Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Computer Science

Jaroslav Šoltýs

Linux Kernel 2.6 Documentation

Master thesis
Thesis advisor: RNDr. Jaroslav Janáček

Bratislava 2006

By this I declare that I wrote this master thesis by myself, only with the help of the referenced literature and internet resources, under the supervision of my thesis advisor.

Bratislava, 2006 Jaroslav Šoltýs

Abstract

This thesis is a study of internal working and interfaces of Linux kernel version 2.6. It emphasizes differences towards kernel 2.4 and it tries to introduce this problems to reader knowledgable only in common principles of operating systems.

The thesis starts with briefing of kernel subsystems and continues with deeper analysis of important issues like various locking, semaphores and other synchronization mechanisms, suitable data structures, interrupt handling, atomic operations, device driver model and memory allocation.

Process management and new scheduler with support for multiprocessing and NUMA architectures and preemption are explored deeply. The core of the work is concluded with description of loadable kernel module creation together with an exemplar module.

Preface

Fall 2003 brought many testing version of Linux kernel 2.6 and the final 2.6.0 was released on December 17th 2003. A lot of new interesting features and improved hardware support were teasing user to try it.

Linux now scales pretty well on both ends: embedded platforms and large servers with either SMP or NUMA support. The first has been achieved by merging ucLinux (branch of Linux for embedded platforms) to main kernel, while the new scheduler written during 2.5 development branch offers now preemption inside kernel and is able to take full use of multiple CPUs. This new facts (together with unified driver model) increase demands on device driver writers' know-how.

There was very little documentation to be found concerning this new kernel at the time I started working on this thesis. I did indeed start before the stable 2.6.0 came out. Also the documentation for older version was incosistent and scattered throughout the whole web. My task was to study 2.6 kernel (with special attention to new features in 2.6) and write documentation, that could help kernel beginners.

Reader is expected to have some preliminary knowledge about operating systems and some basic understanding of hardware.

Thesis Contents

Chapter 1: An Informal Introduction to Operating Systems describes the basic ideas behind an operating system, shows user the difference between monolithic an mikrokernel and mentions Unix traditions. It also contains the roughest introduction to Linux kernel and it's development process.

Chapter 2: Subsystems Overview lists subsystems and describes each one with particular attention to new features introduced in 2.6.

Chapter 3: Synchronization and Workflow Concurrency takes care about locks, semaphores, mutexes and other synchronisation entities. Issues with preemption are also mentioned, since existing locking mechanisms has been

improved to cope with these issues. Also one very interesting mechanism called read-copy-update is described. The last issues in this chapter are per-CPU variables, disabling local interrupts and memory barriers.

Chapter 4: Linked Lists mentions linked lists, which are intensively used wherever possible. The complicated variation protected by RCU is also included.

Chapter 5: Memory Allocation deals with allocating and releasing memory using various approches, such as page allocation, slab allocation and virtual memory allocation.

Chapter 6: Handling Interrupts contain all necessary information on how to work with hardware interrupts and tasklet interface.

Chapter 7: Process management and Scheduling concerns in scheduler, process life cycle and related issues.

Chapter 8: New Driver Model in 2.6 describes kernel object architecture and sysfs hierarchy, interface for device driver writers and user space event delivery mechanism.

Chapter 9: Common Routines and Helper Macros will introduce various macros to access user space memory, manipulate strings and some other common routines.

Chapter 10: Modules contains an exemplary loadable kernel module that exports its parameter using sysfs and explains various macros that could help with module development.

Typographic Conventions

Typographic conventions used in this thesis are simple and intuitive:

- `Typewriter`-like font denotes source code, `function()` or `MACRO` definitions and related things.
- *Notices* are emphasized with *slanted* font.
- **Warnings** and **important** words are printed in `bold`.
- Files and directories like `[kernel/sched.c]`in Linux kernel source tree are enclosed in square brackets.
- Include files are shown as in ordinary C source code: <`linux/list.h`>.

Contents

Chapter 1

An Informal Introduction to Operating Systems

Kernel of an operating system (OS) usually remains hidden beyond common user space programs and system utilities (until some serious error occurs). Let's start with describing what's happening inside your computer:

Your goal is to check for an e-mail. You run your favorite mail user agent and order to get work done for you.

> *Operating system uses notion of a* task*, in this case your mail user agent is a task for him. The word task was often used in past, but users of modern Unixes prefer to use word* process*. OS provides environment for this task/process by using some abstraction or emulation.* [1]

Instead of taking care for every detail, process let kernel to take care of some details, but he takes care of details of his part of the task. Back to the mail agent: there's no need to take care of your video card, your keyboard, your mouse, or details about how are the files stored on your hard drive.

> *Many operating systems exclude graphic routines from the kernel, leaving implementation to user space programs. Your mail agent runs in graphic mode, with colorful icons on buttons. It's common on Unix-like systems to use X server in order to run graphical applications, and you've probably noticed, that these graphic services are not provided by kernel.*

[1] Emulation is scarce in Linux

Where does the kernel fit in? It provides mechanisms for processes to communicate and to cooperate with each other. And sometimes also mechanisms to prevent processes to hurt each other.

Presume you have video conference running in small window next to your mail agent and a music player. How is it possible to run so many applications with only one processor ? Remember the abstraction we mentioned above:

> *Each running task is provided environment like it's the only one running and kernel takes care of switching between tasks so quickly that you can't see that there is always only one task running and the others are stopped.*

This was only a small list of kernel's duties. When you start your computer up, when you shut it down (except for killing him with power blackout), when you access your files, it's always there, watching over resources, time and memory assigned to each process, handling events from outer world, providing needful trifles, but it is also judge and police when some of your processes breaks the law.

1.1 Monolithic or Micro Kernel?

Linux was developed by Linus Torvalds in the beginning of 90s, at the time he used Minix, developed by prof. Andrew Tanenbaum. Minix (providing similar syscall interface like Unix systems) was internally designed as microkernel, while Linux is monolithic (with ability to load external modules).

Microkernels have never gained much success in commercial sphere with probably one exception: QNX[2]. They are more interesting from academic point of view.

Mikrokernels delegate responsibilities to user space processes instead of doing everything needed right in the kernel. This leads to simpler and cleaner internal architecture, modifying one subsystem consists of modifying a user space task and not the kernel. These tasks (called daemons or servers) don't run in privileged mode as kernel does, which should improve stability of the system.

Monolithic kernels proved to be much faster. Linux, Windows (from NT 3.51 upwards), various BSD branches belong to this family. The problem is that serving one syscall needs many context switches in microkernel model and modern processors can gain performance from utilizing cache. This cache is not used efficiently during context switch.

[2]QNX is a real time operating system and was intended for use with embedded hardware

Microkernel fans often accuse Linux of having dirty internal design. In reality kernel developers try to clean any mess in unstable branch[3].

Linux uses kernel threads and loadable kernel modules to increase flexibility. Kernel threads are meant only for slow things (for example swapping) because of two more context switches needed.

1.2 Concerning Traditional Unix Principles

Unix was created more than 3 decades ago and it's often thought of as father of modern operating systems. Original design offered two running processes, but this was overcome. Developed in cooperation between university and commercial sector, it soon included new ideas, such as TCP/IP networking. Its security increased from nothing to some of the most secure systems available today, including various security policy enhancements.

One of the most important Unix concepts is the notion of file. Files are not only regular files; block and character devices, pipes, named pipes (also called FIFOs), but symbolic links and TCP streams are also treated like special files. Pipe allow two programs to cooperate. Unix users sometimes mention the KISS principle: Keep it simple, small. The vast majority of traditional Unix tools does only one thing, but they do it perfectly. Instead of accessing directly keyboard and screen, Unix prefers to provide standard input, output (and error output), which can be redirected and used to automate user tasks.

Preemptible multitasking allows tasks to be switched independently on the 'willingness' of the task, virtual memory provides process more memory that could be allocated only using physical RAM. Processes don't need to care about other processes in memory[4].

Many of these ideas seem to be low-level, yet they provide strong instrument to developer. This is also often criticized, but building operating system on ground not solid and stable does not enforce good security.

1.3 Overview of the Linux Kernel

Linux started as 386-specific system, but soon developers from other architectures adopted the source and in time improved it to support 24 architectures. Writing

[3]Just have a look at the scheduler code in 2.6, all those routines and data structures, one's wishing to have an SMP machine to feel it running...

[4]This depends on hardware architecture, some Unixes running on embedded hardware does not have this feature, because it's impossible to implement it efficiently without hardware support.

portable code and running Linux on many platforms helped to find and fix many bugs. Most of the features are available on all architectures, but some (like address space protection needs MMU[5]) are not.

Linux starts with the boot of the kernel, then it starts its most important internal routines and continues with hardware detection. Since kernel 2.5 (the unstable branch) Linux has very good support for simultaneous multiprocessing. On architectures supporting CPU hot plug Linux could start and stop CPU anytime is needed and migrate assigned processes to other CPUs. In case you have multiprocessor machine, other CPUs are started after boot.

Devices drivers can be compiled statically into the kernel or dynamically loaded later. Not all Unix-like kernels support this, for example OpenBSD's kernel developers deny loadable kernel modules as bad security practice.

Device support is very broad. Linux provides good Hardware Abstraction Layer (HAL), but Hardware Emulation Layer (HEL) is almost non-existent with probably except for sound support.

