add the value of 'i' to 'v' and return 1 if the result is negative. Return 0 if the result is greater than or equal to 0. This operation is used for implementing semaphores

2.13 Spinlocks, Read−write Spinlocks and Big−Reader Spinlocks

Since the early days of Linux support (early 90s, this century), the developers were faced with the classical problem of solving the problem of accessing shared data between different types of context (user process vs interrupt) and different instances of the same context from multiple cpus.

SMP support was added to Linux 1.3.42 on 15 Nov 1995 (the original patch was made to 1.3.37 in October the same year).

If the critical region of code may be executed by either process context and interrupt context, then the way to protect it using cli/sti instructions on UP is:

```
unsigned long flags;

save_flags(flags);
cli();
/* critical code */
restore_flags(flags);
```

While this is ok on UP, it obviously is of no use on SMP because the same code sequence may be executed simultaneously on another cpu and so cli will provide protection against races with interrupt context on each cpu, it will provide no protection against races between contexts running on different cpus. This is where spinlocks are useful for.

There are three types of spinlocks − vanilla (basic), read−write and big−reader spinlocks. Read−write spinlocks should be used when there is a natural tendency of 'many readers and few writers'. Example of this is access to the list of registered filesystems − see fs/super.c. The list is guarded by read−write spinlock file_systems_lock because one needs exclusive access only when registering/unregistering a filesystem but any process can read the file /proc/filesystems of use sysfs(2) system call to force a read−only scan of the file_systems list. This makes it sensible to use read−write spinlocks. With read−write spinlocks, one can have multiple readers at a time but only one writer and there can be no readers while there is a writer. Btw, it would be nice if new readers would not get a lock while there is a writer trying to get a lock, i.e. if Linux could correctly deal with the issue of potential writer starvation by multiple readers. This would mean that readers must be blocked while there is a writer attempting to get the lock. This is not currently the case and it is not obvious whether this should be fixed − the argument to the contrary is − readers usually take the lock for a very short time so should they really be starved while the writer takes the lock for potentially longer periods?

Big−reader spinlocks are a form of read−write spinlocks heavily optimised for very light read access with the penalty for writes. There is a limited number of big−reader spinlocks − currently only two exist, of which one is used only on sparc64 (global irq) and the other is used for networking. In all other cases where the access pattern does not fit into any of these two scenarios one should use basic spinlocks. You cannot block while holding any kind of spinlock.

Spinlocks come in three flavours: plain, _irq() and _bh().

1. Plain spin_lock()/spin_unlock() − if you know the interrupts are always disabled or if you do not race with interrupt context (e.g. from within interrupt handler) then you can use this one. It does not touch interrupt state on the current cpu
2. spin_lock_irq()/spin_unlock_irq() − if you know that interrupts are always enabled then you can use this version which simply disables and re−enables interrupts on the current cpu. For example, rtc_read() uses spin_lock_irq(&rtc_lock) whilst rtc_interrupt() uses spin_lock(&rtc_lock) because inside interrupt handler interrupts are always disabled and inside read() method they are always enabled rtc_read() uses spin_lock_irq() and not the more generic spin_lock_irqsave() because on entry to any system call interrupts are always enabled.
3. spin_lock_irqsave()/spin_unlock_irqrestore() − the strongest form, to be used when the interrupt state is not known, but only if interrupts matter at all, i.e. there is no point in using it we our interrupt handlers don't execute any critical code

The reason you cannot use plain spin_lock() if you race against interrupt handlers is because if you take it and then interrupt comes in on the same cpu − it will busy wait for the lock forever because the lock holder was interrupted and will not continue until the interrupt handler returns.

The most common usage of a spinlock is to access a data structure shared between user process context and interrupt handlers:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

my_ioctl()
{
        unsigned long flags;
        spin_lock_irq(&my_lock, flags);
        /* critical section */
        spin_unlock_irq(&my_lock, flags);
}

my_irq_handler()
{
        spin_lock(&lock);
        /* critical section */
        spin_unlock(&lock);
}
```

There are a couple of things to note about this example:

1. The process context, represented here as a typical driver method − ioctl() (arguments and return values omitted for clarity), must use spin_lock_irq() because it knows that interrupts are always enabled while executing the device ioctl() method
2. Interrupt context, represented here by my_irq_handler() (again arguments omitted for clarity) can use plain spin_lock() form because interrupts are disabled inside interrupt handler

2.14 Semaphores and read/write Semaphores

Sometimes while accessing a shared data structure one must perform operations that can block, for example to copy data to userspace. The locking primitive available for such scenarios under Linux is called a semaphore. There are two types of semaphores − basic and read−write semaphores. Depending on the initial value of the semaphore, they can be used for eithe mutual exclusion (initial value of 1) or to provide more sophisticated type of access.

Read−write semaphores differ from basic semaphores in the same way as read−write spinlocks differ from basic spinlocks, i.e. one can have multiple readers at a time but only one writer and there be no readers while there are writers − i.e. the writer blocks all readers and new readers block while a writer is waiting.

Also, basic semaphores can be interruptible − just use the operations down_interruptible()/up() instead of the plain down()/up() and check the value returned from down_interruptible() − if it is non−0 the operation was interrupted.

Using semaphore for mutual exclusion is ideal in situation where critical code section may call by reference unknown functions registered by other subsystems/modules, i.e. the caller cannot know apriori whether the function blocks or not.

A simple example of semaphore usage is in kernel/sys.c, implementation of gethostname(2)/sethostname(2) system calls.

```
asmlinkage long sys_sethostname(char *name, int len)
{
        int errno;

        if (!capable(CAP_SYS_ADMIN))
                return −EPERM;
        if (len < 0 || len > __NEW_UTS_LEN)
                return −EINVAL;
        down_write(&uts_sem);
        errno = −EFAULT;
        if (!copy_from_user(system_utsname.nodename, name, len)) {
                system_utsname.nodename[len] = 0;
                errno = 0;
        }
        up_write(&uts_sem);
        return errno;
}

asmlinkage long sys_gethostname(char *name, int len)
{
        int i, errno;

        if (len < 0)
                return −EINVAL;
        down_read(&uts_sem);
        i = 1 + strlen(system_utsname.nodename);
        if (i > len)
                i = len;
        errno = 0;
        if (copy_to_user(name, system_utsname.nodename, i))
                errno = −EFAULT;
        up_read(&uts_sem);
        return errno;
}
```

The points to note about this example are:

1. The functions may block while copying data from/to userspace in copy_from_user()/copy_to_user(). Therefore they could not use any form of spinlock here
2. The semaphore type chosen is read−write as opposed to basic because there may be lots of concurrent gethostname(2) requests which need not be mutually exclusive.

Although Linux implementation of semaphores and read−write semaphores is very sophisticated, there are possible scenarios one can think of which are not yet implemented, for example there is no concept of interruptible read−write semaphores. This is obviously because there are no real−world situations which require these exotic flavours of the primitives.

2.15 Kernel Support for Loading Modules

Linux is a monolithic operating system and despite all the modern hype about some "advantages" offered by operating systems based on micro−kernel design, the truth remains (quoting Linus Torvalds himself):

