Linux Kernel 2.4 Internals

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Linux Kernel 2.4 Internals

Tigran Aivazian `tigran@veritas.com`

7 August 2002 (29 Av 6001)

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 guide is now part of the Linux Documentation Project and can also be downloaded in various formats from: http://www.linuxdoc.org/guides.html or can be read online (latest version) at: http://www.moses.uklinux.net/patches/lki.html 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. Thanks to Juan J. Quintela `(quintela@fi.udc.es)`*, Francis Galiegue* `(fg@mandrakesoft.com)`*, Hakjun Mun* `(juniorm@orgio.net)`*, Matt Kraai* `(kraai@alumni.carnegiemellon.edu)`*, Nicholas Dronen* `(ndronen@frii.com)`*, Samuel S Chessman* `(chessman@tux.org)`*, Nadeem Hasan* `(nhasan@nadmm.com)`*, Michael Svetlik* `(m.svetlik@ssi-schaefer-peem.com)` *for various corrections and suggestions. The Linux Page Cache chapter was written by: Christoph Hellwig* `(hch@caldera.de)`*. The IPC Mechanisms chapter was written by: Russell Weight* `(weightr@us.ibm.com)` *and Mingming Cao* `(mcao@us.ibm.com)`

---

1. Booting

2. Process and Interrupt Management

3. Virtual Filesystem (VFS)

4. Linux Page Cache

5. IPC mechanisms

---

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` and `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` and `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 while **tools/build** does validate the size of boot sector, kernel image and lower bound of setup size, it does not check the *upper* bound of said setup size. Therefore it is easy to build a broken kernel by just 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 is 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=0xFFFF0000,%eip = 0x0000FFF0 (ROM BIOS POST code).
4. All POST checks are performed with interrupts disabled.
5. IVT (Interrupt Vector Table) 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, or
- no bootsector (loadlin etc)

We consider here the Linux bootsector in detail. The first few lines initialise 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` and `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.
```

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).

Line 64 jumps to label `go:` in the newly made copy of the bootsector, i.e. in segment 0x9000. This and the following three instructions (lines 64−76) prepare the stack at $INITSEG:0x4000−0xC, i.e. %ss = $INITSEG (0x9000) and %sp = 0x3FF4 (0x4000−0xC). This is where the limit on setup size comes from that we mentioned earlier (see Building the Linux Kernel Image).

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 recognise
 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
```

```
        movw    %di, %fs:(%bx)
        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), 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 old bootloaders (old versions of LILO) could only load the first 4 sectors of setup, which is why 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 the `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 approximately 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. This ensures 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 specialised 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. Initialise segment values (%ds = %es = %fs = %gs = __KERNEL_DS = 0x18).
2. Initialise page tables.
3. Enable paging by setting PG bit in %cr0.
4. Zero−clean BSS (on SMP, only first CPU does this).
5. Copy the first 2k of bootup parameters (kernel commandline).
6. Check 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. Take a global kernel lock (it is needed so that only one CPU goes through initialisation).
2. Perform arch−specific setup (memory layout analysis, copying boot command line again, etc.).
3. Print 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. Initialise traps.
5. Initialise irqs.
6. Initialise data required for scheduler.

7. Initialise time keeping data.
8. Initialise softirq subsystem.
9. Parse boot commandline options.
10. Initialise console.
11. If module support was compiled into the kernel, initialise dynamical module loading facility.
12. If "profile=" command line was supplied, initialise profiling buffers.
13. `kmem_cache_init()`, initialise most of slab allocator.
14. Enable interrupts.
15. Calculate BogoMips value for this CPU.
16. Call `mem_init()` which calculates `max_mapnr`, `totalram_pages` and `high_memory` and prints out the "Memory: ..." line.
17. `kmem_cache_sizes_init()`, finish slab allocator initialisation.
18. Initialise data structures used by procfs.
19. `fork_init()`, create `uid_cache`, initialise `max_threads` based on the amount of memory available and configure `RLIMIT_NPROC` for `init_task` to be `max_threads/2`.
20. Create various slab caches needed for VFS, VM, buffer cache, etc.
21. If System V IPC support is compiled in, initialise the 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. Perform arch−specific "check for bugs" and, whenever possible, activate 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. Set a flag to indicate that a schedule should be invoked at "next opportunity" and create 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; if all these fail, panic with "suggestion" to use "init=" parameter.
25. Go 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 in which 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 in which 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 a 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 always work, provided they are listed sequentially in the same Makefile. If they don't work, change the order in which their object files are listed.

Another thing worth noting is Linux's ability to execute an "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 write 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 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 `initialize_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. This is why `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 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 an 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 modularised if needed. Example of this is pipefs, see `fs/pipe.c`. Even if a given subsystem will never become a module, e.g. bdflush (see `fs/buffer.c`), it is still nice and tidy to use the `module_init()` macro against its initialisation function, provided it does not matter when exactly is the function called.

There are two more macros which work in a similar manner, 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.
3. `arch/i386/kernel/setup.c:parse_mem_cmdline()` (called by `setup_arch()`, itself called by `start_kernel()`) copies 256 bytes from zeropage into `saved_command_line` which is displayed by `/proc/cmdline`. This same routine processes the "mem=" option if present 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 the `__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 `__setup()` does nothing for modules, 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. 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 which can be created on Linux 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`. As an example, on a 512M machine, you can create 32k threads. This is a considerable improvement over the 4k−epsilon limit for older (2.2 and earlier) kernels. Moreover, this can be changed at runtime using the 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, and
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 read−write 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 the `for_each_task()` macro from `include/linux/sched.h`:

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

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 `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()` function walks the task list and thus takes `tasklist_lock` for READ, and it is called from `kill_fasync()` in interrupt context. This is why writers must disable interrupts while 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 correspond to the members of UNIX 'struct proc' and 'struct user' combined together.

The other versions of UNIX separated the task state information into one part which should be kept memory−resident at all times (called 'proc structure' which includes process state, scheduling information etc.) and another 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 a task as `TASK_RUNNING` and placing it on the runqueue is not atomic. You need to hold the `runqueue_lock` read–write spinlock for read in order to look at the runqueue. If you do so, you will then see that every task on the runqueue is in `TASK_RUNNING` state. However, the converse is not true for the reason explained above. Similarly, drivers can mark themselves (or rather the process context they run in) as `TASK_INTERRUPTIBLE` (or `TASK_UNINTERRUPTIBLE`) and then call `schedule()`, which will then remove it from the runqueue (unless there is a pending signal, in which case it is left on the runqueue).
2. **TASK_INTERRUPTIBLE**: means the task is sleeping but can be woken up by a signal or by expiry of a timer.
3. **TASK_UNINTERRUPTIBLE**: same as `TASK_INTERRUPTIBLE`, except it cannot be woken up.
4. **TASK_ZOMBIE**: task has terminated but has not had its status collected (`wait()`–ed for) by the parent (natural or by adoption).
5. **TASK_STOPPED**: task was stopped, either due to job control signals or due to **ptrace(2)**.
6. **TASK_EXCLUSIVE**: this is not a separate state but can be OR–ed to either one of `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 respectively 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 helps minimise 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` relate to the personality of the task, i.e. to the way certain system calls behave in order to emulate the "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.

This structure also includes a reference count because it can be shared between cloned tasks when `CLONE_FS` flag is passed to the **clone(2)** system call.

The field `p->files` contains the file descriptor table. This too can be shared between tasks, provided `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`.

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:

- the idle thread(s),
- kernel threads,
- user tasks.

The idle thread is created at compile time for the first CPU; it is then "manually" created for each CPU by means of arch–specific `fork_by_hand()` in `arch/i386/kernel/smpboot.c`, which unrolls the **fork(2)** system call by hand (on some archs). Idle tasks share one init_task structure but have a private TSS structure, in the 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(2)** 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 (on x86, that is) implies that the kernel threads enjoy all I/O 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 **fork(2)** 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 this is the value which `errno` should be 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)**, this 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 `exec()`ing 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 memcpy? 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, i.e. `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 plain **fork(2)**, this will be `p->flags = PF_FORKNOEXEC`.
15. The child's 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 the `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()` while 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. The 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`: this field is 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 a timer. When this field becomes lower than or equal to zero, it is reset to 0 and `p->need_resched` is set. This is also sometimes called 'dynamic priority' of a process because it can change by itself.
- `p->priority`: the process' 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`: the scheduling policy, specifies which scheduling class the task belongs to. Tasks can change their scheduling class using the **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. A FIFO realtime process will run until either a) it blocks on I/O, b) it explicitly yields the CPU or c) it is preempted by another realtime process with a higher `p->rt_priority` value. `SCHED_RR` is the same as `SCHED_FIFO`, except that when its 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 optimised (for i386) code. Also, note that scheduler (like most of the kernel) was completely rewritten for 2.4, therefore the discussion below does not apply to 2.2 or earlier 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 the `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. Initialise 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. Initialise local pointer `struct schedule_data *sched_data` to point to per−CPU (cacheline−aligned to prevent cacheline ping−pong) scheduling data area, which contains the 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. Therefore, 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` state and a signal is pending, 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 (−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; 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 a 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 the ones on the 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. The reason is that we go through entire set of processes; this can take a long time, during which 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. 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. Wait queues (as well as everything else in Linux) make 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 `wake_up/wake_up_interruptible` to wake up when needed.

An example of the first usage of waitqueues is interaction between the page allocator (in `mm/page_alloc.c:__alloc_pages()`) and the `kswapd` kernel daemon (in `mm/vmscan.c:kswap()`), by means of wait queue `kswapd_wait,` declared in `mm/vmscan.c`; the `kswapd` daemon sleeps on this queue, and it is woken up whenever the 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 I/O 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 ourselves from the wait queue and return
5. If there is no data yet, we check whether the user specified non–blocking I/O 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 us sooner than when the data is available, thus causing unneeded processing.

It is also worth pointing out that, using wait queues, it is rather easy to implement the **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 the device−independent function `poll_wait()` which does the necessary waitqueue manipulations; all we need to do is 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 work (e.g. acknowledging the interrupt, updating the stats etc.) and work 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 Linux 2.0, a new mechanism was added, called 'task queues', which will be the subject of next section.

Bottom halves are serialised by the `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 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, a 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, a subsystem's cleanup routine (`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` list all registered bottom halves on the system? That means `global_bh_lock` must be made read/write, obviously)
- `void mark_bh(int nr)`: marks bottom half in slot `nr` 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 a 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 creates a new task queue using the `DECLARE_TASK_QUEUE()` macro and queues a task onto it using the `queue_task()` function. The task queue then can be processed using `run_task_queue()`. Instead of creating your own task queue (and having to consume it manually) you can use one of Linux' predefined task queues which are consumed at well−known points:

1. **tq_timer**: the timer task queue, run on each timer interrupt and when releasing a tty device (closing or releasing a half−opened terminal device). Since the timer handler runs in interrupt context, the `tq_timer` tasks also run in interrupt context and thus cannot block.
2. **tq_scheduler**: the scheduler task queue, 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**: this 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 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 the special purposes which it was designed for.

Unless a 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_scheduler` 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, and these tasks only make 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 interrupt 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 binaries from foreign flavours of UNIX (Solaris, UnixWare 7 etc.) use the lcall7 mechanism. The name 'lcall7' is historically misleading because it also covers 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 so that vector 0x80 (of type 15, dpl 3) points to the address of system_call entry from `arch/i386/kernel/entry.S`.

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

1. Save registers.

2. Set %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), 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`). 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 used for simple locking, for example to provide exclusive access to open a device. An example of this can be found 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, volatile void *addr)**: set bit `nr` in the bitmap pointed to by `addr`.
* **void clear_bit(int nr, volatile 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 bit `nr` and return the old bit value.
* **int test_and_clear_bit(int nr, volatile void *addr)**: atomically clear bit `nr` and return the old bit value.
* **int test_and_change_bit(int nr, volatile void *addr)**: atomically toggle bit `nr` and return the old bit value.

These operations use the `LOCK_PREFIX` macro, which on SMP kernels 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`; return 1 if the new value is 0, return 0 otherwise.
* **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; return 1 if the new value is 0, return 0 otherwise.
* **int atomic_inc_and_test(volatile atomic_t *v)**: increment the value; return 1 if the new value is 0, return 0 otherwise.
* **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), developers were faced with the classical 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 while `cli()` provides protection against races with interrupt context on each CPU individually, it provides no protection at all 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 the `file_systems_lock` read−write spinlock because one needs exclusive access only when registering/unregistering a filesystem, but any process can read the file `/proc/filesystems` or use the **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 a 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 (on lock) and re−enables (on unlock) interrupts on the current CPU. For example, `rtc_read()` uses `spin_lock_irq(&rtc_lock)` (interrupts are always enabled inside `read()`) whilst `rtc_interrupt()` uses `spin_lock(&rtc_lock)` (interrupts are always disabled inside interrupt handler). Note that `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 if 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 an interrupt comes in on the same CPU, it will busy wait for the lock forever: the lock holder, having been interrupted, 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()
{
        spin_lock_irq(&my_lock);
        /* critical section */
        spin_unlock_irq(&my_lock);
}

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 an 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 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 either 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: one can have multiple readers at a time but only one writer and there can 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/up_interruptible()` instead of the plain `down()/up()` and check the value returned from `down_interruptible()`: it will be non zero if the operation was interrupted.

Using semaphores for mutual exclusion is ideal in situations where a 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 a 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 compile time and is determined by the `CONFIG_MODULES` option. Support for module autoloading 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 modularised):

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`, 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 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 able 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 that 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 with 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 a 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 is being deleted, it is as good as if it were 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()` were guaranteed to be non−blocking and the module loading were executed in the same context atomically. The reason for this is that the module's initialisation function will try to call `register_filesystem()`, which will take the same `file_systems_lock` read−write spinlock for write.
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 successfully loaded the requested module, in which case `request_module()` will fail even though 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; it will refuse to load the module if these symbols differ. 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, 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 the 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 not many changes have been made to it for the last 5−7 years: one can still recognise 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`, where each inode is hashed by the value of the superblock pointer and 32bit inode number. Inodes without a 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` and `i_nlink>0`. Inodes newly allocated by `get_empty_inode()` and `get_new_inode()` are added to the `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`.

The inode cache subsystem is initialised when `inode_init()` function is called from `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 the 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 **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 the nameidata structure containing the dentry and vfsmount structures.
2. `dentry_open()`: given a dentry and vfsmount, this function allocates a new `struct file` and links them together; it also invokes the 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 the filesystem specific `inode_operations->lookup()` method. The role of this method 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. Attempt to find an inode with matching superblock and inode number in the hashtable under protection of `inode_lock`. If inode is found, its reference count (`i_count`) is incremented; if it was 0 prior to incrementation and the inode is not dirty, it 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; finally, `inodes_stat.nr_unused` is decremented.
2. If inode is currently locked, we wait until it is unlocked so that `iget4()` is guaranteed to return an unlocked 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 has dropped the spinlock, it must retry searching the inode in the hashtable afterwards; 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; therefore 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`); it is unlocked after the `read_inode()` method returns 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 the `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 immediately 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 check 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, therefore 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 the inode is removed from hashtable and from its type list; 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 the filesystem−specific `s_op−>delete_inode()` method is invoked, which typically deletes the 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, this 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 an anonymous inode (NetApp .snapshot) then we delete it from the type list and clear/destroy it completely.

3.2 Filesystem Registration/Unregistration

The 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. The VFS interface is exported, and therefore all Linux filesystems can be implemented as modules.

Let us consider the steps required to implement a filesystem under Linux. The code to implement 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 the `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)
```