Linux supports many file-systems, and in the time of writing this, Linux started supporting general user space file system implementation. Mounting remote WebDAV, FTP servers, SCP/SFTP or even local archive files and accessing them like local directories is a feature every desktop user could use. Although this could be standard feature in microkernel system, not many monolithic system support it now[6]. This problem was traditionally addressed in various user space libraries in desktop environments, but many Unix tools does not use these libraries.

It's important for an operating system to have some file system mounted after boot and Linux is not staying behind, it's root file system `/` could be either real file system from your hard disk drive, floppy, cdrom, USB key, or only created in RAM. Mounting NFS as root file system is also possible, therefore Linux is great for making disk-less network workstations.

Memory management takes care for virtual memory of all processes and also the kernel memory. Demand loading is used to load into memory only those pages which are really needed, paging is used to swap out pages which are supposed to by not so important and more important ones are loaded instead. Pages can be also shared to decrease memory footprint of a process and copied for each process only later, when needed.

Basic security on Linux is provided by trinity *user,group,other* while some file systems allows administrator to use Access Control Lists (ACL). User also has con-

[5]Memory Management Unit, part of the CPU

[6]Microsoft Windows supports IFS, which is Installable File System, but the developer fees are very high and almost nobody is using it

trol of his processes. Root is by default totally unrestricted[7]. Vanilla[8] Linux kernels now include SELinux extension to the default security model, BSD secure levels, networking security hooks and there are also any others (mostly rule-set based) extensions like GRSec, PaX, RSBAC or LOMAC available on Internet.

1.4 User Mode and Kernel Mode

The first CPUs allowed all programs to access all memory and I/O ports. At those times a memory leak in text editor could crash whole system. Later CPUs allowed to switch between two[9] privilege levels, so the OS designers decided to divide OS into two levels: kernel mode should work with critical data structures and manage access to the hardware, while user mode should not access to the hardware, only using calls to kernel. Crash in user mode would not cause crash of the whole system, it would only kill the crashed application (and perhaps dump the memory of the application for debugging purposes).

Switching from user mode to kernel mode could happen in using syscall or when an interrupt or an exception occurs.

1.5 About Kernel and Its Development

Past Linux version numbering consisted of three dot-separated numbers, the first one was the major version, the second one was the minor version. Even numbers in minor version field meant stable kernel, odd ones mean unstable i.e. development branch. The third field is number of patch. Patches are intended to fix some bug, they almost never introduce new feature in stable kernel.

Kernel 2.6.11 used new numbering scheme: development is done in 2.6 branch, the third number grows as new features are introduced. Patches that do not bring new features (they should be less than 100 lines in length) increase the fourth number. If the fourth number is zero, it's not written: first patch changes 2.6.16 to 2.6.16.1 and the next one to 2.6.16.2 and so on.

Submitting new patches and drivers became more formal than it was in the past. `[Documentation/SubmittingPatches]` contain all necessary information how and

[7]BSD systems sometimes force root to change permissions of file when he can't write to it, Linux root can write immediately

[8]Vanilla kernel is kernel from `[ftp://ftp.kernel.org]` without any 3rd party patches

[9]Some processors such as i386 allowed 4 privilege levels, but usually only 2 are used: 0 as the kernel mode and 3 as the user mode

to whom should be the patch sent. A signature scheme exists since the beginning of the SCO-Linux lawsuit to improve tracking who did what.

A patch arrives from developer to maintainer and if the maintainer decides not to dump the patch then he (she) signs it off:

```
Signed-off-by:  Random J. Developer rjd@example.org
```

This sign also means that the developer certifies the patch to be his own work and he has the right to submit it under open source license, or that the patch was provided to him (her) by someone who follows the forementioned rules. The list of maintainers is in file `[MAINTAINERS]` in the root of the kernel source package.

The Linux kernel also has its coding style defined in `[Document/CodingStyle]`, which differs from GNU coding standards.

Not to be forgotten is the `linux-kernel@vger.kernel.org` mailing list which contains kernel development discussion. The patches are also sent there as `Cc:` so this mailing list generates much traffic and many e-mails.

Chapter 2

Subsystems Overview

Kernel comprises of various subsystems; the code is split into subsystems logically based on the function provided (i.e. function concerning memory management, communication, file access...). These systems are bound together in monolithic kernels (such as Linux) and provide interfaces to each other. Imagine each subsystem as a library, but they sometimes call each other.

The most important kernel subsystems are process scheduler, memory manager, virtual file system (VFS) and networking, but many others are also essential to any functional kernel: system calls, interrupts and exceptions handling, various type of locks etc.

2.1 System Calls

System call is a way for process running in user space to request some functionality implemented in kernel. Kernel provides abstractions from hardware, manages shared resources and perform some of the many network-related functions.

Syscall is also one of two usual ways of switching between user and kernel mode, the other one is hardware driven interrupt or exception. Details of system calls are architecture specific. The interface for system calls is stable on each architecture, but it varies between CPU architectures. You can run old programs compiled long ago and they will still run.

System call (often abbreviated to *syscall*) allows user mode to perform privileged operations, if the checks in kernel allow to do it. Not everything in `libc` is implemented as syscall, for example memory allocation is solved through mapping Copy-On-Write (COW) pages from `[/dev/zero]` into address space of calling process.

2.2 Process Management and Execution

Process management is mostly about creating (forking) and terminating processes and switching between them. The first process is one of `[/sbin/init]`, `[/etc/init]`, `[/bin/init]` or `[/bin/sh]` (executed in this order). All other processes are forked from init or its children. Process management is also maintaining process states, such as running, sleeping, zombie process.

Since invention of multitasking (and multithreading) various approaches were tried. Preemptive multitasking was preferred to cooperative multitasking found in some systems. Linux support for multithreading can be simply described as creating more processes with shared virtual memory and some other resources, therefore terms process and thread are sometimes mixed up in various Linux documents.

The process scheduler decides which process should run when and how long should it run. It also allocates CPU for each process on machines with multiple CPUs. Linux kernel 2.6 uses various techniques to distinguish between processor-bound[1] and I/O-bound processes and gives a slight advantage to the I/0 bound ones, since they are supposed to be interactive. This should improve desktop responsiveness.

One of the biggest changes (and probably the most popular besides new hardware drivers) is the new O(1) scheduler written by Ingo Molnar, which will be described later in section 7.1.

Preemptible kernel aims to solve latency problem (from desktop user's point of view this can for example cause sound to crisp or) by allowing to switch tasks also inside kernel. Without preemption the code inside kernel runs until it completes its work (which is source of high latency) or it decides by itself to switch to other process (`schedule() function switches processes, this is usually done when kernel decides the thread will sleep until some event occurs`. If a higher priority task becomes runnable, the preemption will make it run. This way the higher priority task does its work when needed and not later.

Linux 2.6 also brings better support for symmetric multiprocessing (SMP) and Non-Uniform Memory Access (NUMA) architectures, which aims to increase performance by making multiple CPUs available to computer. One CPU can execute only one task at a time[2], so kernel must take care of properly assigning tasks to CPUs,

[1] A processor-bound process is process doing many calculations almost all the time, while I/O-bound process does very little computations but it's operations are I/O intensive (e.g. copying a file can be described as sequence of read/write operations)

[2] This is not completely true for SMT and multicore CPUs, but we define CPU as virtual CPU, not the physical one you can buy in a shop.

migrating tasks between CPUs in order to balance the load and achieve maximal performance. Some architectures also support CPU hot-plugging, where CPUs may be added or removed and kernel must handle this situations so that no task is lost with CPU removed.

2.3 Memory Management

Routines used for memory management could be divided into two groups, those taking care of the memory belonging to process and those managing kernel memory at the lowest level.

Basics of the memory management are covered by page management[3]. Many modern CPUs offer segmentation as alternative to paging, but Linux is not using this feature. Since pages (4 or 8KiB on most architectures) are too big for small kernel data structures, kernel includes slab allocator. This allows kernel allocate sub-page memory structures without fragmentation of memory.

Process memory management is based on pages as each page has its properties (permissions, read-only flag, present flag...). Pages can be shared between processes and they are the smallest elements that can be used to share or to be swapped out. Whenever the process switches from user mode to kernel mode, the kernel change Stack Pointer register to point to kernel stack of the process. This is needed because the user mode stack can be swapped out. Swapping is technique used to lay aside pages from process address space, it does not affect kernel pages.

The very first swapping implementation worked by monitoring amount of free memory and whenever certain threshold was passed, they put whole address space of affected process to disk. Pagination brought finer granularity and swapping out *pieces* of address space instead increased performance. Which pages can be swapped out ? Anonymous memory region pages of a process, modified pages (non-modified are usually mmaped from some file and they can be reclaimed from that file, this saves write operations to swap and also saves space) and IPC shared memory pages.

Linux supports multiple swap areas. Each area is divided into slots; whenever kernel needs to swap out page it tries to find slot in areas with bigger priority first. If there are more areas with the same priority, kernel uses them cyclically.

Page replacement has been extended in 2.6 to support one `kswapd` daemon per each CPU node. Previous 2.4 version offered only one daemon shared by all nodes, but Linux wishes to avoid penalty of freeing page on remote node.

[3]Systems without MMU don't have this feature, they usually also miss shared memory and process address space protection.