```
... message passing as the fundamental operation of the OS is
just an exercise in computer science masturbation. It may
feel good, but you don't actually get anything DONE.
```

Therefore, Linux is and will always be based on the monolithic design, which means that all subsystems run in the same privileged mode and share the same address space; communication between them is achieved by the usual C function call means.

However, although separating kernel functionality into separate "processes" as is done in micro−kernels is definitely a bad idea, separating it into dynamically loadable on demand kernel modules is desirable in some circumstances (e.g. on machines with low memory or for installation kernels which could otherwise contain ISA auto−probing device drivers that are mutually exclusive). The decision whether to include support for loadable modules is made at compilation time and is determined by the CONFIG_MODULES option. Support for auto−loading modules via request_module() mechanism is a separate compilation option − CONFIG_KMOD.

The following functionality can be implemented as loadable modules under Linux:

1. Character and block device drivers, including misc device drivers

2. Terminal line disciplines
3. Virtual (regular) files in /proc and in devfs (e.g. /dev/cpu/microcode vs /dev/misc/microcode)
4. Binary file formats (e.g. ELF, aout etc.)
5. Execution domains (e.g. Linux, UnixWare7, Solaris etc.)
6. Filesystems
7. System V IPC

There a few things that cannot be implemented as modules under Linux (probably because it makes no sense for them to be modularized):

1. Scheduling algorithms
2. VM policies
3. Buffer cache, page cache and other caches

Linux provides several system calls to assist in loading modules:

1. caddr_t create_module(const char *name, size_t size) − allocates 'size' bytes using vmalloc() and maps a module structure at the beginning thereof. This new module is then linked into the list headed by module_list. Only a process with CAP_SYS_MODULE can invoke this system call, others will get EPERM returned
2. long init_module(const char *name, struct module *image) − loads the relocated module image and causes the module's initialisation routine to be invoked. Only a process with CAP_SYS_MODULE can invoke this system call, others will get EPERM returned
3. long delete_module(const char *name) − attempts to unload the module. If name == NULL then attempt is made to unload all unused modules
4. long query_module(const char *name, int which, void *buf. size_t bufsize, size_t *ret) − returns information about a module (or about all modules)

The command interface available to users consists of:

- insmod − insert a single module
- modprobe − insert a module including all the other modules it depends on
- rmmod − remove a module
- modinfo − print some information about a module, e.g. author, description, parameters the module accepts etc

Apart from being to load a module manually using either insmod or modprobe it is also possible to have the module inserted automatically by the kernel when a particular functionality is required. The kernel interface for this is the function called request_module(name) which is exported to modules so modules can load other modules as well. The request_module(name) internally creates a kernel thread which execs the userspace command "modprobe −s −k module_name" using the standard exec_usermodehelper() kernel interface (which is also exported to modules). The function returns 0 on success, however it is usually not worth checking the return code from request_module(). Instead, the programming idiom is:

```
if (check_some_feature() == NULL)
    request_module(module);
```

```
if (check_some_feature() == NULL)
    return −ENODEV;
```

For example, this is done by fs/block_dev.c:get_blkfops() to load a module "block−major−N" when attempt is made to open a block device on a major N. Obviously, there is no such module called "block−major−N" (Linux developers only chose sensible names for their modules) but it is mapped to a proper module name using the file /etc/modules.conf. However, for most well−known major numbers (and other kinds of modules) the modprobe/insmod commands know which real module to load without needing an explicit alias statement in /etc/modules.conf.

A good example of loading a module is inside the mount(2) system call. The mount(2) system call accepts the filesystem type as a string which fs/super.c:do_mount() then passes on to fs/super.c:get_fs_type():

```
static struct file_system_type *get_fs_type(const char *name)
{
        struct file_system_type *fs;

        read_lock(&file_systems_lock);
        fs = *(find_filesystem(name));
        if (fs && !try_inc_mod_count(fs−>owner))
                fs = NULL;
        read_unlock(&file_systems_lock);
        if (!fs && (request_module(name) == 0)) {
                read_lock(&file_systems_lock);
                fs = *(find_filesystem(name));
                if (fs && !try_inc_mod_count(fs−>owner))
                        fs = NULL;
                read_unlock(&file_systems_lock);
        }
        return fs;
}
```

A few things to note in this function:

1. First we attempt to find the filesystem with the given name amongst those already registered. This is done under protection of file_systems_lock taken for read (as we are not modifying the list of registered filesystems)
2. If such filesystem is found then we attempt to get a new reference to it by trying to increment its module's hold count. This always returns 1 for statically linked filesystems or for modules not presently being deleted. If try_inc_mod_count() returned 0 then we consider it a failure − i.e. if the module is there but being deleted it is as good as if it was not there at all
3. We drop the file_systems_lock because what we are about to do next (request_module()) is a blocking operation and therefore we can't hold a spinlock over it. Actually, in this specific case, we would have to drop file_systems_lock anyway, even if request_module() was guaranteed to be non−blocking and the module loading was executed in the same context atomically. The reason for this is that module's initialisation will try to call register_filesystem() which will take the same file_systems_lock read−write spinlock for write and we will deadlock

4. If the attempt to load was successful, then we take the file_systems_lock spinlock and try to locate the newly registered filesystem in the list. Note, that this is slightly wrong because it is in principle possible for a bug in modprobe command to cause it to coredump after it successfuly loaded the requested module, in which case request_module() will fail but the new filesystem will be registered and yet get_fs_type() won't find it
5. If the filesystem is found and we are able to get a reference to it we return it. Otherwise we return NULL

When a module is loaded into the kernel it can refer to any symbols that are exported as public by the kernel using EXPORT_SYMBOL() macro or by other currently loaded modules. If the module uses symbols from another module it is marked as depending on that module during dependency recalculation, achieved by running "depmod −a" command on boot (e.g. after installing a new kernel).

Usually, one must match the set of modules with the version of the kernel interfaces they use, which under Linux simply means the "kernel version" as there is no special kernel interface versioning mechanism in general. However, there is a limited functionality called "module versioning" or CONFIG_MODVERSIONS which allows to avoid recompiling modules when switching to a new kernel. What happens here is that the kernel symbol table is treated differently for internal access and for access from modules. The elements of public (i.e. exported) part of the symbol table are built by 32bit checksumming the C declaration. So, in order to resolve a symbol used by a module during loading, the loader must match the full representation of the symbol that includes the checksum and will refuse to load the module. This only happens when both the kernel and the module are compiled with module versioning enabled. If either one of them uses the original symbol names then the loader simply tries to match the kernel version declared by the module and the one exported by the kernel and refuses to load if they differ.

3.Virtual Filesystem (VFS)

3.1 Inode Caches and Interaction with Dcache

In order to support multiple filesystems Linux contains a special kernel interface level called VFS − Virtual Filesystem Switch. This is similar to vnode/vfs interface found in SVR4 derivatives (originally it came from BSD and Sun original implementations).

Linux inode cache is implemented in a single file fs/inode.c which consists of 977 lines of code. It is interesting to note that for the last 5−7 years not many changes were made to it, i.e. one can still recognize some of the code comparing the latest version with, say, 1.3.42.

The structure of Linux inode cache is as follows:

1. A global hashtable inode_hashtable, each inode is hashed by the value of the superblock pointer and 32bit inode number. Inodes without superblock (inode−>i_sb == NULL) are added to a doubly linked list headed by anon_hash_chain instead. Examples of anonymous inodes are sockets created

   by net/socket.c:sock_alloc() by calling fs/inode.c:get_empty_inode()
2. A global type in_use list (inode_in_use) which contains valid inodes with i_count>0, i_nlink>0. Inodes newly allocated by get_empty_inode() and get_new_inode() are added to inode_in_use list
3. A global type unused list (inode_unused) which contains valid inodes with i_count = 0
4. A per−superblock type dirty list (sb−>s_dirty) which contains valid inodes with i_count>0, i_nlink>0 and i_state & I_DIRTY. When inode is marked dirty it is added to the sb−>s_dirty list if it is also hashed. Maintaining a per−superblock dirty list of inodes allows to quickly sync inodes
5. Inode cache proper − a SLAB cache called inode_cachep. As inode objects are allocated and freed, they are taken from and returned to this SLAB cache

The type lists are anchored from inode−>i_list, the hashtable from inode−>i_hash. Each inode can be on a hashtable and one and only one type (in_use, unused or dirty) list.

All these lists are protected by a single spinlock − inode_lock.

Inode cache subsystem is initialised when inode_init() function is called init/main.c:start_kernel(). The function is marked as __init which means its code is thrown away later on. It is passed a single argument − the number of physical pages on the system. This is so that inode cache can configure itself depending on how much memory is available, i.e. create a larger hashtable if there is enough memory.

The only stats information about inode cache is the number of unused inodes, stored in inodes_stat.nr_unused and accessible to user programs via files /proc/sys/fs/inode−nr and /proc/sys/fs/inode−state.

We can examine one of the lists from the gdb running on a live kernel thus:

```
(gdb) printf "%d\n", (unsigned long)(&((struct inode *)0)->i_list)
8
(gdb) p inode_unused
$34 = 0xdfa992a8
(gdb) p (struct list_head)inode_unused
$35 = {next = 0xdfa992a8, prev = 0xdfcdd5a8}
(gdb) p ((struct list_head)inode_unused).prev
$36 = (struct list_head *) 0xdfcdd5a8
(gdb) p (((struct list_head)inode_unused).prev)->prev
$37 = (struct list_head *) 0xdfb5a2e8
(gdb) set $i = (struct inode *)0xdfb5a2e0
(gdb) p $i->i_ino
$38 = 0x3bec7
(gdb) p $i->i_count
$39 = {counter = 0x0}
```

Note that we deducted 8 from the address 0xdfb5a2e8 to obtain the address of the 'struct inode' 0xdfb5a2e0 according to the definition of list_entry() macro from include/linux/list.h.

To understand how inode cache works let us trace a lifetime of an inode of a regular file on ext2 filesystem as it is opened and closed:

```
fd = open("file", O_RDONLY);
close(fd);
```

The open(2) system call is implemented in fs/open.c:sys_open function and the real work is done by fs/open.c:filp_open() function which is split into two parts:

1. open_namei() − fills in nameidata structure containing the dentry and vfsmount structures
2. dentry_open() − given a dentry and vfsmount it allocates a new 'struct file' and links them together, as well as invoking filesystem specific f_op−>open() method which was set in inode−>i_fop when inode was read in open_namei() (which provided inode via dentry−>d_inode).

The open_namei() function interacts with dentry cache via path_walk() which in turn calls real_lookup() which invokes inode_operations−>lookup() method which is filesystem−specific and its job is to find the entry in the parent directory with the matching name and then do iget(sb, ino) to get the corresponding inode which brings us to the inode cache. When the inode is read in, the dentry is instantiated by means of d_add(dentry, inode). While we are at it, note that for UNIX−style filesystems which have the concept of on−disk inode number, it is the lookup method's job to map its endianness to current cpu format, e.g. if the inode number in raw (fs−specific) dir entry is in little−endian 32 bit format one could do:

```
unsigned long ino = le32_to_cpu(de−>inode);
inode = iget(sb, ino);
d_add(dentry, inode);
```

So, when we open a file we hit iget(sb, ino) which is really iget4(sb, ino, NULL, NULL) which does:

1. Attempts to find an inode with matching superblock and inode number in the hashtable under protection of inode_lock. If inode is found then it's reference count (i_count) is incremented and if and if it was 0 and inode is not dirty then inode is removed from whatever type list (inode−>i_list) it is currently on (it has to be inode_unused list, of course) and inserted into inode_in_use type list and inodes_stat.nr_unused is decremented
2. If inode is currently locked we wait until it is not locked so that iget4() is guaranteed to return not locked inode
3. If inode was not found in the hashtable then it is the first time we encounter this inode so we call get_new_inode() passing it the pointer to the place in the hashtable where it should be inserted to
4. get_new_inode() allocates a new inode from the inode_cachep SLAB cache but this operation can block (GFP_KERNEL allocation) so it must drop the inode_lock spinlock which guards the hashtable. Since it dropped the spinlock it must retry searching the inode in the hashtable and if it is found this time, it returns (after incrementing the reference by __iget) the one found in the hashtable and destroys the newly allocated one. If it is still not found in the hashtable then the new inode we have just allocated is the one to be used and so it is initialised to the required values and the fs−specific sb−>s_op−>read_inode() method is invoked to populate the rest of the inode. This brings us from inode cache back to the filesystem code − remember that we came to the inode cache when filesystem−specific lookup() method invoked iget(). While the s_op−>read_inode() method is

reading the inode from disk the inode is locked (i_state = I_LOCK) and after it returns it is unlocked and all the waiters for it are woken up

Now, let's see what happens when we close this file descriptor. The close(2) system call is implemented in fs/open.c:sys_close() function which calls do_close(fd, 1) which rips (replaces with NULL) the descriptor of the process' file descriptor table and invokes filp_close() function which does most of the work. The interesting things happen in fput() which checks if this was the last reference to the file and if so calls fs/file_table.c:_fput() which calls __fput() which is where interaction with dcache (and therefore with inode cache − remember dcache is a Master of inode cache!) happens. The fs/dcache.c:dput() does dentry_iput() which brings us back to inode cache via iput(inode) so let us understand fs/inode.c:iput(inode):

1. if parameter passed to us is NULL, we do absolutely nothing and return
2. if there is a fs−specific sb−>s_op−>put_inode() method it is invoked now with no spinlocks held (so it can block)
3. inode_lock spinlock is taken and i_count is decremented. If this was NOT the last reference to this inode then we simply checked if there are too many references to it and so i_count can wrap around the 32 bits allocated to it and if so we print a warning and return. Note that we call printk() while holding the inode_lock spinlock − this is fine because printk() can never block so it may be called in absolutely any context (even from interrupt handlers!)
4. if this was the last active reference then some work needs to be done.

The work performed by iput() on the last inode reference is rather complex so we separate it into a list of its own:

1. If i_nlink == 0 (e.g. the file was unlinked while we held it open) then inode is removed from hashtable and from its type list and if there are any data pages held in page cache for this inode, they are removed by means of truncate_all_inode_pages(&inode−>i_data). Then filesystem−specific s_op−>delete_inode() method is invoked which typically deletes on−disk copy of the inode. If there is no s_op−>delete_inode() method registered by the filesystem (e.g. ramfs) then we call clear_inode(inode) which invokes s_op−>clear_inode() if registered and if inode corresponds to a block device the device's reference count is dropped by bdput(inode−>i_bdev).
2. if i_nlink != 0 then we check if there are other inodes in the same hash bucket and if there is none, then if inode is not dirty we delete it from its type list and add it to inode_unused list incrementing inodes_stat.nr_unused. If there are inodes in the same hashbucket then we delete it from the type list and add to inode_unused list. If this was anonymous inode (NetApp .snapshot) then we delete it from the type list and clear/destroy it completely

3.2 Filesystem Registration/Unregistration

Linux kernel provides a mechanism for new filesystems to be written with minimum effort. The historical reasons for this are:

1. In the world where people still use non−Linux operating systems to protect their investment in legacy software Linux had to provide interoperability by supporting a great multitude of different

filesystems − most of which would not deserve to exist on their own but only for compatibility with existing non−Linux operating systems
2. The interface for filesystem writers had to be very simple so that people could try to reverse engineer existing proprietary filesystems by writing read−only versions of them. Therefore Linux VFS makes it very easy to implement read−only filesystems − 95% of the work is to finish them by adding full write−support. As a concrete example, I wrote read−only BFS filesystem for Linux in about 10 hours but it took several weeks to complete it to have full write support (and even today some purists claim that it is not complete because "it doesn't have compactification support")
3. All Linux filesystems can be implemented as modules so VFS interface is exported

Let us consider the steps required to implement a filesystem under Linux. The code implementing a filesystem can be either a dynamically loadable module or statically linked into the kernel and the way it is done under Linux is very transparent. All that is needed is to fill in a 'struct file_system_type' structure and register it with the VFS using register_filesystem() function as in the following example from fs/bfs/inode.c:

```
#include <linux/module.h>
#include <linux/init.h>

static struct super_block *bfs_read_super(struct super_block *, void *, int);

static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);

static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}

static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}

module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

These macros ensure that for modules the functions init_bfs_fs() and exit_bfs_fs() turn into init_module() and cleanup_module() respectively and for statically linked objects the exit_bfs_fs() code vanishes as it is unnecessary.

The 'struct file_system_type' is declared in include/linux/fs.h:

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
        struct vfsmount *kern_mnt; /* For kernel mount, if it's FS_SINGLE fs */
        struct file_system_type * next;
};
```

The fields thereof are explained thus:

- **name** − human readable name, appears in /proc/filesystems file and is used as a key to find filesystem by name (type of mount(2)) and to refuse to register a different filesystem under the name of the one already registered − so there can (obviously) be only one filesystem with a given name. For modules, name points to module's address spaces and not copied − this means cat /proc/filesystems can oops if the module was unloaded but filesystem is still registered
- **fs_flags** − one or more (ORed) of the flags: FS_REQUIRES_DEV for filesystems that can only be mounted on a block device, FS_SINGLE for filesystems that can have only one superblock, FS_NOMOUNT for filesystems that cannot be mounted from userspace by means of mount(2) system call − they can however be mounted internally using kern_mount() interface, e.g. pipefs
- **read_super** − a pointer to the function that reads the super block during mount operation. This function is required − if it is not provided, mount operation (whether from userspace or inkernel) will always fail except in FS_SINGLE case where it will Oops in get_sb_single() trying to dereference a NULL pointer in fs_type−>kern_mnt−>mnt_sb with (fs_type−>kern_mnt = NULL)
- **owner** − pointer to the module that implements this filesystem. If the filesystem is statically linked into the kernel then this is NULL. You don't need to set this manually as the macro THIS_MODULE does the right thing automatically
- **kern_mnt** − for FS_SINGLE filesystems only. This is set by kern_mount(). (TODO: kern_mount() should refuse to mount filesystems if FS_SINGLE isnot set)
- **next** − linkage into singly−linked list headed by file_systems (see fs/super.c). The list is protected by the file_systems_lock read−write spinlock and functions register/unregister_filesystem() modify it by linking and unlinking the entry from the list

The job of read_super function is to fill in the fields of the superblock, allocate root inode and initialise any fs−private information associated with this mounted instance of the filesystem. So, typically the read_super() would do:

1. Read the superblock from the device specified via sb−>s_dev argument using buffer cache bread() function. If it anticipates to read a few more subsequent metadata blocks immediately then it makes sense to use breada() to schedule reading extra blocks asynchronously
2. Verify that superblock contains the valid magic number and overall "looks" sane
3. Initialise sb−>s_op to point to 'struct super_block_operations' structure. This structure contains filesystem−specific functions implementing operations like "read inode", "delete inode" etc
4. Allocate root inode and root dentry using d_alloc_root()
5. If the filesystem is not mounted read−only then set sb−>s_dirt = 1 and mark the buffer containing superblock dirty (TODO: why do we do this? I did it in BFS because MINIX did it...)

3.3 File Descriptor Management

Under Linux there are several levels of indirection between user file descriptor and the kernel inode structure. When a process makes open(2) system call, the kernel returns a small non−negative integer which can be used for subsequent io operations on this file. This integer is an index into an array of pointers to 'struct file'. Each file structure points to a dentry via file−>f_dentry. And each dentry points to an inode via dentry−>d_inode.

Each task contains a field tsk−>files which is a pointer to 'struct files_struct' defined in include/linux/sched.h:

```
/*
 * Open file table structure
 */
struct files_struct {
        atomic_t count;
        rwlock_t file_lock;
        int max_fds;
        int max_fdset;
        int next_fd;
        struct file ** fd;      /* current fd array */
        fd_set *close_on_exec;
        fd_set *open_fds;
        fd_set close_on_exec_init;
        fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];
};
```

The file−>count is a reference count, incremented by get_file() (usually called by fget()) and decremented by fput() and by put_filp(). The difference between fput() and put_filp() is that fput() does more work usually needed for regular files, such as releasing flock locks, releasing dentry etc while put_filp() is only manipulating file table structures, i.e. decrements the count, removes the file from the anon_list and adds it to the free_list, under protection of files_lock spinlock.

The tsk−>files can be shared between parent and child if the child thread was created using clone() system call with CLONE_FILES set in the clone flags argument. This can be seen in kernel/fork.c:copy_files() (called by do_fork()) which only increments the file−>count if CLONE_FILES is set instead of the usual copying file descriptor table in time−honoured tradition of classical UNIX fork(2).

When a file is opened the file structure allocated for it is installed into current−>files−>fd[fd] slot and a 'fd' bit is set in the bitmap current−>files−>open_fds. All this is done under the write protection of current−>files−>file_lock read−write spinlock. When the descriptor is closed a 'fd' bit is cleared in current−>files−>open_fds and current−>files−>next_fd is set equal to 'fd' as a hint for finding the first unused descriptor next time this process wants to open a file.

3.4 File Structure Management

The file structure is declared in include/linux/fs.h:

```
struct fown_struct {
        int pid;                /* pid or −pgrp where SIGIO should be sent */
        uid_t uid, euid;        /* uid/euid of process setting the owner */
        int signum;             /* posix.1b rt signal to be delivered on IO */
};

struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;

        unsigned long           f_version;

        /* needed for tty driver, and maybe others */
        void                    *private_data;
};
```

Let us look at the various fields of 'struct file':

1. **f_list** − this field links file structure on one (and only one) of the lists: a) sb−>s_files list of all open files on this filesystem, if the corresponding inode is not anonymous, then dentry_open() (called by filp_open() links the file into this list; b) fs/file_table.c:free_list containing unused file structures; c) fs/file_table.c:anon_list, when a new file structure is created by get_empty_filp() it is placed on this list. All these lists are protected by files_lock spinlock
2. **f_dentry** − the dentry corresponding to this file. The dentry is created at nameidata lookup time by open_namei() (or rather path_walk() which it calls) but the actual file−>f_dentry field is set by dentry_open() to contain the dentry thus found
3. **f_vfsmnt** − the pointer to vfsmount structure of the filesystem containing the file. This is set by dentry_open() but is found as part of nameidata lookup by open_namei() (or rather path_init() which it calls)
4. **f_op** − the pointer to file_operations which contains various methods that can be invoked on the file. This is copied from inode−>i_fop which is placed there by filesystem−specific s_op−>read_inode() method during nameidata lookup. We will look at file_operations methods in detail later on in this section
5. **f_count** − reference count manipulated by get_file/put_filp/fput
6. **f_flags** − O_XXX flags from open(2) system call copied there (with slight modifications by filp_open) by dentry_open and after clearing O_CREAT, O_EXCL, O_NOCTTY, O_TRUNC − there is no point in storing these flags permanently since they cannot be modified by F_SETFL (or queried by F_GETFL) fcntl(2) calls
7. **f_mode** − a combination of userspace flags and mode, set by dentry_open(). The point of the conversion is to store read and write access in separate bits so one could do easy checks like (f_mode & FMODE_WRITE) and (f_mode & FMODE_READ)
8. **f_pos** − a current file position for next read or write to the file. Under i386 it is of type long long, i.e.

a 64bit value

9. **f_reada, f_ramax, f_raend, f_ralen, f_rawin** − to support readahead − too complex to be discussed by mortals ;)
10. **f_owner** − owner of file io to receive asynchronous io notifications via SIGIO mechanism (see fs/fcntl.c:kill_fasync())
11. **f_uid, f_gid** − set to user id and group id of the process that opened the file, when the file structure is created in get_empty_filp(). If the file is a socket, used by ipv4 netfilter
12. **f_error** − used by NFS client to return write errors. It is set in fs/nfs/file.c and checked in mm/filemap.c:generic_file_write()
13. **f_version** − versioning mechanism for invalidating caches, incremented (using global 'event') whenever f_pos changes
14. **private_data** − private per−file data which can be used by filesystems (e.g. coda stores credentials here) or by device drivers. Device drivers (in the presence of devfs) could use this field to differentiate between multiple instances instead of the classical minor number encoded in file−>f_dentry−>d_inode−>i_rdev

Now let us look at file_operations structure which contains the methods that can be invoked on files. Let us recall that it is copied from inode−>i_fop where it is set by s_op−>read_inode() method. It is declared in include/linux/fs.h:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

1. **owner** − a pointer to the module that owns the subsystem in question. Only drivers need to set it to THIS_MODULE, filesystems can happily ignore it because their module counts are controlled at mount/umount time whilst the drivers need to control it at open/release time
2. **llseek** − implements the lseek(2) system call. Usually it is omitted and fs/read_write.c:default_llseek() is used which does the right thing (TODO: force all those who set it to NULL currently to use default_llseek − that way we save an if() in llseek())
3. **read** − implements read(2) system call. Filesystems can use mm/filemap.c:generic_file_read() for regular files and fs/read_write.c:generic_read_dir() (which simply returns −EISDIR) for directories here
4. **write** − implements write(2) system call. Filesystems can use mm/filemap.c:generic_file_write() for

regular files and ignore it for directories here

5. **readdir** − used by filesystems. Ignored for regular files and implements readdir(2) and getdents(2) system calls for directories
6. **poll** − implements poll(2) and select(2) system calls
7. **ioctl** − implements driver or filesystem−specific ioctls. Note that generic file ioctls like FIBMAP, FIGETBSZ, FIONREAD are implemented by higher levels so they never read f_op−>ioctl() method
8. **mmap** − implements mmap system call. Filesystems can use generic_file_mmap here for regular files and ignore it on directories
9. **open** − called at open(2) time by dentry_open(). Filesystems rarely use this, e.g. coda tries to cache the file locally at open time
10. **flush** − called at each close(2) of this file, not necessarily the last one (see release() method below). The only filesystem that uses this is NFS client to flush all dirty pages. Note that this can return an error which will be passed back to userspace which made the close(2) system call
11. **release** − called at the last close(2) of this file, i.e. when file−>f_count reaches 0. Although defined as returning int, the return value is ignored by VFS (see fs/file_table.c:__fput())
12. **fsync** − maps directly to fsync(2)/fdatasync(2) system calls, with the last argument specifying whether it is fsync or fdatasync. Almost no work is done by VFS around this, except to map file descriptor to a file structure (file = fget(fd)) and down/up inode−>i_sem semaphore. Ext2 filesystem currently ignores the last argument and does exactly the same for fsync(2) and fdatasync(2)
13. **fasync** − this method is called when file−>f_flags & FASYNC changes
14. **lock** − the filesystem−specific portion of the POSIX fcntl() file region locking mechanism. The only bug here is that because it is called before fs−independent portion (posix_lock_file()), if it succeeds but the standard posix lock code fails then it will never be unlocked on fs−dependent level..
15. **readv** − implements readv(2) system call
16. **writev** − implements writev(2) system call

3.5 Superblock and Mountpoint Management

Under Linux, information about mounted filesystems is kept in two separate structures − super_block and vfsmount. The reason for this is that Linux allows to mount the same filesystem (block device) under multiple mount points, which means that the same super_block can correspond to multiple vfsmount structures.

Let us look at struct super_block first, declared in include/linux/fs.h:

```
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        kdev_t                  s_dev;
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_lock;
        unsigned char           s_dirt;
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct dquot_operations *dq_op;
        unsigned long           s_flags;
        unsigned long           s_magic;
        struct dentry           *s_root;
        wait_queue_head_t       s_wait;

        struct list_head        s_dirty;        /* dirty inodes */
        struct list_head        s_files;

        struct block_device     *s_bdev;
        struct list_head        s_mounts;       /* vfsmount(s) of this one */
        struct quota_mount_options s_dquot;     /* Diskquota specific options */

       union {
                struct minix_sb_info    minix_sb;
                struct ext2_sb_info     ext2_sb;
                ..... all filesystems that need sb−private info ...
                void                    *generic_sbp;
        } u;
       /*
         * The next field is for VFS *only*. No filesystems have any business
         * even looking at it. You had been warned.
         */
        struct semaphore s_vfs_rename_sem;      /* Kludge */