The `module_init()/module_exit()` macros ensure that, when BFS is compiled as a module, the functions `init_bfs_fs()` and `exit_bfs_fs()` turn into `init_module()` and `cleanup_module()` respectively; if BFS is statically linked into the kernel, 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 a filesystem by its name; this same name is used for the filesystem type in **mount(2)**, and it should be unique: 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` is not 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 the `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` to 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 I/O 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 the `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 I/O to receive asynchronous I/O 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 the **mmap(2)** 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(2)** 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 a 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 the 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)`. 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)` . 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 I/O 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 `mnt_list` field allows to find all instances for all superblocks system−wide. The `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 modularised).

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 in `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 sets `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, 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 the block device layer via the 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 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 fail 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 I/O 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`; opening a file will propagate `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. These 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 if so, 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`, 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 three bytes − bug emulation: `STICKY_TIMEOUTS`, `WHOLE_SECONDS`, etc.
2. low byte − personality proper, a unique number.

By changing the 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 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 the **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`.

4. Linux Page Cache

In this chapter we describe the Linux 2.4 pagecache. The pagecache is − as the name suggests − a cache of physical pages. In the UNIX world the concept of a pagecache became popular with the introduction of SVR4 UNIX, where it replaced the buffercache for data IO operations.

While the SVR4 pagecache is only used for filesystem data cache and thus uses the struct vnode and an offset into the file as hash parameters, the Linux page cache is designed to be more generic, and therefore uses a struct address_space (explained below) as first parameter. Because the Linux pagecache is tightly coupled to the notation of address spaces, you will need at least a basic understanding of adress_spaces to understand the way the pagecache works. An address_space is some kind of software MMU that maps all pages of one object (e.g. inode) to an other concurrency (typically physical disk blocks). The struct address_space is defined in `include/linux/fs.h` as:

```
        struct address_space {
                struct list_head        clean_pages;
                struct list_head        dirty_pages;
                struct list_head        locked_pages;
                unsigned long           nrpages;
                struct address_space_operations *a_ops;
                struct inode            *host;
                struct vm_area_struct   *i_mmap;
                struct vm_area_struct   *i_mmap_shared;
                spinlock_t              i_shared_lock;

        };
```

To understand the way address_spaces works, we only need to look at a few of this fields: `clean_pages`, `dirty_pages` and `locked_pages` are double linked lists of all clean, dirty and locked pages that belong to this address_space, `nrpages` is the total number of pages in this address_space. `a_ops` defines the methods of this object and `host` is an pointer to the inode this address_space belongs to − it may also be NULL, e.g. in the case of the swapper address_space (`mm/swap_state.c,`).