IA-32 architecture supports more than just 4 KiB page size, but Linux uses this pages by default only for mapping the actual kernel image. It is possible to use 4 MiB pages (referred to as *huge pages*) also for user space processes. The root of the implementation is a Huge TLB File System `hugetlbfs`, a pseudo file system based on ramfs. Any file that exists in that file system is backed by huge pages. Huge pages can be accessed using `shmget()` (shared memory) or `mmap()` (memory mapped file) on a file opened in this file system.

2.4 Interrupts and Exceptions

Interrupts are used to serve requests, usually from hardware devices (IRQs, interrupt requests), or from CPU itself (these are the exceptions, e.g. page fault, math coprocessor used and others invoked by various instructions) or syscall (on i386 architecture, interrupt `0x80`). IRQs are usually called when hardware device needs to notify kernel about some change, while CPU invoked interrupts (save for syscall, which has its own section in this chapter) are used to handle faults concerning process. Some of these faults can be intentional, e.g. page fault could indicate either that page is swapped out or that process tried to address the memory not belonging to him.

Common exceptions include Page Fault, General Protection Fault, Invalid Opcode, and various others concerning FPU, memory, debugging or simply division by zero. Some of these exceptions are used not only to handle errors: FPU stack contents needs not to be reloaded on each context switch (i.e. task switch from the OS point of view) so coprocessor exceptions are used to indicate that *it's time* to change the FPU stack content. Page faults are likewise used to bring in swapped out page.

2.5 Synchronization and Workflow Concurrency

Imagine two threads accessing a shared variable, e.g. counter of opened files. First thread opens a file and wants to increase the counter. so it retrieves its current value (for example 5), increases by value of one (the value is now 6) and stores the new value. A bit later the second thread opens a file and does the same operation. The final value of our counter is 7.

What if happens the following situation ? The first thread retrieves the value (5), then scheduler decides to give opportunity to the second one and it also retrieves the value and increases it by value of one and stores it back (6). Then the scheduler

resumes the first process, which increases the value it has read before and stores the new value (6). As we can see, the number of opened files is incorrect.

This is the problem of mutual exclusion. A common solution is usage of critical regions: critical region is a region of code and when one is being executed other threads/processes can't enter their critical regions. The other solutions is usage of atomic operations, (i.e. operations that can be executed so quickly that scheduler can't interfere). The first solution allows more complex operations to be done, but is also more difficult to be used correctly.

Critical regions are protected by locks. Whenever a thead enters the critical region, it locks the lock and when the other thread tries to enter the region and lock behind itself it finds out that it must wait for the lock to be unlocked (some locks prefer word 'sleep' to word 'wait').

The most simple locks used in Linux kernel are spinlocks. When a spinlock is locked, the other process trying to lock the spinlock enters a busy loop and continually checks whether the lock did not become unlocked. This behavior condemns to be used only to protect small critical regions which are supposed to be executed quickly and without any blocking.

Kernel developer with need of critical region that could block or will take longer time to execute should use semaphores (or newer mutexes). A semaphore can be locked by more than one lock-holders at the same time, this number is set at initialization. In case of being locked, the semaphore provides the advantage of switching CPU from busy loop to some other (hopefully) more important work. The disadvantage is caused by unsuccessful locker falling asleep and then taking some more time to wake up than spinlock.

The 2.6 branch of Linux kernel introduced two new notions: preemption and futexes. Futex stands for fast userspace mutex and preemption is the ability to switch threads also after they have entered kernel mode. These topics will be more deeply discussed in chapter 3.

2.6 Time and Timers

Keeping current time and date and executing tasks at given time (or periodically at some interval) is essential to modern computing. Preemptive process scheduling wouldn't work without correct use of timers.

Linux periodically updates current (system) time, time since system startup (uptime), it decides for each CPU whether the process running on this CPU has drained time allocated (scheduling timeslice, or quanta), updates resource usage

statistics (track the system load, profile kernel...) and make statistics for BSD process accounting and checks for elapsed software timers.

Software timers are based on `jiffies`, each timer has associated value of how much ticks in the future (added to `jiffies` at timer start). The jiffie itself is constant interval of time, but it's different on each architecture.

2.7 Signals and Inter-Process Communication

Sometimes processes need to cooperate with each other to achieve a goal, various mechanisms to solve this problem were introduced.

Signals are the oldest of all inter-process communication facilities, both user space processes and kernel are using them to send a short message to process or a group of processes. This message consists only of its identifier, i.e. integral number. No arguments are used besides this number.

Signals are used either to make process aware of events or to force process to deal with something. Process could specify signal handler which will handle the signal using `sigaction()` system call. Signals could be generated by other processes and sent to target process via system call, or by kernel (for example `SIGSEGV` sent to process trying to access beyond its address space). Signals can cause process to terminate, stop its execution or continue it again. Some signals could be blocked, `SIGKILL` and `SIGSTOP` can't.

Another inter-process communication facility common to all Unixes is pipe. System call `pipe()` returns two file descriptors. One is for reading only and the other for writing. According to POSIX standard, process must close one of them before communicating with the other one (half-duplex pipe), but Linux allows to read from one and write to another.[4] Writing and reading from pipe can be blocking call and cause the process to sleep.

FIFOs could be best described as extension to pipes; they're associated with a filename on the filesystem, therefore any process knowing the name can access this pipe, which implies the name 'named pipes'. They are created by issuing `mknod()` system call with mode `S_IFIFO`.

IPC in Linux was inspired by SystemV IPC: semaphores, message queues and shared memory. All these entities are created/joined using *key*. The same key used in different processes indicate that they want to share e.g. message queue. Convenient way to generate keys is the function `ftok()`. Each mechanism provides `...get()` function to acquire an entity and `...ctl()` to change permissions and

[4]Some Unixes support full duplex pipes through both file descriptors.

remove entity (and mechanism specific things).

Semaphores can be used to synchronize processes, their 'kernel-space relatives' were described in chapter 3.2. A set of semaphores is allocated by `semget()`. IPC Semaphore operations (`semop()` and `semtimedop()`) work on semaphore sets, the advantage is that process is able to (un)lock arbitrary number of semaphores atomically. This could prevent some situations leading to deadlock. Semaphore sets can by controlled by `semctl()` function.

Message Queues can be created (`msgget()`) in kernel to allow processes to exchange messages. Message queue is created by some process, it can overlive its creator and lives until it's killed by `msgctl()`. Messages are appended to the queue in order in which they're sent (`msgsnd()`. Each one has its type set upon sending (a positive number) and it's reception (`msgrcv()`) is based on this type.

The last of IPC mechanisms described here is shared memory. One process creates shared memory (using `shmget()`) and other processes refers to it by its key. Each process can map this shared memory into specific address inside its VM independently on the other processes or let `shmat()` choose appropriate address. Note that removing shared memory using `shmctl()` does not actually removes shared memory, it only marks it to be removed. The shared memory itself is removed once the last process detaches using `shmdt()`. If all processes detach themself without destroying shared memory, it keeps on existing.

While System V introduced IPC, the BSD branch of unices has chosen sockets as their main inter-process communication primitive. They appeared first in BSD 4.2. The API taken from BSD is supported on all unix and also on many non-unix platforms.

Socket is end-point for communication, it has some type and one or more associated processes. Sockets can send/receive data through streams, using datagrams, raw packets and sequenced packet. The most interesting protocol/address families are `PF_UNIX`, `PF_INET`, `PF_INET6`, `PF_PACKET` and `PF_NETLINK`. For example infrared and bluetooth devices also has their protocol families: `PF_IRDA` and `PF_BLUETOOTH`. Linux implementation is based on `sockfs` file system and so some file operations can be applied to opened sockets.

A stream socket (`SOCK_STREAM`) can be viewed as network-transparent duplex pipes, because it connects two distinct end-points, communication is reliable and sequenced and ensures unduplicated data flow without record boundaries.

A datagram socket (`SOCK_DGRAM`) does not promises to be reliable, sequenced and unduplicated, but it allows sending broadcast or multicast packets. Data boundary are preserved.

A raw socket (`SOCK_RAW`) provides user access to the underlying protocols which

support socket abstractions. They are usually datagram oriented and non-root user is not allowed to use these.

A sequenced packet (`SOCK_SEQPACET`) is like stream packet with the difference that message boundaries are preserved, but it is based on datagrams.[5].

`socket()` call creates socket descriptor for given address/protocol family, socket type and protocol. This socket is created without any *local name* and therefore it is not yet addressable. `bind()` call binds the socket e.g. to some *local path* (in case of `PF_UNIX`) or *local address and port* (in case of `PF_INET`).

Next step could differ for `SOCK_STREAM` and `SOCK_DGRAM`, stream oriented sockets are required to use `connect()` call to connect to remote endpoint, while datagrams could either follow their 'stream cousins' or use `sendto()` and `recvfrom()` without being fixed on one remote endpoint. This is essential for broadcasting and multi-casting.

The listening `SOCK_STREAM` side will use `listen()` to indicate maximum number of outstanding connections to be queued for processing. These connections are picked from the queue using `accept()` call.

Linux allows the use of ordinary `read()` and `write()` calls to operate on sockets after `connect()` or `accept`. Sockets also support their own calls `send()` and `recv()`, which can be used for extended operations.

Connections are closed using `close()`, but they can be shut down prior to this call using `shutdown()`.

One nice advantage of having socket descriptors acting similarly to file descriptors is that they can be processed by `select()` call and related functions and macros.