        /* The next field is used by knfsd when converting a (inode number based)
         * file handle into a dentry. As it builds a path in the dcache tree from
         * the bottom up, there may for a time be a subpath of dentrys which is not
         * connected to the main tree.  This semaphore ensure that there is only ever
         * one such free path per filesystem.  Note that unconnected files (or other
         * non−directories) are allowed, but not unconnected diretories.
         */
        struct semaphore s_nfsd_free_path_sem;
};
```

The various fields in the super_block structure are:

1. **s_list** − a doubly−linked list of all active superblocks, note I don't say "of all mounted filesystems" because under Linux one can have multiple instances of a mounted filesystem corresponding to a single superblock
2. **s_dev** − for filesystems which require a block to be mounted on, i.e. for FS_REQUIRES_DEV filesystems, this is the i_dev of the block device. For others (called anonymous filesystems) this is an integer MKDEV(UNNAMED_MAJOR, i) where i is the first unset bit in unnamed_dev_in_use array, between 1 and 255 inclusive. See fs/super.c:get_unnamed_dev()/put_unnamed_dev(). It has been suggested many times that anonymous filesystems should not use s_dev field
3. **s_blocksize, s_blocksize_bits** − blocksize and log2(blocksize)
4. **s_lock** − indicates whether superblock is currently locked by lock_super()/unlock_super()
5. **s_dirt** − set when superblock is changed and cleared whenever it is written back to disk
6. **s_type** − pointer to 'struct file_system_type' of the corresponding filesystem. Filesystem's read_super() method doesn't need to set it as VFS fs/super.c:read_super() sets it for you if fs−specific read_super() succeeds and resets to NULL if it fails
7. **s_op** − pointer to super_operations structure which contains fs−specific methods to read/write inodes etc. It is the job of filesystem's read_super() method to initialise s_op correctly
8. **dq_op** − disk quota operations
9. **s_flags** − superblock flags
10. **s_magic** − filesystem's magic number. Used by minix filesystem to differentiate between multiple flavours of itself

11. **s_root** − dentry of the filesystem's root. It is the job of read_super() to read the root inode from the disk and pass it to d_alloc_root() to allocate the dentry and instantiate it. Some filesystems spell "root" other than "/" and so use more generic d_alloc() function to bind the dentry to a name, e.g. pipefs mounts itself on "pipe:" as its own root instead of "/"
12. **s_wait** − waitqueue of processes waiting for superblock to be unlocked
13. **s_dirty** − a list of all dirty inodes. Recall that if inode is dirty (inode−>i_state & I_DIRTY) then it is on superblock−specific dirty list linked via inode−>i_list
14. **s_files** − a list of all open files on this superblock. Useful for deciding whether filesystem can be remounted read−only, see fs/file_table.c:fs_may_remount_ro() which goes through sb−>s_files list and denies remounting if there are files opened for write (file−>f_mode & FMODE_WRITE) or files with pending unlink (inode−>i_nlink == 0)
15. **s_bdev** − for FS_REQUIRES_DEV this points to the block_device structure describing the device the filesystem is mounted on
16. **s_mounts** − a list of all vfsmount structures, one for each mounted instance of this superblock
17. **s_dquot** − more diskquota stuff

The superblock operations are described in the super_operations structure declared in include/linux/fs.h:

```
struct super_operations {
        void (*read_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
};
```

1. **read_inode** − reads the inode from the filesystem. It is only called from fs/inode.c:get_new_inode() from iget4() (and therefore iget()). If filesystem wants to use iget() then read_inode() must be implemented − otherwise get_new_inode() will panic. While inode is being read it is locked (inode−>i_state = I_LOCK). When the function returns all waiters on inode−>i_wait are woken up. The job of filesystem's read_inode() method is to locate the disk block which contains the inode to be read and use buffer cache bread() function to read it in and initialise the various fields of inode structure, for example the inode−>i_op and inode−>i_fop so that VFS level knows what operations can be performed on the inode or corresponding file. Filesystems that don't implement read_inode() are ramfs and pipefs. For example, ramfs has its own inode−generating function ramfs_get_inode() with all the inode operations calling it as needed
2. **write_inode** − write inode back to disk. Similar to read_inode() in that it needs to locate the relevant block on disk and interact with buffer cache by calling mark_buffer_dirty(bh, 0), 0 meaning that the block must be flushed at "normal priority" instead of "superblock priority" which means less frequently. This method is called on dirty inodes (those marked dirty with mark_inode_dirty) when the inode needs to be sync'd either individually or as part of syncing the entire filesystem
3. **put_inode** − called whenever the reference count is decreased
4. **delete_inode** − called whenever both inode−>i_count and inode−>i_nlink reach 0. Filesystem deletes

the on−disk copy of the inode and calls clear_inode() on VFS inode to "terminate it with extreme prejudice"

5. **put_super** − called at the last stages of umount(2) system call to notify the filesystem that any private information held by the filesystem about this instance should be freed. Typically this would brelse() the block containing the superblock and kfree() any bitmaps allocated for free blocks, inodes etc
6. **write_super** − called when superblock needs to be written back to disk. It should find the block containing the superblock (usually kept in sb−private area) and mark_buffer_dirty(bh, 1), 1 signifying that it is an "important" block that should be flushed at higher frequency than the others. It should also clear sb−>s_dirt flag
7. **statfs** − implements fstatfs(2)/statfs(2) system calls. Note that the pointer to 'struct statfs' passed as argument is a kernel pointer, not a user pointer so we don't need to do any io to userspace. If not implemented then statfs(2) will fail with ENOSYS
8. **remount_fs** − called whenever filesystem is being remounted
9. **clear_inode** − called from VFS level clear_inode(). Filesystems that attach private data to inode structure (via generic_ip field) must free it here
10. **umount_begin** − called during forced umount to notify the filesystem beforehand, so that it can do its best to make sure that nothing keeps the filesystem busy. Currently used only by NFS. This has nothing to do with the idea of generic VFS level forced umount support

So, let us look at what happens when we mount a on−disk (FS_REQUIRES_DEV) filesystem. The implementation of the mount(2) system call is in fs/super.c:sys_mount() which is the just a wrapper that copies the options, filesystem type and device name for the do_mount() function which does the real work:

1. Filesystem driver is loaded if needed and its module's reference count is incremented. Note that during mount operation the filesystem module's reference count is incremented twice − once by do_mount() calling get_fs_type() and once by get_sb_dev() calling get_filesystem() if read_super() was successful. The first increment is to prevent module unloading while we are inside read_super() method and the second increment is to indicate that the module is in use by this mounted instance. Obviously do_mount() decrements the count before returning so overall the count only grows by 1 after each mount
2. Since in our case fs_type−>fs_flags & FS_REQUIRES_DEV is true, the superblock is initialised by a call to get_sb_bdev() which obtains the reference to the block device and interacts with the filesystem's read_super() method to fill in the superblock. If all goes well, the super_block structure is initialised and we have an extra reference to the filesystem's module and a reference to the underlying block device
3. A new vfsmount structure is allocated and linked to sb−>s_mounts list and to the global vfsmntlist list. The vfsmount field mnt_instances allows to find all instances mounted on the same superblock as this one. The vfsmount field mnt_list allows to find all instances for all superblocks system−wide. The vfsmount structure's mnt_sb field points to this superblock and mnt_root has a new reference to the sb−>s_root dentry

3.6 Example Virtual Filesystem: pipefs

As a simple example of Linux filesystem that does not require a block device for mounting let us consider pipefs from fs/pipe.c. The filesystem's preamble is rather straightforward and requires little explanation:

```
static DECLARE_FSTYPE(pipe_fs_type, "pipefs", pipefs_read_super,
        FS_NOMOUNT|FS_SINGLE);

static int __init init_pipe_fs(void)
{
        int err = register_filesystem(&pipe_fs_type);
        if (!err) {
                pipe_mnt = kern_mount(&pipe_fs_type);
                err = PTR_ERR(pipe_mnt);
                if (!IS_ERR(pipe_mnt))
                        err = 0;
        }
        return err;
}

static void __exit exit_pipe_fs(void)
{
        unregister_filesystem(&pipe_fs_type);
        kern_umount(pipe_mnt);
}

module_init(init_pipe_fs)
module_exit(exit_pipe_fs)
```

The filesystem is of type FS_NOMOUNT|FS_SINGLE which means it cannot be mounted from userspace and can only have one superblock system−wide. The FS_SINGLE file also means that it must be mounted via kern_mount() after it is successfully registered via register_filesystem() which is exactly what happens in init_pipe_fs(). The only bug in this function is that if kern_mount() fails (e.g. because kmalloc() failed in add_vfsmnt) then the filesystem is left as registered but module initialisation fails. This will cause "cat /proc/filesystems" to Oops. (have just sent a patch to Linus mentioning that although this is not a real bug today as pipefs can't be compiled as a module it should be written with the view that in the future it may become modularized).

The result of register_filesystem() is that pipe_fs_type is linked into the file_systems list so one can read /proc/filesystems and find "pipefs" entry in there with "nodev" flag indicating that FS_REQUIRES_DEV was not set. The /proc/filesystems file should really be enhanced to support all the new FS_ flags (and I made a patch to do so) but it cannot be done because it will break all the user applications that use it. Despite Linux kernel interfaces changing every minute (only for the better) when it comes to the userspace compatibility, Linux is a very conservative operating system which allows many applications to be used for a long time without being recompiled.

The result of kern_mount() is that:

1. A new unnamed (anonymous) device number is allocated by setting a bit unnamed_dev_in_use bitmap, If there are no more bits then kern_mount() fails with EMFILE
2. A new superblock structure is allocated by means of get_empty_super(). The get_empty_super() function walks the list of superblocks headed by super_block and looks for empty entry, i.e. s−>s_dev == 0. If no such empty superblock is found then a new one is allocated using kmalloc() at GFP_USER priority. The maximum system−wide number of superblocks is checked in get_empty_super() so if it starts failing, one can adjust the tunable /proc/sys/fs/super−max
3. A filesystem−specific pipe_fs_type−>read_super() method, i.e. pipefs_read_super() is invoked which

allocates root inode and root dentry sb−>s_root and set sb−>s_op to be &pipefs_ops

4. Then kern_mount() calls add_vfsmnt(NULL, sb−>s_root, "none") which allocates a new vfsmount structure and links it into vfsmntlist and sb−>s_mounts
5. The pipe_fs_type−>kern_mnt is set to this new vfsmount structure and it is returned. The reason why the return value of kern_mount() is a vfsmount structure is because even FS_SINGLE filesystems can be mounted multiple times and so their mnt−>mnt_sb will point to the same thing which would be silly to return from multiple calls to kern_mount()

Now that the filesystem is registered and inkernel−mounted we can use it. The entry point into the pipefs filesystem is the pipe(2) system call implemented in arch−dependent function sys_pipe() but the real work is done by a portable fs/pipe.c:do_pipe() function. Let us look at do_pipe() then. The interaction with pipefs happens when do_pipe() calls get_pipe_inode() to allocate a new pipefs inode. For this inode inode−>i_sb is set to pipefs' superblock pipe_mnt−>mnt_sb, the file operations i_fop is set to rdwr_pipe_fops and the number of readers and writers (held in inode−>i_pipe) is set to 1. The reason why there is a separate inode field i_pipe instead of keeping it in the fs−private union is that pipes and FIFOs share the same code and FIFOs can exist on other filesystems which use the other access paths within the same union which is very bad C and can work only by pure luck. So, yes, 2.2.x kernels work only by pure luck and will stop working as soon as you slightly rearrange the fields in the inode.

Each pipe(2) system call increments a reference count on the pipe_mnt mount instance.

Under Linux, the pipes are not symmetric (bidirection or STREAM pipes), i.e. two sides of the file have different file−>f_op operations − the read_pipe_fops and write_pipe_fops respectively. The write on read side returns EBADF and so does read on write side.

3.7 Example Disk Filesystem: BFS

As a simple example of ondisk Linux filesystem let us consider BFS. The preamble of the BFS module is in fs/bfs/inode.c:

```
static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);

static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}

static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}

module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

A special fstype declaration macro DECLARE_FSTYPE_DEV() is used which sets the fs_type−>flags to FS_REQUIRES_DEV to signify that BFS requires a real block device to be mounted on.

The module's initialisation function registers the filesystem with VFS and the cleanup function (only present when BFS is configured to be a module) unregisters it.

With the filesystem registered, we can proceed to mount it, which would invoke out fs_type−>read_super() method which is implemented in fs/bfs/inode.c:bfs_read_super(). It does the following:

1. set_blocksize(s−>s_dev, BFS_BSIZE) − since we are about to interact with block device layer via buffer cache we must initialise a few things, namely set the block size and also inform VFS via fields s−>s_blocksize and s−>s_blocksize_bits
2. bh = bread(dev, 0, BFS_BSIZE) − we read the block 0 of the device passed via s−>s_dev. This block is the filesystem's superblock
3. Superblock is validated against BFS_MAGIC number and if valid, stored in the sb−private field s−>su_sbh (which is really s−>u.bfs_sb.si_sbh)
4. Then we allocate inode bitmap using kmalloc(GFP_KERNEL) and clear all bits to 0 except the first two which we set to 1 to indicate that we should never allocate inodes 0 and 1. Inode 2 is root and the corresponding bit will be set to 1 a few lines later anyway − the filesystem should have a valid root inode at mounting time!
5. Then we initialise s−>s_op which means that we can from this point invoke inode cache via iget() which results in s_op−>read_inode() to be invoked. This finds the block that contains the specified (by inode−>i_ino and inode−>i_dev) inode and reads it in. If we fails to get root inode then we free the inode bitmap and release superblock buffer back to buffer cache and return NULL. If root inode was read OK, then we allocate a dentry with name "/" (as becometh root) and instantiate it with this inode
6. Now we go through all inodes on the filesystem and read them all in order to set the corresponding bits in our internal inode bitmap and also to calculate some other internal parameters like the offset of last inode and the start/end blocks of last file. Each inode we read is returned back to inode cache via iput() − we don't hold a reference to it longer than needed
7. If the filesystem was not mounted read−only we mark the superblock buffer dirty and set s−>s_dirt flag (TODO: why do I do this? Originally, I did it because minix_read_super() did but neither minix nor BFS seem to modify superblock in the read_super())
8. All is well so we return this initialised superblock back to the caller at VFS level, i.e. fs/super.c:read_super()

After the read_super() function returns successfully, VFS obtains the reference to the filesystem module via call to get_filesystem(fs_type) in fs/super.c:get_sb_bdev() and a reference to the block device.