The usage of `clean_pages`, `dirty_pages`, `locked_pages` and `nrpages` is obvious, so we will take a tighter look at the `address_space_operations` structure, defined in the same header:

```
struct address_space_operations {
        int (*writepage)(struct page *);
        int (*readpage)(struct file *, struct page *);
        int (*sync_page)(struct page *);
        int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
        int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
        int (*bmap)(struct address_space *, long);
};
```

For a basic view at the principle of address_spaces (and the pagecache) we need to take a look at `−>writepage` and `−>readpage`, but in practice we need to take a look at `−>prepare_write` and `−>commit_write`, too.

You can probably guess what the address_space_operations methods do by virtue of their names alone; nevertheless, they do require some explanation. Their use in the course of filesystem data I/O, by far the most common path through the pagecache, provides a good way of understanding them. Unlike most other UNIX−like operating systems, Linux has generic file operations (a subset of the SYSVish vnode operations) for data IO through the pagecache. This means that the data will not directly interact with the file− system on read/write/mmap, but will be read/written from/to the pagecache whenever possible. The pagecache has to get data from the actual low−level filesystem in case the user wants to read from a page not yet in memory, or write data to disk in case memory gets low.

In the read path the generic methods will first try to find a page that matches the wanted inode/index tuple.

```
hash = page_hash(inode->i_mapping, index);
```

Then we test whether the page actually exists.

```
hash = page_hash(inode->i_mapping, index); page =
__find_page_nolock(inode->i_mapping, index, *hash);
```

When it does not exist, we allocate a new free page, and add it to the page− cache hash.

```
page = page_cache_alloc(); __add_to_page_cache(page, mapping,
index, hash);
```

After the page is hashed we use the `−>readpage` address_space operation to actually fill the page with data. (file is an open instance of inode).

```
error = mapping->a_ops->readpage(file, page);
```

Finally we can copy the data to userspace.

For writing to the filesystem two pathes exist: one for writable mappings (mmap) and one for the write(2) family of syscalls. The mmap case is very simple, so it will be discussed first. When a user modifies mappings, the VM subsystem marks the page dirty.

```
SetPageDirty(page);
```

The bdflush kernel thread that is trying to free pages, either as background activity or because memory gets low will try to call `->writepage` on the pages that are explicitly marked dirty. The `->writepage` method does now have to write the pages content back to disk and free the page.

The second write path is _much_ more complicated. For each page the user writes to, we are basically doing the following: (for the full code see `mm/filemap.c:generic_file_write()`).

```
page = __grab_cache_page(mapping, index, &cached_page);
mapping->a_ops->prepare_write(file, page, offset,
offset+bytes); copy_from_user(kaddr+offset, buf, bytes);
mapping->a_ops->commit_write(file, page, offset,
offset+bytes);
```

So first we try to find the hashed page or allocate a new one, then we call the `->prepare_write` address_space method, copy the user buffer to kernel memory and finally call the `->commit_write` method. As you probably have seen ->prepare_write and `->commit_write` are fundamentally different from `->readpage` and `->writepage`, because they are not only called when physical IO is actually wanted but everytime the user modifies the file. There are two (or more?) ways to handle this, the first one uses the Linux buffercache to delay the physical IO, by filling a `page->buffers` pointer with buffer_heads, that will be used in try_to_free_buffers (`fs/buffers.c`) to request IO once memory gets low, and is used very widespread in the current kernel. The other way just sets the page dirty and relies on `->writepage` to do all the work. Due to the lack of a validitity bitmap in struct page this does not work with filesystem that have a smaller granuality then `PAGE_SIZE`.

5. IPC mechanisms

This chapter describes the semaphore, shared memory, and message queue IPC mechanisms as implemented in the Linux 2.4 kernel. It is organized into four sections. The first three sections cover the interfaces and support functions for semaphores, message queues, and shared memory respectively. The last section describes a set of common functions and data structures that are shared by all three mechanisms.

5.1 Semaphores

The functions described in this section implement the user level semaphore mechanisms. Note that this implementation relies on the use of kernel splinlocks and kernel semaphores. To avoid confusion, the term "kernel semaphore" will be used in reference to kernel semaphores. All other uses of the word "sempahore" will be in reference to the user level semaphores.

Semaphore System Call Interfaces

sys_semget()

The entire call to sys_semget() is protected by the global sem_ids.sem kernel semaphore.

In the case where a new set of semaphores must be created, the newary() function is called to create and initialize a new semaphore set. The ID of the new set is returned to the caller.

In the case where a key value is provided for an existing semaphore set, ipc_findkey() is invoked to look up the corresponding semaphore descriptor array index. The parameters and permissions of the caller are verified before returning the semaphore set ID.

sys_semctl()

For the IPC_INFO, SEM_INFO, and SEM_STAT commands, semctl_nolock() is called to perform the necessary functions.

For the GETALL, GETVAL, GETPID, GETNCNT, GETZCNT, IPC_STAT, SETVAL,and SETALL commands, semctl_main() is called to perform the necessary functions.

For the IPC_RMID and IPC_SET command, semctl_down() is called to perform the necessary functions. Throughout both of these operations, the global sem_ids.sem kernel semaphore is held.

sys_semop()

After validating the call parameters, the semaphore operations data is copied from user space to a temporary buffer. If a small temporary buffer is sufficient, then a stack buffer is used. Otherwise, a larger buffer is allocated. After copying in the semaphore operations data, the global semaphores spinlock is locked, and the user−specified semaphore set ID is validated. Access permissions for the semaphore set are also validated.

All of the user−specified semaphore operations are parsed. During this process, a count is maintained of all the operations that have the SEM_UNDO flag set. A `decrease` flag is set if any of the operations subtract from a semaphore value, and an `alter` flag is set if any of the semaphore values are modified (i.e. increased or decreased). The number of each semaphore to be modified is validated.

If SEM_UNDO was asserted for any of the semaphore operations, then the undo list for the current task is searched for an undo structure associated with this semaphore set. During this search, if the semaphore set ID of any of the undo structures is found to be −1, then freeundos() is called to free the undo structure and remove it from the list. If no undo structure is found for this semaphore set then alloc_undo() is called to allocate and initialize one.

The try_atomic_semop() function is called with the `do_undo` parameter equal to 0 in order to execute the sequence of operations. The return value indicates that either the operations passed, failed, or were not executed because they need to block. Each of these cases are further described below:

Non−blocking Semaphore Operations

The try_atomic_semop() function returns zero to indicate that all operations in the sequence succeeded. In this case, update_queue() is called to traverse the queue of pending semaphore operations for the semaphore set and awaken any sleeping tasks that no longer need to block. This completes the execution of the sys_semop() system call for this case.

Failing Semaphore Operations

If try_atomic_semop() returns a negative value, then a failure condition was encountered. In this case, none of the operations have been executed. This occurs when either a semaphore operation would cause an invalid semaphore value, or an operation marked IPC_NOWAIT is unable to complete. The error condition is then returned to the caller of sys_semop().

Before sys_semop() returns, a call is made to update_queue() to traverse the queue of pending semaphore operations for the semaphore set and awaken any sleeping tasks that no longer need to block.

Blocking Semaphore Operations

The try_atomic_semop() function returns 1 to indicate that the sequence of semaphore operations was not executed because one of the semaphores would block. For this case, a new sem_queue element is initialized containing these semaphore operations. If any of these operations would alter the state of the semaphore, then the new queue element is added at the tail of the queue. Otherwise, the new queue element is added at the head of the queue.

The `semsleeping` element of the current task is set to indicate that the task is sleeping on this sem_queue element. The current task is marked as TASK_INTERRUPTIBLE, and the `sleeper` element of the sem_queue is set to identify this task as the sleeper. The global semaphore spinlock is then unlocked, and schedule() is called to put the current task to sleep.

When awakened, the task re−locks the global semaphore spinlock, determines why it was awakened, and how it should respond. The following cases are handled:

- If the the semaphore set has been removed, then the system call fails with EIDRM.
- If the `status` element of the sem_queue structure is set to 1, then the task was awakened in order to retry the semaphore operations. Another call to try_atomic_semop() is made to execute the sequence of semaphore operations. If try_atomic_sweep() returns 1, then the task must block again as described above. Otherwise, 0 is returned for success, or an appropriate error code is returned in case of failure. Before sys_semop() returns, current−>semsleeping is cleared, and the sem_queue is removed from the queue. If any of the specified semaphore operations were altering operations (increase or decrease), then update_queue() is called to traverse the queue of pending semaphore operations for the semaphore set and awaken any sleeping tasks that no longer need to block.
- If the `status` element of the sem_queue structure is NOT set to 1, and the sem_queue element has not been dequeued, then the task was awakened by an interrupt. In this case, the system call fails with EINTR. Before returning, current−>semsleeping is cleared, and the sem_queue is removed from the queue. Also, update_queue() is called if any of the operations were altering operations.
- If the `status` element of the sem_queue structure is NOT set to 1, and the sem_queue element has been dequeued, then the semaphore operations have already been executed by update_queue(). The queue `status`, which could be 0 for success or a negated error code for failure, becomes the return value of the system call.