`SOCK_STREAM` sockets do support *out-of-band* data, which is logically independent transmission channel associated with each pair of stream socket endpoints. Data inside this channel are delivered independently on data inside normal transmission channel. Incoming out-of-band data are indicated by `SIGURG` signal and normal data are indicated by `SIGIO`, just as it is with file interface. Sockets can be further tuned by `getsockopt()` and `setsockopt()`.

2.8 Virtual File System

Virtual File System (VFS) is layer providing abstraction of one file system, while there may be many file systems of different types mounted at various places in directory tree. This allows to access files independently on whether they are stored on `ext2` or `umsdos` or even on some remote computer. VFS separates applications

[5]Not implemented by `PF_INET`

from concrete implementation of file system, all data are stored in files separated into directory tree[6], whose root is marked as `/`.

The largest group of file systems can be described as disk based file systems, its typical representants are `ext2/3`, `reiserfs`, `xfs`, `vfat`... These file systems use some kind of block device to store information.

Network file systems `smbfs`, `nfs`, `coda` allow transparent access to files on remote computers.

The last group are special file systems, like `proc`, `sysfs`. Many of them do not store any real information, for example `procfs` offers information about processes in the system (usually `/proc/<pid>`) or various system information. Other interesting are `bdev` (block device filesystem), `rootfs` (used as root during bootstrap), `binfmt_misc` (miscellaneous binary executable file formats), `pipefs` (pipes implementation) and `sockfs` (sockets implementation).

Linux kernel supports one really interesting file system since 2.6.14: FUSE, which is file system in user space. This file system is more interesting for user space developers than kernel developers.

How does the VFS work? The idea is based on object oriented programming techniques: data and operations are encapsulated in objects, but since linux is coded in plain C, developers used structures with pointers to functions to implement methods.

Superblock object stores overall information about a mounted filesystem, inode stores information about specific files (directory is special file that contains information about other files and directories. Each inode has its unique inode number. `struct dentry` object stores information about linking of a directory entry with the corresponging file objects used by user mode processes. Kernel uses dentry cache to speed up translation of a pathname to the inode. File object keeps information about file opened by process (this object is only in memory, previous objects can have their file system dependent representation on disk-based file systems). Linux also features inode cache for better performance.

Each file system provides its `struct file_operations`; this object offers methods such as `open()`, `read()`, linux-specific `sendfile()`, asynchronous `aio_...()` operations and many others. When user mode process opens a file, kernel finds the right `file_operations` depending on the file system and newly created `file` structure has pointer to this structure.[7]

[6]Hard links can't link to directories, but symbolic links can, therefore directory structure with symlinks can create cyclic graph.

[7]Users complaining that they cannot umount partition with opened files on it should always think about pointers pointing to non-existing `file_operations` structure...

User process has its set of `struct file` opened files and `struct fs_struct`, because each process has its own root directory and current working directory. `fs_struct` also contains pointer to mounted filesystem of root directory and pointer to mounted file system of current working directory.[8] Structure `fs_struct` is used to map file descriptors to `struct file` objects.

Modern operating systems use various buffers to improve file system performance, linux caches are called page cache and buffer cache. Page cache keeps pages from memory-mapped files, pages read from block device directly, swapped-out pages from user space processes, shared pages from IPC etc. The buffer cache saves processes from suffering from slow block devices.

File locking is technique preventing simultaneous access to file. `fcntl()`-based file locking (from POSIX) allows to lock whole files and portions of file, but it is just advisory locking (affected processes must cooperate). Linux inheritance from BSD brought `flock()` system call, which allows also mandatory locks (based on the file system mount flags).

It's common to say in unix circles that everything is a file and so are devices. They're treated as files and they traditionally had their major/minor numbers, while udev is new way of managing devices in `[/dev]`. There are two types of devices: block devices (they can be accessed randomly) and character devices (they can't be accessed randomly, they're usually processed as streams).

`epoll()` is attempt to make `select()` and `poll()` system calls in more effective way. It works by building data structure in kernel first, then each file descriptor is added to it. This structure is built only once as opposed to forementioned syscalls, which usually go through the three arrays and check each one and add it to its wait queue.

Desktop systems often need to modify user about changes to files in directory shown in opened window. Kernel developers decided to switch from `dnotify` mechanism to `inotify`. This new approach allows to watch both files and directories for changes and does not cause any problems while umounting (`inotify` also notifies about part of directory tree being unmounted). Dnotify required to open one file descriptor per each directory watched and it did not notified about changes to files. Inotify is not implemented using syscall, it's based on `/dev/inotify`.

While it is sometimes sufficient to wait for an I/O, in some cases the process would lose performance while sleeping. The situation gets worse with more I/O requests pending at the same time. Asynchronous I/O (AIO) allows process to do

[8] Many of these constructions are used to speed up going through many structures, Linux kernel uses sometimes duplicity of information to gain speed.

some calculations while kernel performs I/O operations in the following manner: process submits one more I/O requests and uses separate interface to check for completed I/O operations. AIO works only with `O_DIRECT` files and user space buffer must be aligned in memory. Although according to POSIX signal *can* be delivered, Linux does support neither RT signals for AIO nor listio notification, cancellation of IO, sockets nor pipes. There are ongoing project aiming to implement complete POSIX AIO.

2.9 New Driver Model and Sysfs

Past kernel versions included various disparate driver models. The new driver model unifies them and consolidates set of data and operations into globally accessible data structures. In this new model all devices are inserted into one global tree. The new model is bus-centric with orientation towards seamless Plug'n'Play, power management and hot plug. Most future buses will support these operations. Operations common to all devices include probe, init, resume, suspend and remove calls. Two same devices always use one driver.

This new tree-based approach evokes a directory tree; indeed `sysfs` was created to export this device tree to user space. `sysfs` is tied closely to `kobject` infrastructure. Basic `sysfs` interface layers are devices, bus drivers and device drivers.

The kobject infrastructure implements basic functionality for larger data structures and subsystems: reference counting, maintaining sets of objects, object set locking and userspace representation. Devices drivers are implemented around this infrastructure.‘

2.10 Networking

Networking as inter-process communication solution was described in one of previous sections. But strength of Linux lies also in its routing, firewalling and shaping possibilities.

Computer networks are interconnected, this allows users to communicate even if they aren't in the same network. The solution to problem how to deliver packages of data (called packets) to other network is routing. Router is a machine connected to two networks, which is able to route packets between them. The routing is done inside the kernel: packet arrives from one network interface adapter and kernel decides whether it belongs to this computer (e.g. in TCP/IP networks this could be done by comparing IP address, but there is also exception: network address

translation). If the packet does not belong to this computer, it could be forwarded (based on the settingsi, e.g. routing can be disabled) to the other network.

TCP/IP implementation relies on IP routing table, which is maintained from user mode, but routing is done in kernel. There are also user space deamons for more dynamic routing, but they're not interesting from our point of view.

Opening networks to whole world provides opportunities to share computing sources, but also enables malevolent forces to take over computers, steal data from companies... The idea of ability to look inside packets being routed to *our* network formed the idea of packet filter, which was initiallyr stateless and later became stateful. Stateful packet filter (from now on abbreviated to PF) is aware of state of (e.g. TCP) connections.

Packet filter decides what to do by looking inside the packet and to the known state of this connection. These decision rules are stored in chains; packet traverses the chain from beginning til end until it is matched by some rule. Common targets are `ACCEPT`, `REJECT` or `DROP`, some others will be covered later. If the packet reaches the end of the chain, default target (policy) for this chain is used.

PF implementation for Linux is called Netfilter. Netfilter contains four tables: `filter`, `nat`, `mangle` and `raw`. The simplest table is `filter` which has three default chains. `INPUT` chain checks the packets destined to local sockets, `OUTPUT` checks locally generated packets. `FORWARD` chain takes care of packets trying to be routed through this machine.

Table `nat` takes care of network address translation (therefore the abbreviation NAT, which will be used from now). Linux has the possibility to do both source and destination NAT, which makes Linux popular with ISPs nowadays. The last two tables are outside of scope of this work.

The power of netfilter allowed to use connection tracking to create modules, that allow more complex operation, such as FTP, IRC DCC or PPTP over NAT. This could not be possible (or at least not so simply) with stateless PF.

Traffic control (or sometimes call shaping) is a mechanism to alter the traffic characteristics. Slowing down chosen connections, bandwidth throttling, allowing short and fast bursts of packets, it all can be done using Linux. Traffic control inside Linux relies either on classless or classful queueing disciplines (from now on: qdisc). Classless qdiscs don't have any internal subdivisions, while classful ones can compose whole trees of classes, of which some ones can contain further qdics (classful or not). A packet is examined by classifier and sent to appropriate class and so on. This approach provides very strong and flexible tool for system administrators.

Linux also has the ability to tunnel one protocol inside other protocol (or even the same protocol: IPIP), to bind multiple physical links to aggregate bandwidth or

provide higher availability, to bridge (almost) various network interfaces (and also filtering bridge is available).

2.11 Linked Lists and Their Implementation

Linked lists are used throughout whole the kernel and it's necessary for kernel developers to understand how to use them correctly. The essential file containing all the list stuff is <`linux/list.h`>. Their API and implementation is described in chapter 4.