Now, let us examine what happens when we do io on the filesystem. We already examined how inodes are read when iget() is called and how they are released on iput(). Reading inodes sets up among other things, inode−>i_op and inode−>i_fop and opening a file propagates inode−>i_fop into file−>f_op.

Let us examine the code path of the link(2) system call. The implementation of the system call is in fs/namei.c:sys_link():

1. The userspace names are copied into kernel space by means of getname() function which does the error checking
2. This names are nameidata converted using path_init()/path_walk() interaction with dcache. The result

is stored in old_nd and nd structures
3. If old_nd.mnt != nd.mnt then "cross−device link" EXDEV is returned − one cannot link between filesystems, in Linux this translates into − one cannot link between mounted instances of a filesystem (or, in particular between filesystems)
4. A new dentry is created corresponding to nd by lookup_create()
5. A generic vfs_link() function is called which checks if we can create a new entry in the directory and invokes the dir−>i_op−>link() method which brings us back to filesystem−specific fs/bfs/dir.c:bfs_link() function
6. Inside bfs_link() we check if we are trying to link a directory and refuse with EPERM error. This is the same behaviour as standard (ext2)
7. We attempt to add a new directory entry to the specified directory by calling the helper function bfs_add_entry() which goes through all entries looking for unused slot (de−>ino == 0) and when found writes out the name/inode pair into the corresponding block and marks it dirty (at non−superblock priority)
8. If we successfully added the directory entry then there is no way to fail the operation so we increment inode−>i_nlink and update inode−>i_ctime and mark this inode dirty as well as instantiating the new dentry with the inode

Other related inode operations like unlink()/rename() etc work in a similar way so not much is gained by examining them all in details.

3.8 Execution Domains and Binary Formats

Linux supports loading user application binaries from disk. More interestingly, the binaries can be stored in different formats and the operating system's response to programs via system calls can deviate from norm (norm being the Linux behaviour) as required, in order to emulate formats found in other flavours of UNIX (coff etc.) and also to emulate system calls behaviour of other flavours (Solaris, UnixWare etc.). This is what execution domains and binary formats are for.

Each Linux task has a personality stored in its task_struct p−>personality. The currently existing (either in the official kernel or as addon patch) personalities include support for FreeBSD, Solaris, UnixWare, OpenServer and many other popular operating systems. The value of current−>personality is split into two parts:

1. high byte − bug emulation:STICKY_TIMEOUTS,WHOLE_SECONDS etc
2. low byte − personality proper, a unique number

By changing personality we can change the way the operating system treats certain system calls, for example adding a STICKY_TIMEOUT to current−>personality makes select(2) system call to preserve the value of last argument (timeout) instead of storing the unslept time. Some buggy programs rely on buggy operating systems (non−Linux) and so Linux provides a way to emulate bugs in cases where the source code is not available and so bugs cannot be fixed.

Execution domain is a contiguous range of personalities implemented by a single module. Usually a single execution domain implements a single personality but sometimes it is possible to implement "close" personalities in a single module without too many conditionals.

Execution domains are implemented in kernel/exec_domain.c and were completely rewritten for 2.4 kernel, compared with 2.2.x. The list of execution domains currently supported by the kernel, along with the range of personalities they support, is available by reading the /proc/execdomains file. Execution domains, except the PER_LINUX one, can be implemented as dynamically loadable modules.

The user interface is via personality(2) system call which sets the current process' personality or returns the value of current−>personality if the argument is set to impossible personality 0xffffffff. Obviously, the behaviour of this system call itself does not depend on personality..

The kernel interface to execution domains registration consists of two functions:

- int register_exec_domain(struct exec_domain *) − registers the execution domain by linking it into single−linked list exec_domains under the write protection of the read−write spinlock exec_domains_lock. Returns 0 on success, non−zero on failure
- int unregister_exec_domain(struct exec_domain *) − unregisters the execution domain by unlinking it from the exec_domains list, again using exec_domains_lock spinlock in write mode. Returns 0 on success
-

The reason why exec_domains_lock is a read−write is that only registration and unregistration requests modify the list, whilst doing "cat /proc/filesystems" calls fs/exec_domain.c:get_exec_domain_list() which needs only read access to the list. Registering a new execution domain defines a "lcall7 handler" and a signal number conversion map. Actually, ABI patch extends this concept of exec domain to include extra information (like socket options, socket types, address family and errno maps).

The binary formats are implemented in a similar manner, i.e. a single−linked list formats is defined in fs/exec.c and is protected by a read−write lock binfmt_lock. As with exec_domains_lock, the binfmt_lock is taken read on most occasions except for registration/unregistration of binary formats. Registering a new binary format enhances the execve(2) system call with new load_binary()/load_shlib() functions as well as ability to core_dump(). The load_shlib() method is used only by the old uselib(2) system call while the load_binary() method is called by the search_binary_handler() from do_execve() which implements execve(2) system call.

The personality of the process is determined at binary format loading by the corresponding format's load_binary() method using some heuristics. For example to determine UnixWare7 binaries one first marks the binary using elfmark(1) utility which sets the ELF header's e_flags to the magic value 0x314B4455 which is detected at ELF loading time and current−>personality is set to PER_UW7. If this heuristic fails, then a more generic one, such as treat ELF interpreter paths like "/usr/lib/ld.so.1" or "/usr/lib/libc.so.1" to indicate a SVR4 binary, is used and personality is set to PER_SVR4. One could write a little utility program that uses Linux's ptrace(2) capabilities to single−step the code and force a running program into any personality.

Once personality (and therefore current−>exec_domain) is known, the system calls are handled as follows. Let us assume that a process makes a system call by means of lcall7 gate instruction. This transfers control to ENTRY(lcall7) of arch/i386/kernel/entry.S because it was prepared in arch/i386/kernel/traps.c:trap_init(). After appropriate stack layout conversion entry.S:lcall7 obtains the pointer to exec_domain from current and then an offset of lcall7 handler within the exec_domain (which is hardcoded as 4 in asm code so you can't shift the 'handler' field around in C declaration of struct exec_domain) and jumps to it. So, in C, it would look like this:

```
static void UW7_lcall7(int segment, struct pt_regs * regs)
```

```
{
       abi_dispatch(regs, &uw7_funcs[regs->eax & 0xff], 1);
}
```

where abi_dispatch() is a wrapper around the table of function pointers that implement this personality's system calls uw7_funcs.