Semaphore Specific Support Structures

The following structures are used specifically for semaphore support:

struct sem_array

```
/* One sem_array data structure for each set of semaphores in the system. */
struct sem_array {
    struct kern_ipc_perm sem_perm; /* permissions .. see ipc.h */
    time_t sem_otime; /* last semop time */
    time_t sem_ctime; /* last change time */
    struct sem *sem_base; /* ptr to first semaphore in array */
    struct sem_queue *sem_pending; /* pending operations to be processed */
    struct sem_queue **sem_pending_last; /* last pending operation */
    struct sem_undo *undo; /* undo requests on this array * /
    unsigned long sem_nsems; /* no. of semaphores in array */
};
```

struct sem

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
        int     semval;         /* current value */
        int     sempid;         /* pid of last operation */
};
```

struct seminfo

```
struct  seminfo {
        int semmap;
        int semmni;
        int semmns;
        int semmnu;
        int semmsl;
        int semopm;
        int semume;
        int semusz;
        int semvmx;
        int semaem;
};
```

struct semid64_ds

```
struct semid64_ds {
        struct ipc64_perm sem_perm;             /* permissions .. see
ipc.h */
        __kernel_time_t sem_otime;              /* last semop time */
        unsigned long   __unused1;
        __kernel_time_t sem_ctime;              /* last change time */
        unsigned long   __unused2;
        unsigned long   sem_nsems;              /* no. of semaphores in
array */
```

```
        unsigned long   __unused3;
        unsigned long   __unused4;
};
```

struct sem_queue

```
/* One queue for each sleeping process in the system. */
struct sem_queue {
        struct sem_queue *      next;    /* next entry in the queue */
        struct sem_queue **     prev;    /* previous entry in the queue, *(q->pr
ev) == q */
        struct task_struct*     sleeper; /* this process */
        struct sem_undo *       undo;    /* undo structure */
        int                     pid;     /* process id of requesting process */
        int                     status;  /* completion status of operation */
        struct sem_array *      sma;     /* semaphore array for operations */
        int                     id;      /* internal sem id */
        struct sembuf *         sops;    /* array of pending operations */
        int                     nsops;   /* number of operations */
        int                     alter;   /* operation will alter semaphore */
};
```

struct sembuf

```
/* semop system calls takes an array of these. */
struct sembuf {
        unsigned short  sem_num;        /* semaphore index in array */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

struct sem_undo

```
/* Each task has a list of undo requests. They are executed automatically
 * when the process exits.
 */
struct sem_undo {
        struct sem_undo *       proc_next;      /* next entry on this process */
        struct sem_undo *       id_next;        /* next entry on this semaphore set */
        int                     semid;          /* semaphore set identifier */
        short *                 semadj;         /* array of adjustments, one per
 semaphore */
};
```

Semaphore Support Functions

The following functions are used specifically in support of semaphores:

newary()

newary() relies on the ipc_alloc() function to allocate the memory required for the new semaphore set. It allocates enough memory for the semaphore set descriptor and for each of the semaphores in the set. The allocated memory is cleared, and the address of the first element of the semaphore set descriptor is passed to ipc_addid(). ipc_addid() reserves an array entry for the new semaphore set descriptor and initializes the (struct kern_ipc_perm) data for the set. The global `used_sems` variable is updated by the number of semaphores in the new set and the initialization of the (struct kern_ipc_perm) data for the new set is completed. Other initialization for this set performed are listed below:

- The `sem_base` element for the set is initialized to the address immediately following the (struct sem_array) portion of the newly allocated data. This corresponds to the location of the first semaphore in the set.
- The `sem_pending` queue is initialized as empty.

All of the operations following the call to ipc_addid() are performed while holding the global semaphores spinlock. After unlocking the global semaphores spinlock, newary() calls ipc_buildid() (via sem_buildid()). This function uses the index of the semaphore set descriptor to create a unique ID, that is then returned to the caller of newary().

freeary()

freeary() is called by semctl_down() to perform the functions listed below. It is called with the global semaphores spinlock locked and it returns with the spinlock unlocked

- The ipc_rmid() function is called (via the sem_rmid() wrapper) to delete the ID for the semaphore set and to retrieve a pointer to the semaphore set.
- The undo list for the semaphore set is invalidated.
- All pending processes are awakened and caused to fail with EIDRM.
- The number of used semaphores is reduced by the number of semaphores in the removed set.
- The memory associated with the semaphore set is freed.

semctl_down()

semctl_down() provides the IPC_RMID and IPC_SET operations of the semctl() system call. The semaphore set ID and the access permissions are verified prior to either of these operations, and in either case, the global semaphore spinlock is held throughout the operation.

IPC_RMID

The IPC_RMID operation calls freeary() to remove the semaphore set.

IPC_SET

The IPC_SET operation updates the `uid`, `gid`, `mode`, and `ctime` elements of the semaphore set.

semctl_nolock()

semctl_nolock() is called by sys_semctl() to perform the IPC_INFO, SEM_INFO and SEM_STAT functions.

IPC_INFO and SEM_INFO

IPC_INFO and SEM_INFO cause a temporary seminfo buffer to be initialized and loaded with unchanging semaphore statistical data. Then, while holding the global `sem_ids.sem` kernel semaphore, the `semusz` and `semaem` elements of the seminfo structure are updated according to the given command (IPC_INFO or SEM_INFO). The return value of the system call is set to the maximum semaphore set ID.

SEM_STAT

SEM_STAT causes a temporary semid64_ds buffer to be initialized. The global semaphore spinlock is then held while copying the `sem_otime`, `sem_ctime`, and `sem_nsems` values into the buffer. This data is then copied to user space.

semctl_main()

semctl_main() is called by sys_semctl() to perform many of the supported functions, as described in the subsections below. Prior to performing any of the following operations, semctl_main() locks the global semaphore spinlock and validates the semaphore set ID and the permissions. The spinlock is released before returning.

GETALL

The GETALL operation loads the current semaphore values into a temporary kernel buffer and copies them out to user space. The small stack buffer is used if the semaphore set is small. Otherwise, the spinlock is temporarily dropped in order to allocate a larger buffer. The spinlock is held while copying the semaphore values in to the temporary buffer.

SETALL

The SETALL operation copies semaphore values from user space into a temporary buffer, and then into the semaphore set. The spinlock is dropped while copying the values from user space into the temporary buffer, and while verifying reasonable values. If the semaphore set is small, then a stack buffer is used, otherwise a larger buffer is allocated. The spinlock is regained and held while the following operations are performed on the semaphore set:

- The semaphore values are copied into the semaphore set.
- The semaphore adjustments of the undo queue for the semaphore set are cleared.
- The `sem_ctime` value for the semaphore set is set.
- The update_queue() function is called to traverse the queue of pending semops and look for any tasks that can be completed as a result of the SETALL operation. Any pending tasks that are no longer blocked are awakened.

IPC_STAT

In the IPC_STAT operation, the `sem_otime`, `sem_ctime`, and `sem_nsems` value are copied into a stack buffer. The data is then copied to user space after dropping the spinlock.

GETVAL

For GETVAL in the non−error case, the return value for the system call is set to the value of the specified semaphore.

GETPID

For GETPID in the non−error case, the return value for the system call is set to the `pid` associated with the last operation on the semaphore.

GETNCNT

For GETNCNT in the non−error case, the return value for the system call is set to the number of processes waiting on the semaphore being less than zero. This number is calculated by the count_semncnt() function.

GETZCNT

For GETZCNT in the non−error case, the return value for the system call is set to the number of processes waiting on the semaphore being set to zero. This number is calculated by the count_semzcnt() function.

SETVAL

After validating the new semaphore value, the following functions are performed:

- The undo queue is searched for any adjustments to this semaphore. Any adjustments that are found are reset to zero.
- The semaphore value is set to the value provided.
- The `sem_ctime` value for the semaphore set is updated.
- The update_queue() function is called to traverse the queue of pending semops and look for any tasks that can be completed as a result of the SETALL operation. Any pending tasks that are no longer blocked are awakened.

count_semncnt()

count_semncnt() counts the number of tasks waiting on the value of a semaphore to be less than zero.

count_semzcnt()

count_semzcnt() counts the number of tasks waiting on the value of a semaphore to be zero.

update_queue()

update_queue() traverses the queue of pending semops for a semaphore set and calls try_atomic_semop() to determine which sequences of semaphore operations would succeed. If the status of the queue element indicates that blocked tasks have already been awakened, then the queue element is skipped over. For other elements of the queue, the `q-alter` flag is passed as the undo parameter to try_atomic_semop(), indicating that any altering operations should be undone before returning.

If the sequence of operations would block, then update_queue() returns without making any changes.

A sequence of operations can fail if one of the semaphore operations would cause an invalid semaphore value, or an operation marked IPC_NOWAIT is unable to complete. In such a case, the task that is blocked on the sequence of semaphore operations is awakened, and the queue status is set with an appropriate error code. The queue element is also dequeued.

If the sequence of operations is non−altering, then they would have passed a zero value as the undo parameter to try_atomic_semop(). If these operations succeeded, then they are considered complete and are removed from the queue. The blocked task is awakened, and the queue element `status` is set to indicate success.

If the sequence of operations would alter the semaphore values, but can succeed, then sleeping tasks that no longer need to be blocked are awakened. The queue status is set to 1 to indicate that the blocked task has been awakened. The operations have not been performed, so the queue element is not removed from the queue. The semaphore operations would be executed by the awakened task.

try_atomic_semop()

try_atomic_semop() is called by sys_semop() and update_queue() to determine if a sequence of semaphore operations will all succeed. It determines this by attempting to perform each of the operations.