Chapter 3

Synchronization and Workflow Concurrency

Data structures so complex, that modifying them by multiple writers at once could break consistency, need to be protected. Solutions of this problems fall into category of *mutual exclusion*. Critical region is the region of code which (when executed) cannot be interrupted/preempted by other process or thread.

Linux kernel addresses problem of mutual exclusion using many techniques, depending on many factors: how long will run the code in critical region, whether IRQ handlers can access protected data structures or simply whether the mutual exclusion wanted is in user mode or kernel mode.

Technique used to protect bigger pieces of code (and those, which will run for longer time) is semaphore. Some semaphore implementation uses spinlocks or atomic operations to access internals of semaphore structure, while the semaphore itself could cause process to sleep and to `schedule()` another process to run. This is unwanted in IRQ handler, in which only spinlocks should be used (and atomic operations, as will be described later).

In 2.6.16 new mutexes were introduced, these are similiar to semaphores, but have better debugging support and therefore should be prefered by developers.

Atomic operations are simple operations (such as *subtract and test*) that can often be performed by one instruction with `LOCK` prefix, but C compiler could generate code consisting of more instructions and without `LOCK`, but that wouldn't be atomic. Implementation of atomic operations is always architecture dependent.

Futex is an abbreviation for Fast Userspace muTEXes. The basic thought is to make system call only when there is contention between processes/threads accessing the same futex. Its implementation is based on some kind of shared memory (`mmap()`, shared segments or simply because the threads share VM) and threads are

using atomic operations on integer. When they find contention, they use system call to arbitrate this case.

Special forms of locking are needed for preemptible kernel: both concurrency and reentrancy could collapse kernel. However preemptive locking is based on SMP locking mechanisms and we need only few additional functions/macros for this situations: per-CPU data needs protection (protection between processors is done with SMP locking) and FPU state was saved only when user space tasks were switched.

3.1 Spinlock

The simplest and basic technique used is spinlock, to use it include <linux/spinlock.h>. This mechanism prevents to enter critical section when the lock is locked by entering busy loop until lock is unlocked. Spinlocks are implemented by set of macros, some prevent concurrency with IRQ handlers while the other ones not. Spinlock user should always decide which one is most suitable, since incorrect usage could cause performance loss on SMP machines.

Spinlocks are suitable to protect small pieces of code which are intended to run for a very short time. Spinlocks have been extended to support read/write spinlocks to increase performace in case of many readers and one or very few writers.

Spinlock macros are in non-preemptible UP kernels evaluated to empty macros (or some of them to macros just disabling/enabling interrupts). UP kernels with preemption enabled use spinlocks to disable preemption. For most purposes, preemption can be tought of as SMP equivalent.

```
spinlock_t some_lock=SPIN_LOCK_UNLOCKED;

spin_lock(&lock);
/* critical section goes here */
spin_unlock(&lock);
```

This example shows us the simplest usage of spinlock. If the data protected by this section can be changed in an interrupt handling routine, simply use `spin_lock_irqsave(lock, flags)` and `spin_lock_irqrestore(lock, flags`. `spin_lock_irqsave()` saves the flags and disables IRQs and `spin_lock_irqrestore()` restores the previous state of interrupts, i.e. if they were disabled they will stay disabled and vice versa. The `read...` and `write...` versions of spinlock are implementation of read/write spinlock, which allows either multiple readers or one writer to enter the critical section at a time. This type of spinlock is very suitable when there is a lot of readers and writing is performed scarcely.

`spin_lock(lock)`,
`read_lock(lock)`,
`write_lock(lock)`
If the spinlock is unlocked, lock it and enter the critical section. The read/write spinlock locked for read does not stop `read_lock()` from entering the critical section; the readers counter is increased. Writer may lock r/w spinlock only if there is no reader present in the critical section. If the lock is locked, *spin* until it is unlocked (therefore the name spinlock).

`spin_lock_bh(lock)`,
`read_lock_bh(lock)`,
`write_lock_bh(lock)`
Same as the `spin_lock` and derivates, but prevents execution of softIRQs (bh is abbreviation for bottom halves, which is the older name for softIRQs).

`spin_lock_irq(lock)`,
`read_lock_irq(lock)`,
`write_lock_irq(lock)`
Lock the lock (for details look at `spin_lock`) and in addition disable IRQs.

`spin_lock_irqsave(lock, flags)`,
`read_lock_irqsave(lock, flags)`
`write_lock_irqsave(lock, flags)`
Lock the lock same as with `spin_lock()` and friends, but in addition store whether IRQs are enabled and disable them.

`spin_trylock(lock)`,
`read_trylock(lock)`,
`write_trylock(lock)`,
`spin_trylock_bh(lock)`
Try to lock the lock (for details see `spin_lock()` and friends), but if it is not possible to lock the lock exit with zero return value). Otherwise the return value is non-zero.

`spin_unlock(lock)`,
`read_unlock(lock)`,
`write_unlock(lock)`
Unlock the lock, in case of read lock decrease the readers counter. When this counter reaches zero, the writer may enter the critical section.

`spin_unlock_bh(lock)`,
`write_unlock_bh(lock)`,
`read_unlock_bh(lock)`
Unlock the lock locked by `spin_lock_bh()` or its appropriate cousin and allow softirqs to be processed.

`spin_unlock_irq(lock)`,
`read_unlock_irq(lock)`,
`write_unlock_irq(lock)`
Unlock the lock and enable IRQs. `...unlock_irqrestore()` is more suitable in most common cases. `read_unlock_irq()` enables IRQs after the last reader has left critical section.

`spin_unlock_irqrestore(lock, flags)`,
`read_unlock_irqrestore(lock, flags)`,
`write_unlock_irqrestore(lock, flags)`
Unlock the lock and enable IRQs only if they were enabled before locking the lock. `read_unlock_irq()` enables IRQs after the last reader has left critical section.

Please note that `spin_lock_irqsave()` and similar macros should have their unlocking counterparts in the same funcions, this is caused by limitations of SPARC architecture. The other good reason is readability. Spinlock may not block, switch context or enter idle loop.

3.2 Semaphores

This locking entity is deprecated since Linux 2.6.16.

Semaphore is defined in <`asm/semaphore.h`> and it's represented by structure `struct semaphore` containing counter `count` of threads trying to acquire the semaphore, number of `sleepers` in the wait queue `wait`. Semaphore structs can be initalized using `__SEMAPHORE_INITIALIZER(name, count)` macro, or together with declaration using `__DECLARE_SEMAPHORE_GENERIC(name, count)`.

`void sema_init(struct semaphore *sem, int val);`
Initialize a semaphore's counter `sem->count` to given value `val` and prepare wait queue.

`inline void down(struct semaphore *sem);`
Try to lock the critical section by decreasing `sem->count`, if there are already enough threads inside the critical region, mark the `current` thread

as TASK_UNINTERRUPTIBLE and sleep (schedule())until it is woken up by some thread exiting the critical section. Then mark current thread as TASK_RUNNING and enter the critical section.

`inline int down_interruptible(struct semaphore *sem);`
This function does mostly the same operation as down(), but sleeping process is marked as TASK_INTERRUPTIBLE. If there are any signals pending for the current thread, exit before entering sleep state.

`inline int down_trylock(struct semaphore *sem);`
Try to acquire semaphore, if it fails return non-zero value, otherwise zero.

`inline void up(struct semaphore *sem);`
Release the semaphore and if there are any tasks sleeping on this semaphore, wake them up.

Special case of semaphores are binary semaphores, also called mutexes. Unlocked mutex can be declared using DECLARE_MUTEX(name) macro and locked one using DECLARE_MUTEX_LOCKED(name). There are also two functions with self-explaining names for initalize mutexes: inline void init_MUTEX(struct semaphore *sem) and inline void init_MUTEX_LOCKED(struct semaphore *sem). The same operations work on mutexes as they did on semaphores (since mutex is simply a semaphore with initial count value of 1).

The schizophrenia of lock names was strengthened with arrival of 2.6.16: a new Mutex was introduced (while this old one still exists). This new mutex is described in its own section.

3.3 Read/Write Semaphores

Read/Write semaphores offer the possibility to have either one writer or any number of readers at a time. struct rw_semaphore contains member variable activity, which indicates current status: any positive number means number of readers, -1 means one writer and 0 means no activity. RW semaphore also features its wait queue protected by spinlock. RW semaphore can be initalized either using DECLARE_RWSEM(name), which declares variable struct rw_semaphore name; or by assigning __RWSEM_INITIALIZER(name to variable.

`void init_rwsem(struct rw_semaphore *sem)`
Initialize rw semaphore internals.

`void down_read(struct rw_semaphore *rwsem)`,
`void down_write(struct rw_semaphore *rwsem)`
Acquire semaphore for reading (`up_write()` for writing). If it's not possible at the moment, sleep until some thread exiting critical section wakes us up.

`int down_read_trylock(struct rw_semaphore *rwsem)`,
`int down_write_trylock(struct rw_semaphore *rwsem)`
Try to acquire semaphore for reading (`up_write()` for writing). If the semaphore was acquired, return 1, otherwise 0.

`void up_read(struct rw_semaphore *rwsem)`,
`void up_write(struct rw_semaphore *rwsem)`
Release previously acquired lock and wake up waiting threads.

`void downgrade_write(struct rw_semaphore *rwsem)`
Downgrade previously acquired write lock to read lock (and thus allow other readers to enter critical section).