If a blocking operation is encountered, then the process is aborted and all operations are reversed. −EAGAIN is returned if IPC_NOWAIT is set. Otherwise 1 is returned to indicate that the sequence of semaphore operations is blocked.

If a semaphore value is adjusted beyond system limits, then then all operations are reversed, and −ERANGE is returned.

If all operations in the sequence succeed, and the `do_undo` parameter is non−zero, then all operations are reversed, and 0 is returned. If the `do_undo` parameter is zero, then all operations succeeded and remain in force, and the `sem_otime`, field of the semaphore set is updated.

sem_revalidate()

sem_revalidate() is called when the global semaphores spinlock has been temporarily dropped and needs to be locked again. It is called by semctl_main() and alloc_undo(). It validates the semaphore ID and permissions and on success, returns with the global semaphores spinlock locked.

freeundos()

freeundos() traverses the process undo list in search of the desired undo structure. If found, the undo structure is removed from the list and freed. A pointer to the next undo structure on the process list is returned.

alloc_undo()

alloc_undo() expects to be called with the global semaphores spinlock locked. In the case of an error, it returns with it unlocked.

The global semaphores spinlock is unlocked, and kmalloc() is called to allocate sufficient memory for both the sem_undo structure, and also an array of one adjustment value for each semaphore in the set. On success, the global spinlock is regained with a call to sem_revalidate().

The new semundo structure is then initialized, and the address of this structure is placed at the address provided by the caller. The new undo structure is then placed at the head of undo list for the current task.

sem_exit()

sem_exit() is called by do_exit(), and is responsible for executing all of the undo adjustments for the exiting task.

If the current process was blocked on a semaphore, then it is removed from the sem_queue list while holding the global semaphores spinlock.

The undo list for the current task is then traversed, and the following operations are performed while holding and releasing the the global semaphores spinlock around the processing of each element of the list. The following operations are performed for each of the undo elements:

- The undo structure and the semaphore set ID are validated.
- The undo list of the corresponding semaphore set is searched to find a reference to the same undo structure and to remove it from that list.
- The adjustments indicated in the undo structure are applied to the semaphore set.
- The `sem_otime` parameter of the semaphore set is updated.
- update_queue() is called to traverse the queue of pending semops and awaken any sleeping tasks that no longer need to be blocked as a result of executing the undo operations.
- The undo structure is freed.

When the processing of the list is complete, the current−>semundo value is cleared.

5.2 Message queues

Message System Call Interfaces

sys_msgget()

The entire call to sys_msgget() is protected by the global message queue semaphore (msg_ids.sem).

In the case where a new message queue must be created, the newque() function is called to create and initialize a new message queue, and the new queue ID is returned to the caller.

If a key value is provided for an existing message queue, then ipc_findkey() is called to look up the corresponding index in the global message queue descriptor array (msg_ids.entries). The parameters and permissions of the caller are verified before returning the message queue ID. The look up operation and verification are performed while the global message queue spinlock(msg_ids.ary) is held.

sys_msgctl()

The parameters passed to sys_msgctl() are: a message queue ID (`msqid`), the operation (`cmd`), and a pointer to a user space buffer of type msgid_ds (`buf`). Six operations are provided in this function: IPC_INFO, MSG_INFO,IPC_STAT, MSG_STAT, IPC_SET and IPC_RMID. The message queue ID and the operation parameters are validated; then, the operation(cmd) is performed as follows:

IPC_INFO (or MSG_INFO)

The global message queue information is copied to user space.

IPC_STAT (or MSG_STAT)

A temporary buffer of type struct msqid64_ds is initialized and the global message queue spinlock is locked. After verifying the access permissions of the calling process, the message queue information associated with the message queue ID is loaded into the temporary buffer, the global message queue spinlock is unlocked, and the contents of the temporary buffer are copied out to user space by copy_msqid_to_user().

IPC_SET

The user data is copied in via copy_msqid_to_user(). The global message queue semaphore and spinlock are obtained and released at the end. After the the message queue ID and the current process access permissions are validated, the message queue information is updated with the user provided data. Later, expunge_all() and ss_wakeup() are called to wake up all processes sleeping on the receiver and sender waiting queues of the message queue. This is because some receivers may now be excluded by stricter access permissions and some senders may now be able to send the message due to an increased queue size.

IPC_RMID

The global message queue semaphore is obtained and the global message queue spinlock is locked. After validating the message queue ID and the current task access permissions, freeque() is called to free the resources related to the message queue ID. The global message queue semaphore and spinlock are released.

sys_msgsnd()

sys_msgsnd() receives as parameters a message queue ID (`msqid`), a pointer to a buffer of type struct msg_msg (`msgp`), the size of the message to be sent (`msgsz`), and a flag indicating wait vs. not wait (`msgflg`). There are two task waiting queues and one message waiting queue associated with the message queue ID. If there is a task in the receiver waiting queue that is waiting for this message, then the message is delivered directly to the receiver, and the receiver is awakened. Otherwise, if there is enough space available in the message waiting queue, the message is saved in this queue. As a last resort, the sending task enqueues itself on the sender waiting queue. A more in−depth discussion of the operations performed by sys_msgsnd() follows:

1. Validates the user buffer address and the message type, then invokes load_msg() to load the contents of the user message into a temporary object `msg` of type struct msg_msg. The message type and message size fields of `msg` are also initialized.

2. Locks the global message queue spinlock and gets the message queue descriptor associated with the message queue ID. If no such message queue exists, returns EINVAL.
3. Invokes ipc_checkid() (via msg_checkid())to verify that the message queue ID is valid and calls ipcperms() to check the calling process' access permissions.
4. Checks the message size and the space left in the message waiting queue to see if there is enough room to store the message. If not, the following substeps are performed:
    1. If IPC_NOWAIT is specified in `msgflg` the global message queue spinlock is unlocked, the memory resources for the message are freed, and EAGAIN is returned.
    2. Invokes ss_add() to enqueue the current task in the sender waiting queue. It also unlocks the global message queue spinlock and invokes schedule() to put the current task to sleep.
    3. When awakened, obtains the global spinlock again and verifies that the message queue ID is still valid. If the message queue ID is not valid, ERMID is returned.
    4. Invokes ss_del() to remove the sending task from the sender waiting queue. If there is any signal pending for the task, sys_msgsnd() unlocks the global spinlock, invokes free_msg() to free the message buffer, and returns EINTR. Otherwise, the function goes back to check again whether there is enough space in the message waiting queue.
5. Invokes pipelined_send() to try to send the message to the waiting receiver directly.
6. If there is no receiver waiting for this message, enqueues `msg` into the message waiting queue(msq−>q_messages). Updates the `q_cbytes` and the `q_qnum` fields of the message queue descriptor, as well as the global variables `msg_bytes` and `msg_hdrs`, which indicate the total number of bytes used for messages and the total number of messages system wide.
7. If the message has been successfully sent or enqueued, updates the `q_lspid` and the `q_stime` fields of the message queue descriptor and releases the global message queue spinlock.

sys_msgrcv()

The sys_msgrcv() function receives as parameters a message queue ID (`msqid`), a pointer to a buffer of type msg_msg (`msgp`), the desired message size(`msgsz`), the message type (`msgtyp`), and the flags (`msgflg`). It searches the message waiting queue associated with the message queue ID, finds the first message in the queue which matches the request type, and copies it into the given user buffer. If no such message is found in the message waiting queue, the requesting task is enqueued into the receiver waiting queue until the desired message is available. A more in−depth discussion of the operations performed by sys_msgrcv() follows:

1. First, invokes convert_mode() to derive the search mode from `msgtyp`. sys_msgrcv() then locks the global message queue spinlock and obtains the message queue descriptor associated with the message queue ID. If no such message queue exists, it returns EINVAL.
2. Checks whether the current task has the correct permissions to access the message queue.
3. Starting from the first message in the message waiting queue, invokes testmsg() to check whether the message type matches the required type. sys_msgrcv() continues searching until a matched message is found or the whole waiting queue is exhausted. If the search mode is SEARCH_LESSEQUAL, then the first message on the queue with the lowest type less than or equal to `msgtyp` is searched.
4. If a message is found, sys_msgrcv() performs the following substeps:
    1. If the message size is larger than the desired size and `msgflg` indicates no error allowed, unlocks the global message queue spinlock and returns E2BIG.
    2. Removes the message from the message waiting queue and updates the message queue statistics.
    3. Wakes up all tasks sleeping on the senders waiting queue. The removal of a message from the queue in the previous step makes it possible for one of the senders to progress. Goes to the last step
5. If no message matching the receivers criteria is found in the message waiting queue, then `msgflg` is

checked. If IPC_NOWAIT is set, then the global message queue spinlock is unlocked and ENOMSG is returned. Otherwise, the receiver is enqueued on the receiver waiting queue as follows:

1. A msg_receiver data structure `msr` is allocated and is added to the head of waiting queue.
2. The `r_tsk` field of `msr` is set to current task.
3. The `r_msgtype` and `r_mode` fields are initialized with the desired message type and mode respectively.
4. If `msgflg` indicates MSG_NOERROR, then the r_maxsize field of `msr` is set to be the value of `msgsz` otherwise it is set to be INT_MAX.
5. The `r_msg` field is initialized to indicate that no message has been received yet.
6. After the initialization is complete, the status of the receiving task is set to TASK_INTERRUPTIBLE, the global message queue spinlock is unlocked, and schedule() is invoked.

6. After the receiver is awakened, the `r_msg` field of `msr` is checked. This field is used to store the pipelined message or in the case of an error, to store the error status. If the `r_msg` field is filled with the desired message, then go to the last step Otherwise, the global message queue spinlock is locked again.
7. After obtaining the spinlock, the `r_msg` field is re−checked to see if the message was received while waiting for the spinlock. If the message has been received, the last step occurs.
8. If the `r_msg` field remains unchanged, then the task was awakened in order to retry. In this case, `msr` is dequeued. If there is a signal pending for the task, then the global message queue spinlock is unlocked and EINTR is returned. Otherwise, the function needs to go back and retry.
9. If the `r_msg` field shows that an error occurred while sleeping, the global message queue spinlock is unlocked and the error is returned.
10. After validating that the address of the user buffer `msp` is valid, message type is loaded into the `mtype` field of `msp`,and store_msg() is invoked to copy the message contents to the `mtext` field of `msp`. Finally the memory for the message is freed by function free_msg().

Message Specific Structures

Data structures for message queues are defined in msg.c.

struct msg_queue

```
/* one msq_queue structure for each present queue on the system */
struct msg_queue {
        struct kern_ipc_perm q_perm;
        time_t q_stime;                 /* last msgsnd time */
        time_t q_rtime;                 /* last msgrcv time */
        time_t q_ctime;                 /* last change time */
        unsigned long q_cbytes;         /* current number of bytes on queue */
        unsigned long q_qnum;           /* number of messages in queue */
        unsigned long q_qbytes;         /* max number of bytes on queue */
        pid_t q_lspid;                  /* pid of last msgsnd */
        pid_t q_lrpid;                  /* last receive pid */

        struct list_head q_messages;
        struct list_head q_receivers;
        struct list_head q_senders;
};
```

struct msg_msg

```
/* one msg_msg structure for each message */
struct msg_msg {
        struct list_head m_list;
        long  m_type;
        int m_ts;           /* message text size */
        struct msg_msgseg* next;
        /* the actual message follows immediately */
};
```

struct msg_msgseg

```
/* message segment for each message */
struct msg_msgseg {
        struct msg_msgseg* next;
        /* the next part of the message follows immediately */
};
```

struct msg_sender

```
/* one msg_sender for each sleeping sender */
struct msg_sender {
        struct list_head list;
        struct task_struct* tsk;
};
```

struct msg_receiver

```
/* one msg_receiver structure for each sleeping receiver */
struct msg_receiver {
        struct list_head r_list;
        struct task_struct* r_tsk;

        int r_mode;
        long r_msgtype;
        long r_maxsize;

        struct msg_msg* volatile r_msg;
};
```

struct msqid64_ds

```
struct msqid64_ds {
        struct ipc64_perm msg_perm;
        __kernel_time_t msg_stime;      /* last msgsnd time */
        unsigned long   __unused1;
        __kernel_time_t msg_rtime;      /* last msgrcv time */
        unsigned long   __unused2;
        __kernel_time_t msg_ctime;      /* last change time */
        unsigned long   __unused3;
```

```
        unsigned long  msg_cbytes;      /* current number of bytes on queue */
        unsigned long  msg_qnum;        /* number of messages in queue */
        unsigned long  msg_qbytes;      /* max number of bytes on queue */
        __kernel_pid_t msg_lspid;       /* pid of last msgsnd */
        __kernel_pid_t msg_lrpid;       /* last receive pid */
        unsigned long  __unused4;
        unsigned long  __unused5;
};
```

struct msqid_ds

```
 struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg *msg_first;          /* first message on queue,unused  */
        struct msg *msg_last;           /* last message in queue,unused */
        __kernel_time_t msg_stime;      /* last msgsnd time */
        __kernel_time_t msg_rtime;      /* last msgrcv time */
        __kernel_time_t msg_ctime;      /* last change time */
        unsigned long  msg_lcbytes;     /* Reuse junk fields for 32 bit */
        unsigned long  msg_lqbytes;     /* ditto */
        unsigned short msg_cbytes;      /* current number of bytes on queue */
        unsigned short msg_qnum;        /* number of messages in queue */
        unsigned short msg_qbytes;      /* max number of bytes on queue */
        __kernel_ipc_pid_t msg_lspid;   /* pid of last msgsnd */
        __kernel_ipc_pid_t msg_lrpid;   /* last receive pid */
};
```

msg_setbuf

```
struct msq_setbuf {
        unsigned long   qbytes;
        uid_t           uid;
        gid_t           gid;
        mode_t          mode;
};
```

Message Support Functions

newque()

newque() allocates the memory for a new message queue descriptor (struct msg_queue) and then calls ipc_addid(), which reserves a message queue array entry for the new message queue descriptor. The message queue descriptor is initialized as follows:

- The kern_ipc_perm structure is initialized.
- The `q_stime` and `q_rtime` fields of the message queue descriptor are initialized as 0. The `q_ctime` field is set to be CURRENT_TIME.
- The maximum number of bytes allowed in this queue message (`q_qbytes`) is set to be MSGMNB, and the number of bytes currently used by the queue (`q_cbytes`) is initialized as 0.
- The message waiting queue (`q_messages`), the receiver waiting queue (`q_receivers`), and the sender waiting queue (`q_senders`) are each initialized as empty.

All the operations following the call to ipc_addid() are performed while holding the global message queue spinlock. After unlocking the spinlock, newque() calls msg_buildid(), which maps directly to ipc_buildid(). ipc_buildid() uses the index of the message queue descriptor to create a unique message queue ID that is then returned to the caller of newque().

freeque()

When a message queue is going to be removed, the freeque() function is called. This function assumes that the global message queue spinlock is already locked by the calling function. It frees all kernel resources associated with that message queue. First, it calls ipc_rmid() (via msg_rmid()) to remove the message queue descriptor from the array of global message queue descriptors. Then it calls expunge_all to wake up all receivers and ss_wakeup() to wake up all senders sleeping on this message queue. Later the global message queue spinlock is released. All messages stored in this message queue are freed and the memory for the message queue descriptor is freed.

ss_wakeup()

ss_wakeup() wakes up all the tasks waiting in the given message sender waiting queue. If this function is called by freeque(), then all senders in the queue are dequeued.

ss_add()

ss_add() receives as parameters a message queue descriptor and a message sender data structure. It fills the `tsk` field of the message sender data structure with the current process, changes the status of current process to TASK_INTERRUPTIBLE, then inserts the message sender data structure at the head of the sender waiting queue of the given message queue.

ss_del()

If the given message sender data structure (`mss`) is still in the associated sender waiting queue, then ss_del() removes `mss` from the queue.

expunge_all()

expunge_all() receives as parameters a message queue descriptor(`msq`) and an integer value (`res`) indicating the reason for waking up the receivers. For each sleeping receiver associated with `msq`, the `r_msg` field is set to the indicated wakeup reason (`res`), and the associated receiving task is awakened. This function is called when a message queue is removed or a message control operation has been performed.

load_msg()

When a process sends a message, the sys_msgsnd() function first invokes the load_msg() function to load the message from user space to kernel space. The message is represented in kernel memory as a linked list of data blocks. Associated with the first data block is a msg_msg structure that describes the overall message. The datablock associated with the msg_msg structure is limited to a size of DATA_MSG_LEN. The data block and the structure are allocated in one contiguous memory block that can be as large as one page in memory. If the full message will not fit into this first data block, then additional data blocks are allocated and are organized into a linked list. These additional data blocks are limited to a size of DATA_SEG_LEN, and each include an associated msg_msgseg) structure. The msg_msgseg structure and the associated data block are allocated in one contiguous memory block that can be as large as one page in memory. This function returns the address of the new msg_msg structure on success.

store_msg()

The store_msg() function is called by sys_msgrcv() to reassemble a received message into the user space buffer provided by the caller. The data described by the msg_msg structure and any msg_msgseg structures are sequentially copied to the user space buffer.

free_msg()

The free_msg() function releases the memory for a message data structure msg_msg, and the message segments.

convert_mode()

convert_mode() is called by sys_msgrcv(). It receives as parameters the address of the specified message type (`msgtyp`) and a flag (`msgflg`). It returns the search mode to the caller based on the value of `msgtyp` and `msgflg`. If `msgtyp` is null, then SEARCH_ANY is returned. If `msgtyp` is less than 0, then `msgtyp` is set to it's absolute value and SEARCH_LESSEQUAL is returned. If MSG_EXCEPT is specified in `msgflg`, then SEARCH_NOTEQUAL is returned. Otherwise SEARCH_EQUAL is returned.

testmsg()

The testmsg() function checks whether a message meets the criteria specified by the receiver. It returns 1 if one of the following conditions is true:

- The search mode indicates searching any message (SEARCH_ANY).
- The search mode is SEARCH_LESSEQUAL and the message type is less than or equal to desired type.
- The search mode is SEARCH_EQUAL and the message type is the same as desired type.
- Search mode is SEARCH_NOTEQUAL and the message type is not equal to the specified type.

pipelined_send()

pipelined_send() allows a process to directly send a message to a waiting receiver rather than deposit the message in the associated message waiting queue. The testmsg() function is invoked to find the first receiver which is waiting for the given message. If found, the waiting receiver is removed from the receiver waiting queue, and the associated receiving task is awakened. The message is stored in the `r_msg` field of the receiver, and 1 is returned. In the case where no receiver is waiting for the message, 0 is returned.