3.4 Completions

Completion is simple synchronization mechanism close to semaphore. Imagine two cooperating threads: thread A needs to wait for thread B to finish some work. Idle loop would be inefficient, thread A decides to sleep until thread B wakes him up. Completion API is accessible through <`linux/completion.h`>

A completion is declared using `DECLARE_COMPLETION(work)` macro, which declares a variable of type `struct completion`:

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

Member variable `done` will be explained later, `wait` is a waitqueue keeping all waiters for this completion.

`void init_completion(struct completion *x);`
Reinitialize the `completion` structure: both `done` and `wait` member variables. This function is used for dynamic initialization. For reinitialization see `INIT_COMPLETION()`.

`INIT_COMPLETION(x);`
: This macro initializes completion: only `done` member variable. `wait` is untouched, see `init_completion()`. This macro should be used for fast reinitialization.

`void wait_for_completion(struct completion *x);`
: Wait for completion: the task will become `TASK_UNINTERRUPTIBLE` and will sleep until woken up by someone (using `complete()` or `complete_all()`).

`int wait_for_completion_interruptible(struct completion *x);`
: Wait for completion: the task will become `TASK_INTERRUPTIBLE` and will sleep until woken up by someone (using `complete()` or `complete_all()`). The sleep may be interrupted (and woken up) by a signal.

`unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout);`
: Wait for completion: the task will become `TASK_UNINTERRUPTIBLE` and will sleep until woken up by someone (using `complete()` or `complete_all()`) or `timeout` expires. Timeout is given in jiffies.

`unsigned long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout);`
: Wait for completion: the task will become `TASK_INTERRUPTIBLE` and will sleep until woken up by someone (using `complete()` or `complete_all()`) or `timeout` expires. Timeout is given in jiffies.

`void complete(struct completion *);`
: Wake up one sleeper.

`void complete_all(struct completion *);`
: Wake up all sleepers.

Why the reinitialization ? What happens if the `complete()`-like call is called before `wait_for_completion()`-like is called ? Nothing wrong, because this is the reason why `completion->done` exists: when a thread calls `wait_for_completion()`, `done` is first checked, if it's true then the second thread has already completed its part and there's no need to sleep. If it's false, the thread goes to sleep.

3.5 *New* Mutexes

This new locking entity was introduced in 2.6.16, but the name conflicts a bit with some macros concerning semaphores (mutexes were implemented as semaphores with limit of only 1 thread being able to entry critical section). <`linux/mutex.h`> should be included instead of **deprecated** semaphores (<`asm/semaphore.h`>).

Only one task may hold the lock and only the owner may unlock the mutex. Multiple unlocks and recursive locking are not permitted. Mutex must be initialized via the API. A task may not exist with mutex held and memory areas where held locks reside must not be freed. Held mutexes must not be reinitalized and finally mutexes cannot be used in irq contexts. These rules are enforced, when `DEBUG_MUTEXES` is enabled, which helps tracking down locking bugs. See `include/linux/mutex.h` for more details.

The non-debug version of mutex is simply a `struct mutex` with three fields: `atomic_t count` holding one of following values: 1 means unlocked, 0 means locked and negative: locked, possible waiters. Next field is `spinlock_t wait_lock` which protects `struct list_head wait_list` from being inconsitently modified.

The task waiting for a mutex to be unlocked is represented by `struct mutex_waiter` control structed, which contains only two self-explaining fields, `list_head` and `struct task_struct *task`.

`int mutex_is_locked(struct mutex *lock);`
: Return 1 if mutex is locked and 0 otherwise.

`DEFINE_MUTEX(name);`
: Macro to declare a `struct mutex` variable.

`mutex_init(mutex);`
: Initialize the mutex to unlocked state.

`void mutex_lock(struct mutex *lock);`
: Acquire mutex. Debug version of this function also checks if forementioned rules are obeyed.

`int mutex_lock_interruptible(struct mutex *lock);`
: Lock the mutex. If the lock has been acquired, return 0. If a signal arrives while waiting for the lock, then this function returns with `-EINTR`.

`int mutex_trylock(struct mutex *lock);`
: Try to acquire the lock, if it's locked then return value is 1 otherwise 0.

`void mutex_unlock(struct mutex *lock);`

Release previously acquired mutex.

Some inconsistency is introduced by `mutex_trylock()` and `mutex_lock_interruptible()`: if the lock is successfully acquired, one function returns 1 while the other one returns 0.

3.6 Futexes

Futex is abbreviation for **F**ast **u**ser space mu**tex**. Futex was designed to be as fast as possible. Down and up operations are performed in userspace using atomic assembler instructions. When there is no contention, there's no need to use syscall (and therefore this save expensive context-switches). If one there is contention, one of the processes calls `sys_futex()`. The API is provided by <`linux/futex.h`>.

`sys_futex(u32 __user *uaddr, int op, int val, struct timespec __user *utime, u32 __user *uaddr2, int val3);`. `uaddr` points to 32bit integer in the user-space, `op` is the operation to be done with the futex. The meaning of the rest of the arguments is depends on the operation `op` specified:

`FUTEX_WAIT`

The kernel is asked to suspend the thread, return value is 0 in this case. Before the thread is suspended value of `*addr` is compared to `val` and if they differ the syscall is exited with error code `EWOULDBLOCK`.

The `utime` argument indicates whether the thread will sleep for an unlimited time or whether it should be woken up with return code `ETIMEDOUT`.

If the thread falls asleep and a signal occurs, the syscall is exited with `EINTR`.

`FUTEX_WAKE`

This operations wakes up `val` sleeping threads. To wake up all sleepers use value of `INT_MAX`. From the user mode point of view the thread waking up the others does not know how many sleepers are there and which one will be woken up, so usual `val` values is either 1 or `INT_MAX`. The return value is number of threads woken up.

`FUTEX_FD`

This operation depends on `val` value:

If it is zero, the return value of this syscall is file descriptor, which can be used to `poll()`/`epoll()`/`select()`. After the poll the `revents` are set to `POLLIN|POLLRDNORM` if there are no waiters.

If the `val` is non-zero the kernel associates it as signal with returned file descriptor so that it is sent when the thread is woken while waiting on the futex.

`FUTEX_CMP_REQUEUE`

This operation allows to wake up a given number of waiters. Moreover the threads that did not woke up are removed from `addr` futex's wait queue and are added to the wait queue of futex `addr2`. Pointer to `timeout` is typecasted to int and used to limit the amount of threads to be requeued. Whole this operation is started only if `val3` equals to `*addr`, otherwise the syscall will end up with return code `EAGAIN`.

`FUTEX_REQUEUE`

This is predecessor to `FUTEX_CMP_REQUEUE`, but it's now considered buggy, broken and unstable. Use `FUTEX_CMP_REQUEUE` and prevent deadlocks.

`FUTEX_WAKE_OP`

This is the most complicated of all futex operations: Imagine having more than one futex at the same time and a conditional variable implementation, where we have to obtain lock before every operation. This would lead to heavy context switching and thus performance degradation. The solution is to have internal sleep array and waking up the thread only when the condition is fulfilled.

Operators supported are *set value*, *add*, and bitwise *or*, *and*, and *xor*. Comparison operators are *equivalence*, *non-equivalence*, *lesser than*, *lesser or equal*, *greater than*, *greater or equal*.

Whether the wakeup actually happens depends on the result of conditional expression.

3.7 Preemption

The preemption is the ability to switch the threads also in kernel mode. The implementation is mostly based on existing SMP-locking infrastructure, but there are some special cases that need to be taken into account. Including <`linux/preempt.h`> will provide developer macros for disabling/enabling preemption.

Preemption opens new locking issues inside the kernel. Existing locking primitives are preempt-compatible (for example spinlocks implementation use `preempt_disable()` and `preempt_enable()`). Problems appear, when thread is

doing something CPU-dependent, for example modifying per-CPU data structure. The thread could be preempted and woken up again on another CPU, therefore modifications to per-CPU structures are not consistent.

Another case is when CPU state needs to be protected and some other process preempts current one. FPU state is saved only when switching user space tasks, so kernel routines using FPU must use `kernel_fpu_begin()` and `kernel_fpu_end()`, which use `preempt_disable()` and `preempt_enable()`.

Beginning of critical section is marked with `preempt_disable()`, which decreases preempt counter. `preempt_enable_no_resched()` increases the counter, while `preempt_enable()` unrolls forementioned macro and checks whether there is opportunity to reschedule. This macros also mark end of critical section. The check for need for rescheduling is performed by `preempt_check_resched()` macro.

Macro `preempt_count()` is used to check preempt-counter, non-zero value means that preemption is forbidden at this moment, otherwise it is allowed.

Preemption must be enabled by the same thread that disabled it. The reason is that `preempt_count()` is defined as `current_thread_info()->preempt_count`, which is local to this thread.

It is possible to prevent a preemption using `local_irq_disable()` and `local_irq_save()`, but the danger is when an event occurs that would set `need_resched` and result int preemption check. There is no need to explicitly disable preemption when holding any locks, but any `spin_unlock()` decreasing the preemption count to zero might trigger a reschedule (e.g. a simple `printk()` might trigger a reschedule. The best practice is to disable preemption explicitly and only for small areas of atomic code, it is not recommended to call complex functions from inside these areas.

3.8 Seqlock and Seqcount

Seqlock is defined in <`linux/seqlock.h`>. Seqlocks are specialized primitive locks, readers and writers are distinguished. Readers never block and they are willing to retry if information changes. Writers do not wait for readers. This type of lock should not be used to protect pointers, because writer could invalidate pointer that a reader was using.