In the process of searching for a receiver, potential receivers may be found which have requested a size that is too small for the given message. Such receivers are removed from the queue, and are awakened with an error status of E2BIG, which is stored in the `r_msg` field. The search then continues until either a valid receiver is found, or the queue is exhausted.

copy_msqid_to_user()

copy_msqid_to_user() copies the contents of a kernel buffer to the user buffer. It receives as parameters a user buffer, a kernel buffer of type msqid64_ds, and a version flag indicating the new IPC version vs. the old IPC version. If the version flag equals IPC_64, then copy_to_user() is invoked to copy from the kernel buffer to the user buffer directly. Otherwise a temporary buffer of type struct msqid_ds is initialized, and the kernel data is translated to this temporary buffer. Later copy_to_user() is called to copy the contents of the the temporary buffer to the user buffer.

copy_msqid_from_user()

The function copy_msqid_from_user() receives as parameters a kernel message buffer of type struct msq_setbuf, a user buffer and a version flag indicating the new IPC version vs. the old IPC version. In the case of the new IPC version, copy_from_user() is called to copy the contents of the user buffer to a temporary buffer of type msqid64_ds. Then, the `qbytes`,`uid`, `gid`, and `mode` fields of the kernel buffer are filled with the values of the corresponding fields from the temporary buffer. In the case of the old IPC version, a temporary buffer of type struct msqid_ds is used instead.

5.3 Shared Memory

Shared Memory System Call Interfaces

sys_shmget()

The entire call to sys_shmget() is protected by the global shared memory semaphore.

In the case where a new shared memory segment must be created, the newseg() function is called to create and initialize a new shared memory segment. The ID of the new segment is returned to the caller.

In the case where a key value is provided for an existing shared memory segment, the corresponding index in the shared memory descriptors array is looked up, and the parameters and permissions of the caller are verified before returning the shared memory segment ID. The look up operation and verification are performed while the global shared memory spinlock is held.

sys_shmctl()

IPC_INFO

A temporary shminfo64 buffer is loaded with system−wide shared memory parameters and is copied out to user space for access by the calling application.

SHM_INFO

The global shared memory semaphore and the global shared memory spinlock are held while gathering system−wide statistical information for shared memory. The shm_get_stat() function is called to calculate both the number of shared memory pages that are resident in memory and the number of shared memory pages that are swapped out. Other statistics include the total number of shared memory pages and the number of shared memory segments in use. The counts of `swap_attempts` and `swap_successes` are hard−coded to zero. These statistics are stored in a temporary shm_info buffer and copied out to user space for the calling application.

SHM_STAT, IPC_STAT

For SHM_STAT and IPC_STATA, a temporary buffer of type struct shmid64_ds is initialized, and the global shared memory spinlock is locked.

For the SHM_STAT case, the shared memory segment ID parameter is expected to be a straight index (i.e. 0 to n where n is the number of shared memory IDs in the system). After validating the index, ipc_buildid() is called (via shm_buildid()) to convert the index into a shared memory ID. In the passing case of SHM_STAT, the shared memory ID will be the return value. Note that this is an undocumented feature, but is maintained for the ipcs(8) program.

For the IPC_STAT case, the shared memory segment ID parameter is expected to be an ID that was generated by a call to shmget(). The ID is validated before proceeding. In the passing case of IPC_STAT, 0 will be the return value.

For both SHM_STAT and IPC_STAT, the access permissions of the caller are verified. The desired statistics are loaded into the temporary buffer and then copied out to the calling application.

SHM_LOCK, SHM_UNLOCK

After validating access permissions, the global shared memory spinlock is locked, and the shared memory segment ID is validated. For both SHM_LOCK and SHM_UNLOCK, shmem_lock() is called to perform the function. The parameters for shmem_lock() identify the function to be performed.

IPC_RMID

During IPC_RMID the global shared memory semaphore and the global shared memory spinlock are held throughout this function. The Shared Memory ID is validated, and then if there are no current attachments, shm_destroy() is called to destroy the shared memory segment. Otherwise, the SHM_DEST flag is set to mark it for destruction, and the IPC_PRIVATE flag is set to prevent other processes from being able to reference the shared memory ID.

IPC_SET

After validating the shared memory segment ID and the user access permissions, the `uid`, `gid`, and `mode` flags of the shared memory segment are updated with the user data. The `shm_ctime` field is also updated. These changes are made while holding the global shared memory semaphore and the global share memory spinlock.

sys_shmat()

sys_shmat() takes as parameters, a shared memory segment ID, an address at which the shared memory segment should be attached(`shmaddr`), and flags which will be described below.

If `shmaddr` is non−zero, and the SHM_RND flag is specified, then `shmaddr` is rounded down to a multiple of SHMLBA. If `shmaddr` is not a multiple of SHMLBA and SHM_RND is not specified, then EINVAL is returned.

The access permissions of the caller are validated and the `shm_nattch` field for the shared memory segment is incremented. Note that this increment guarantees that the attachment count is non−zero and prevents the shared memory segment from being destroyed during the process of attaching to the segment. These operations are performed while holding the global shared memory spinlock.

The do_mmap() function is called to create a virtual memory mapping to the shared memory segment pages. This is done while holding the `mmap_sem` semaphore of the current task. The MAP_SHARED flag is passed to do_mmap(). If an address was provided by the caller, then the MAP_FIXED flag is also passed to do_mmap(). Otherwise, do_mmap() will select the virtual address at which to map the shared memory segment.

NOTE shm_inc() will be invoked within the do_mmap() function call via the `shm_file_operations` structure. This function is called to set the PID, to set the current time, and to increment the number of attachments to this shared memory segment.

After the call to do_mmap(), the global shared memory semaphore and the global shared memory spinlock are both obtained. The attachment count is then decremented. The the net change to the attachment count is 1 for a call to shmat() because of the call to shm_inc(). If, after decrementing the attachment count, the resulting count is found to be zero, and if the segment is marked for destruction (SHM_DEST), then shm_destroy() is called to release the shared memory segment resources.

Finally, the virtual address at which the shared memory is mapped is returned to the caller at the user specified address. If an error code had been returned by do_mmap(), then this failure code is passed on as the return value for the system call.

sys_shmdt()

The global shared memory semaphore is held while performing sys_shmdt(). The `mm_struct` of the current process is searched for the `vm_area_struct` associated with the shared memory address. When it is found, do_munmap() is called to undo the virtual address mapping for the shared memory segment.

Note also that do_munmap() performs a call−back to shm_close(), which performs the shared−memory book keeping functions, and releases the shared memory segment resources if there are no other attachments.

sys_shmdt() unconditionally returns 0.

Shared Memory Support Structures

struct shminfo64

```
struct shminfo64 {
        unsigned long   shmmax;
        unsigned long   shmmin;
        unsigned long   shmmni;
        unsigned long   shmseg;
        unsigned long   shmall;
        unsigned long   __unused1;
        unsigned long   __unused2;
        unsigned long   __unused3;
        unsigned long   __unused4;
```

```
};
```

struct shm_info

```
struct shm_info {
        int used_ids;
        unsigned long shm_tot;  /* total allocated shm */
        unsigned long shm_rss;  /* total resident shm */
        unsigned long shm_swp;  /* total swapped shm */
        unsigned long swap_attempts;
        unsigned long swap_successes;
};
```

struct shmid_kernel

```
struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm    shm_perm;
        struct file *           shm_file;
        int                     id;
        unsigned long           shm_nattch;
        unsigned long           shm_segsz;
        time_t                  shm_atim;
        time_t                  shm_dtim;
        time_t                  shm_ctim;
        pid_t                   shm_cprid;
        pid_t                   shm_lprid;
};
```

struct shmid64_ds

```
struct shmid64_ds {
        struct ipc64_perm       shm_perm;       /* operation perms */
        size_t                  shm_segsz;      /* size of segment (bytes) */
        __kernel_time_t         shm_atime;      /* last attach time */
        unsigned long           __unused1;
        __kernel_time_t         shm_dtime;      /* last detach time */
        unsigned long           __unused2;
        __kernel_time_t         shm_ctime;      /* last change time */
        unsigned long           __unused3;
        __kernel_pid_t          shm_cpid;       /* pid of creator */
        __kernel_pid_t          shm_lpid;       /* pid of last operator */
        unsigned long           shm_nattch;     /* no. of current attaches */
        unsigned long           __unused4;
        unsigned long           __unused5;
};
```

struct shmem_inode_info

```
struct shmem_inode_info {
        spinlock_t      lock;
        unsigned long   max_index;
```

```
        swp_entry_t     i_direct[SHMEM_NR_DIRECT]; /* for the first blocks */
        swp_entry_t   **i_indirect; /* doubly indirect blocks */
        unsigned long   swapped;
        int             locked;     /* into memory */
        struct list_head        list;
};
```

Shared Memory Support Functions

newseg()

The newseg() function is called when a new shared memory segment needs to be created. It acts on three parameters for the new segment the key, the flag, and the size. After validating that the size of the shared memory segment to be created is between SHMMIN and SHMMAX and that the total number of shared memory segments does not exceed SHMALL, it allocates a new shared memory segment descriptor. The shmem_file_setup() function is invoked later to create an unlinked file of type tmpfs. The returned file pointer is saved in the `shm_file` field of the associated shared memory segment descriptor. The files size is set to be the same as the size of the segment. The new shared memory segment descriptor is initialized and inserted into the global IPC shared memory descriptors array. The shared memory segment ID is created by shm_buildid() (via ipc_buildid()). This segment ID is saved in the `id` field of the shared memory segment descriptor, as well as in the `i_ino` field of the associated inode. In addition, the address of the shared memory operations defined in structure `shm_file_operation` is stored in the associated file. The value of the global variable `shm_tot`, which indicates the total number of shared memory segments system wide, is also increased to reflect this change. On success, the segment ID is returned to the caller application.

shm_get_stat()

shm_get_stat() cycles through all of the shared memory structures, and calculates the total number of memory pages in use by shared memory and the total number of shared memory pages that are swapped out. There is a file structure and an inode structure for each shared memory segment. Since the required data is obtained via the inode, the spinlock for each inode structure that is accessed is locked and unlocked in sequence.

shmem_lock()

shmem_lock() receives as parameters a pointer to the shared memory segment descriptor and a flag indicating lock vs. unlock.The locking state of the shared memory segment is stored in an associated inode. This state is compared with the desired locking state; shmem_lock() simply returns if they match.

While holding the semaphore of the associated inode, the locking state of the inode is set. The following list of items occur for each page in the shared memory segment:

- find_lock_page() is called to lock the page (setting PG_locked) and to increment the reference count of the page. Incrementing the reference count assures that the shared memory segment remains locked in memory throughout this operation.
- If the desired state is locked, then PG_locked is cleared, but the reference count remains incremented.
- If the desired state is unlocked, then the reference count is decremented twice once for the current reference, and once for the existing reference which caused the page to remain locked in memory. Then PG_locked is cleared.

shm_destroy()

During shm_destroy() the total number of shared memory pages is adjusted to account for the removal of the shared memory segment. ipc_rmid() is called (via shm_rmid()) to remove the Shared Memory ID. shmem_lock is called to unlock the shared memory pages, effectively decrementing the reference counts to zero for each page. fput() is called to decrement the usage counter `f_count` for the associated file object, and if necessary, to release the file object resources. kfree() is called to free the shared memory segment descriptor.

shm_inc()

shm_inc() sets the PID, sets the current time, and increments the number of attachments for the given shared memory segment. These operations are performed while holding the global shared memory spinlock.

shm_close()

shm_close() updates the `shm_lprid` and the `shm_dtim` fields and decrements the number of attached shared memory segments. If there are no other attachments to the shared memory segment, then shm_destroy() is called to release the shared memory segment resources. These operations are all performed while holding both the global shared memory semaphore and the global shared memory spinlock.

shmem_file_setup()

The function shmem_file_setup() sets up an unlinked file living in the tmpfs file system with the given name and size. If there are enough systen memory resource for this file, it creates a new dentry under the mount root of tmpfs, and allocates a new file descriptor and a new inode object of tmpfs type. Then it associates the new dentry object with the new inode object by calling d_instantiate() and saves the address of the dentry object in the file descriptor. The `i_size` field of the inode object is set to be the file size and the `i_nlink` field is set to be 0 in order to mark the inode unlinked. Also, shmem_file_setup() stores the address of the `shmem_file_operations` structure in the `f_op` field, and initializes `f_mode` and `f_vfsmnt` fields of the file descriptor properly. The function shmem_truncate() is called to complete the initialization of the inode object. On success, shmem_file_setup() returns the new file descriptor.

5.4 Linux IPC Primitives

Generic Linux IPC Primitives used with Semaphores, Messages,and Shared Memory

The semaphores, messages, and shared memory mechanisms of Linux are built on a set of common primitives. These primitives are described in the sections below.

ipc_alloc()

If the memory allocation is greater than PAGE_SIZE, then vmalloc() is used to allocate memory. Otherwise, kmalloc() is called with GFP_KERNEL to allocate the memory.

ipc_addid()

When a new semaphore set, message queue, or shared memory segment is added, ipc_addid() first calls grow_ary() to insure that the size of the corresponding descriptor array is sufficiently large for the system maximum. The array of descriptors is searched for the first unused element. If an unused element is found, the count of descriptors which are in use is incremented. The kern_ipc_perm structure for the new resource descriptor is then initialized, and the array index for the new descriptor is returned. When ipc_addid() succeeds, it returns with the global spinlock for the given IPC type locked.

ipc_rmid()

ipc_rmid() removes the IPC descriptor from the the global descriptor array of the IPC type, updates the count of IDs which are in use, and adjusts the maximum ID in the corresponding descriptor array if necessary. A pointer to the IPC descriptor associated with given IPC ID is returned.

ipc_buildid()

ipc_buildid() creates a unique ID to be associated with each descriptor within a given IPC type. This ID is created at the time a new IPC element is added (e.g. a new shared memory segment or a new semaphore set). The IPC ID converts easily into the corresponding descriptor array index. Each IPC type maintains a sequence number which is incremented each time a descriptor is added. An ID is created by multiplying the sequence number with SEQ_MULTIPLIER and adding the product to the descriptor array index. The sequence number used in creating a particular IPC ID is then stored in the corresponding descriptor. The existence of the sequence number makes it possible to detect the use of a stale IPC ID.

ipc_checkid()

ipc_checkid() divides the given IPC ID by the SEQ_MULTIPLIER and compares the quotient with the seq value saved corresponding descriptor. If they are equal, then the IPC ID is considered to be valid and 1 is returned. Otherwise, 0 is returned.

grow_ary()

grow_ary() handles the possibility that the maximum (tunable) number of IDs for a given IPC type can be dynamically changed. It enforces the current maximum limit so that it is no greater than the permanent system limit (IPCMNI) and adjusts it down if necessary. It also insures that the existing descriptor array is large enough. If the existing array size is sufficiently large, then the current maximum limit is returned. Otherwise, a new larger array is allocated, the old array is copied into the new array, and the old array is freed. The corresponding global spinlock is held when updating the descriptor array for the given IPC type.

ipc_findkey()

ipc_findkey() searches through the descriptor array of the specified ipc_ids object, and searches for the specified key. Once found, the index of the corresponding descriptor is returned. If the key is not found, then −1 is returned.

ipcperms()

ipcperms() checks the user, group, and other permissions for access to the IPC resources. It returns 0 if permission is granted and −1 otherwise.

ipc_lock()

ipc_lock() takes an IPC ID as one of its parameters. It locks the global spinlock for the given IPC type, and returns a pointer to the descriptor corresponding to the specified IPC ID.

ipc_unlock()

ipc_unlock() releases the global spinlock for the indicated IPC type.

ipc_lockall()

ipc_lockall() locks the global spinlock for the given IPC mechanism (i.e. shared memory, semaphores, and messaging).

ipc_unlockall()

ipc_unlockall() unlocks the global spinlock for the given IPC mechanism (i.e. shared memory, semaphores, and messaging).

ipc_get()

ipc_get() takes a pointer to a particular IPC type (i.e. shared memory, semaphores, or message queues) and a descriptor ID, and returns a pointer to the corresponding IPC descriptor. Note that although the descriptors for each IPC type are of different data types, the common kern_ipc_perm structure type is embedded as the first entity in every case. The ipc_get() function returns this common data type. The expected model is that ipc_get() is called through a wrapper function (e.g. shm_get()) which casts the data type to the correct descriptor data type.

ipc_parse_version()

ipc_parse_version() removes the IPC_64 flag from the command if it is present and returns either IPC_64 or IPC_OLD.

Generic IPC Structures used with Semaphores,Messages, and Shared Memory

The semaphores, messages, and shared memory mechanisms all make use of the following common structures:

struct kern_ipc_perm

Each of the IPC descriptors has a data object of this type as the first element. This makes it possible to access any descriptor from any of the generic IPC functions using a pointer of this data type.

```
/* used by in-kernel data structures */
struct kern_ipc_perm {
    key_t key;
    uid_t uid;
    gid_t gid;
    uid_t cuid;
    gid_t cgid;
    mode_t mode;
    unsigned long seq;
};
```

struct ipc_ids

The ipc_ids structure describes the common data for semaphores, message queues, and shared memory. There are three global instances of this data structure–– `semid_ds`, `msgid_ds` and `shmid_ds`–– for semaphores, messages and shared memory respectively. In each instance, the `sem` semaphore is used to protect access to the structure. The `entries` field points to an IPC descriptor array, and the `ary` spinlock protects access to this array. The `seq` field is a global sequence number which will be incremented when a new IPC resource is created.

```
struct ipc_ids {
    int size;
    int in_use;
    int max_id;
    unsigned short seq;
    unsigned short seq_max;
    struct semaphore sem;
    spinlock_t ary;
    struct ipc_id* entries;
};
```

struct ipc_id

An array of struct ipc_id exists in each instance of the ipc_ids structure. The array is dynamically allocated and may be replaced with larger array by grow_ary() as required. The array is sometimes referred to as the descriptor array, since the kern_ipc_perm data type is used as the common descriptor data type by the IPC generic functions.

```
struct ipc_id {
    struct kern_ipc_perm* p;
};
```