The implementation is a `struct seqlock_t` containing `spinlock_t lock` and `unsigned sequence`. Seqlock can be initialized using `seqlock_init()` macro, which initalizes the seqlock to be unlocked. The `sequence` is always initialized to be 0 and it is increased with each operation on seqlock.

Behavior of writing routines is identical to spinlock (with addition of `sequence` increasing each time):

`static inline void write_seqlock(seqlock_t *sl);`
Acquire the seqlock for writing.

`static inline void write_sequnlock(seqlock_t *sl);`
Release the seqlock previously acquired for writing.

`static inline void write_tryseqlock(seqlock_t *sl);`
Try to acquire the seqlock, returning non-zero value on success and zero otherwise.

The difference is in behavior for readers, common use is as follows:

```
do {
    seq=read_seqbegin(&my_seqlock);
    ...
} while (read_seqretry(&my_seqlock, seq));
```

Following routines are available to readers:

`static inline unsigned read_seqbegin(const seqlock_t *sl);`
Return `sequence` number.

`static inline int read_seqretry(const seqlock_t *sl, unsigned seq);`
Check whether the reader processed consitent/correct information. This check first checks whether there was no writer at the time of acquiring the `sequence` (odd `seq` means there was a writer) and if then check whether the `seq` and `sequence` equal. Zero return value means that the information processed was consistent, non-zero value otherwise.

The second 'locking' type introduced in this section is not truly a locking primitive: `seqcount_t` does not uses any lock by itself.

The behavior from reader's point of view is the same as it is with `seqlock_t`, only the routines have changed their names to `read_seqcount_begin()` and `read_seqcount_retry()`.

Writers are assumed to use their own locking (probably mutexes, because spinlock is implemented in `seqlock_t`). There are two helper routines used to manipulate `sequence` number (`sequence` should not be changed directly, this routines use `smp_wmb()` memory barrier).

`static inline void write_seqcount_begin(seqcount_t *s);`

This routine should be called in the beginning of critical section.

`static inline void write_seqcount_end(seqcount_t *s);`

This routine should be called at the end of critical section.

The file `[<seqlock.h>]` also include `..._irq()`, `..._irqsave()`, `..._irqrestore()` routines that save/disable and restore IRQs, `..._bh()` routines that disable and enable tasklets.

3.9 Atomic Operations

Atomic operations are simple operations that are guaranteed to be executed without interruption or being interleaved by other operations. Since C compiler can't guarantee us atomicity, their implementation is highly architecture-dependent: `[asm/atomic.h]`.

These operations are based around structure `atomic_t`. The initialization is implemented by macro `ATOMIC_INIT`:

```
atomic_t my_atomic_counter = ATOMIC_INIT(0);
```

Operations are also implemented as macros (at least on x86) for performance reasons:

`atomic_read(v)`

Atomically read the value of `v`.

`atomic_set(i, v)`

Atomically set the value of `v` to `i`.

`atomic_add(i, v)`

Atomically add the `i` to the value of `v`.

`atomic_sub(i, v)`

Atomically subtract the `i` from the value of `v`.

`atomic_sub_and_test(i, v)`

Atomically subtract `i` from `v` and return true if and only if the result is zero.

`atomic_inc(v)`

Atomically increase value of `v` by 1.

`atomic_dec(v)`
: Atomically decrease value of `v` by 1.

`atomic_dec_and_test(v)`
: Atomically decrease value of `v` by 1 and return true if and only if the result is zero.

`atomic_inc_and_test(v)`
: Atomically increase value of `v` by 1 and return true if and only if the result is zero.

`atomic_add_negative(i, v)`
: Atomically add `i` to `v` and return true if and only if the result is negative.

`atomic_add_return(i, v)`
: Atomically add the `i` to the value of `v` and return the result.

`atomic_sub_return(i, v)`
: Atomically subtract the `i` from the value of `v` and return the result.

`atomic_inc_return(v)`
: Atomically increase value of `v` by 1 and return the result.

`atomic_dec_return(v)`
: Atomically decrease value of `v` by 1 and return the result.

3.10 Disabling Local IRQs

Previous kernel version contained `cli()` and `sti()` macros to disable and enable IRQs. These macros together with `save_flags(flags)`, `save_cli_flags(flags)` and `restore_flags(flags)` are now deprecated.

From now on a better approach is to use spinlocks with `spin_lock_irqsave()`, `spin_unlock_irqrestore()`, `spin_lock_irq()` and `spin_unlock_irq()`. They provide better readability, spinlock exactly shows the critical section. Spinlocks are faster than global IRQ disabling.

Drivers that want to disable local IRQs (i.e. only on current CPU) can use following macros from <`asm/system.h`>:

`local_irq_disable();`
: Disable IRQs on current CPU.

`local_irq_enable();`
: Enable IRQs on current CPU.

`local_save_flags(flags);`
: Save the current IRQ state on local CPU into `flags`. On most architecture the state can be on and off, some ones are using even more bits than one.

`local_irq_save(flags);`
: Save the current IRQ state on local CPU into `flags` and disable IRQs.

`local_irq_restore(flags);`
: Restore previously saved IRQ flags.

3.11 Read-copy Update

Read-copy update is a synchronization mechanism optimized for mostly-read access. Reader has to acquire and later to release the lock, but acquiring a lock never blocks. Writer's behavior is more complicated: update is split into two phases: *removal* and *reclamation*. The removal phase removes reference to old data (possibly by replacing them with new data) and can run concurrently with readers. RCU works under presumption that updates are scarce and reader sometimes retrieves *out-dated* information. The old data are freed during reclamation phase. Because this could disrupt readers, the reclamation phase could not start until readers no longer hold reference to old data.

Splitting the code into two phases allows the `removal` phase to start immediately. The reclamation phase is deffered. Note that only readers, who were already active before removal phase should be considered, subsequent readers are working with new data.

Reader *may not* block, switch task context or enter idle loop. This indicates that preemption is disabled inside the reader's critical section.

Following code example declares variable `foo_var`, which will be protected by RCU:

```
struct foo {
    ...
};
DEFINE_SPINLOCK(foo_spinlock);

struct foo *foo_var;
```

The spinlock is used to prevent race condition of multiple writers. Reader's code:

```
rcu_read_lock();
some_var=rcu_dereference(foo_var);
rcu_read_unlock();
```

And writer's code:

```
struct foo *new, *old;

// 'new' points to new data
spin_lock(&foo_spinlock);

old=foo_var;
rcu_assign_pointer(foo_var, new);
synchronize_rcu();

spin_unlock(&foo_spinlock);
kfree(old);
```

As can be seen, the reader's code is very simple. On UP machines without preemption the macros `rcu_read_lock()` and `rcu_read_unlock()` are evaluated to empty strings, thus as fast as possible.

Include <`linux/rcupdate.h`> to gain access to following API:

`rcu_read_lock();`
: This macro marks the beginning of RCU protected section and disables preemption on preemptible kernels.

`rcu_read_unlock();`
: This macro marks the beginning of RCU protected section and enables preemption on preemptible kernels.

`rcu_read_lock_bh();`
: This macro marks the beginning of RCU protected section, disables softIRQs and also preemption on preemptible kernels.

`rcu_read_unlock_();`
: This macro marks the beginning of RCU protected section, enables softIRQs and also enables preemption on preemptible kernels.

Note: RCU-read critical section may be nested.

`rcu_dereference(pointer);`
: Fetch a pointer protected by RCU. This macro must be used in RCU-read critical section. This pointer may be dereferenced after `rcu_read_unlock()` or equivalent. Memory barrier is used on some architectures.

`rcu_assign_pointer(pointer, value);`
: Assign new `value` to `pointer` in the removal phase. Memory barrier is used on some architectures.

codevoid synchronize_rcu(); and `void synchronize_sched();` Wait until all affected readers (i.e. those that acquired the old pointer) have left the critical section. The other way is to use callback function to free old pointer when time comes: `call_rcu()` or equivalent.

`void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head))`
: ; Queue an RCU reclamation callback. The callback will be called when all affected reader (i.e. those working with the old pointer) have left the critical section.

`void call_rcu_bh(struct rcu_head *head, void (*func)(struct rcu_head *head))`
: ; Queue an RCU reclamation callback. The callback will be called when all affected reader (i.e. those working with the old pointer) have left the critical section. This version assumes that RCU-read critical section ends on completion of a softIRQ handler: therefore RCU-read critical sections may not be interrupted by softIRQs.

Note: this mechanism is covered by some patents in U.S.A.

3.12 Per-CPU Variables

Per-CPU variables (<`linux/percpu.h`>) are extensively used in 2.6 kernel (scheduler being nice example). The idea is to hold array of pointers instead variable of the same type, but this implementation is hidden behind a set of macros and functions. The good is that there is no synchronisation needed, because each CPU has its own value. The bad is that that some overhead is needed around critical section for preemptible kernels. Moreover the per-CPU variable's single values for

each CPU are scattered in memory so that cache on SMP CPUs is not invalidated by other CPUs.

`DEFINE_PER_CPU(type, name);`
: This macro defines the a per-CPU variable `name` of given `type`.

`get_cpu_var(var);`
: Mark `var` per-CPU variable accessible to this CPU. This in fact disables preemption on preemptible kernels. This is the beginning of critical section. Evaluates to l-value of the per-CPU variable, therefore `get_per_cpu(socket_in_use)++;` works.

`put_cpu_var(var);`
: Mark the end of critical section for per-CPU variable `var`, this also enables preemption.

Per-CPU variables can be allocated in the runtime too. The essential data structure is as follows:

```
struct percpu_data {
        void *ptrs[NR_CPUS];
        void *blkp;
};
```

Pointer array `ptrs` holds pointer to each instance of allocated type for each CPU possibly available in the system. They're not allocated as one array, if they were then writing value to one item could invalidate cache on some other CPUs (especially data types with small memory footprint).

`void *alloc_percpu(type);`
: Allocate per-CPU variable of given `type` and zero allocated memory. One copy is allocated for each CPU.

`void *free_percpu(const void *objp);`
: Free a per-CPU variable previously allocated by `alloc_percpu()`.

`per_cpu_ptr(ptr, cpu)`
: Access per-CPU variable of other `cpu`. Use with care and choose additional locking.

If the variable is to be exported to modules, use one of the following macros:

```
EXPORT_PER_CPU_SYMBOL(variable);
EXPORT_PER_CPU_SYMBOL_GPL(variable);
```

3.13 Memory Barriers

Designers of modern CPUs try to increase performance by executing instructions out of order they are in memory. This includes memory loads and stores. The reordering is not noticable by single thread, but CPUs in SMP machines accessing the same data at once could find data incosistent (for example one CPU is writing data out of order and the other is trying to read it in correct order).

The second part of the problem is compile-time optimization: gcc does the reordering by itself

Memory barriers are in Linux implemented as architecture-dependent macros (<`asm/system.h`>). The most common one are

`barrier();`
: Prevent compile-time reordering by inserting optimalization barrier (empty code, thus no performance loss).

`mb();`
: Prevent read, write and optimalization reordering (SMP and UP).

`rmb();`
: Prevent read and optimalization reordering (SMP and UP).

`wmb();`
: Prevent write and optimalization reordering (SMP and UP).

`smp_mb();`
: Prevent read, write and optimalization reordering (SMP only).

`smp_rmb();`
: Prevent read and optimalization reordering (SMP only).

`smp_wmb();`
: Prevent write and optimalization reordering (SMP only).

The reason to implement barriers in so much macros is optimalization, for example Intel CPUs does not do write reordering, so `wmb();` accts only as optimalization barrier.

Chapter 4

Linked lists

Double-linked lists are provided to kernel developer through <linux/list.h>.

Double-linked circular list is implemented around struct list_head structure. The list is anchored by one (initial) list_head variable. How is it possible to add other data structures into this list? A structure has to have member of type list_head, which will be part of the list. The next and prev members of struct list_head point to next and previous list heads (i.e. not to the structure containing struct list_head, but to its member of type list_head). This complicated approach provides some flexibility (a structure may contain more members of type list_head and also be part of more than one linked list at one time). Access to the original data structure (which the developer surely wanted to keep in the list) is provided by list_entry() macro.

The easiest way to define new list is the LIST_HEAD() macro, which unrolls itself into struct list_head name=LIST_HEAD_INIT(name). As we can see in file include/linux/list.h, list is a structure containing pointers to next and previous items in list. The list is double-linked circular, macro LIST_HEAD_INIT(name) is defined as { &name, &name }, thus the list is initialized as circle containing one item pointing to itself in both directions. List can be also initialized after declaration using INIT_LIST_HEAD(ptr), which provides the same functionality as described above.

4.1 List API

Following macros/inline functions are defined to ease developer with list manipulation:

```
void list_add(struct list_head *item, struct list_head *head);
```

Add the item to the beginning of the list, i.e. right behind the anchoring

list_head.

`void list_add_tail(struct list_head *item, struct list_head *head);`
Add the `item` to the end of the list, i.e. insert it between `head->prev` and `head`.

`void list_del(struct list_head *item);`
Delete `item` from list it's contained in. `prev` and `next` are set to `LIST_POISON1` and `LIST_POISON2`, which marks that the item has been deleted, but also prevents `list_empty(entry)` from returning true.

`void list_del_init(struct list_head *entry);`
Removes the `entry` from list (the rest of the list is left untouched) and `entry` is reinitialized as separate list.

`void list_move(struct list_head *source_item, struct list_head *target_list);`
Move the `source_item` from its list to the beginning of list `target_list`. It is possible to move the `source_item` from either other list to `target_list` or just move the `source_item` from inside the list to the beginning.

`void list_move_tail(struct list_head *source_item, struct list_head *target_list);` Move the `source_item` from its list to the end of list `target_list`. See `list_move()`.

`int list_empty(const struct list_head *head);`
Checks whether a list is empty. `head` is the anchoring `list_head`.

`int list_empty_careful(const struct list_head *head);`
Test whether the list is empty and check that the list isn't modified right now. This function is safe only if the only activity that can happen is `list_del_init()`. If some other CPU is adding items then it's not safe and some synchronization will be needed.

`void list_splice(struct list_head *source_list, struct list_head *target_list);`
Move `source_list` to the beginning of `target_list`, so that the last item of `source_list` will be just before `target_list`'s first item. The `source_list`'s members `next` and `prev` are not changed, so it's unsafe to use `source_list` after `list_splice()`. See `list_splice_init()`.

`list_splice_init(struct list_head *source_list, struct list_head *target_list);`

Move items in `source_list` to the beginning of `target_list`. See `list_splice` for details. In addition the `source_list` is reinitialized to empty list again.

`list_entry(pointer, type, member)` This macro returns pointer to structure of `type`, which contains `list_head` as its `member` variable. The `pointer` argument points this `member` inside the structure we want to access. See general description of linked lists in the beginning of this section and example in `list_for_each()`.

`list_for_each(iterator, list)`

Iterate over a `list` using given `iterator`. Common use of this macro is as follows:

```
struct list_head *iter;

list_for_each(iter, init_task) {
    struct task_struct *task=list_entry(iter,
        struct task_struct, tasks);

    /* do something with 'task' here */
}
```

This code iterates over a task list (`init_task` is the head of list of all process descriptors in Linux) and 'does' something for each task in the list. `struct task_struct` contains member variable `struct list_head *tasks`, which keeps it in list of tasks.

`list_for_each()` also does prefetching to speed up list processing.

`__list_for_each(iterator, list);`

Iterate over a `list` using `iterator`. see `list_for_each()` for details, but this macro does not do any prefetching, which makes it suitable only for very short lists.

`list_for_each_prev(iterator, list);`

Iterates backwards, see `list_for_each()` for details.

`list_for_each_safe(iterator, helper, list);`
Iterate over a `list` like `list_for_each()`, but this implementation adds `helper` variable and is safe against removal of list entry.

`list_for_each_entry(type *iterator, list, member);`
Iterate over a `list`. `struct list_head` is hidden inside the `type` as `member`. This macro can be viewed as union of `list_for_each()` and `list_entry`.

Example from `list_for_each()` can be rewritten:

```
struct task_struct *task;

list_for_each_entry(iter, init_task, tasks) {

    /* do something with 'task' here */
}
```

`list_for_each_entry_reverse(type *iterator, list, member);`
The backward version of `list_for_each_entry()`.

`list_for_each_entry_safe(type *iterator, type *helper, list, member);` Iterate over a `list` of give `type`. This version is safe against removal of list entry.

`list_prepare_entry(type *position, list, member);` Prepare a start point for iteration over `list`. `position` should be initialized to null pointer. This macro checks the value of `position` and it it isn't false then the `position` is not initialized to correct pointer for iteration.

`list_for_each_entry_continue(type *position, head, member);` Iterate over a list of given `type` starting (or continuing) from existing `position`.

4.2 RCU Protected Lists

Following functions are defined to help developer accessing RCU protected macros:

`void list_add_rcu(struct list_head *item, struct list_head *head);`
Add the `item` to the beginning of the RCU protected list, i.e. right behind the anchoring `list_head`. Writer synchronization (e.g. using spinlocks or mutexes) is left on developer.

`void list_add_tail_rcu(struct list_head *item, struct list_head *head);`

Add the `item` to the end of the RCU protected list, i.e. insert it between `head->prev` and `head`. Writer synchronization (e.g. using spinlocks or mutexes) is left on developer.

`void list_del_rcu(struct list_head *item);`

Delete `item` from an RCU protected list it's contained in. `prev` and `next` are set to `LIST_POISON1` and `LIST_POISON2`, which marks that the item has been deleted, but also prevents `list_empty(entry)` from returning true. Deleted item may not be freed immediately, instead use `synchronize_kernel()` or `call_rcu()`. Writer synchronization (e.g. using spinlocks or mutexes) is left on developer.

`void list_replace_rcu(struct list_head *old, struct list_head *new);`

Replace `old` entry by a `new` one. The entry may not be deleted, use `synchronize_kernel()` or `call_rcu()`. Writer synchronization (e.g. using spinlocks or mutexes) is left on developer.

`list_for_each_rcu(iterator, list);`

Iterate over an RCU protected `list` using `iterator`. Whole iteration must be protected using `rcu_read_lock()`, `rcu_read_unlock()` or equivalents.

`list_for_each_safe_rcu(iterator, helper, list);`

Iterate over an RCU protected `list` like `list_for_each()`, but this implementation adds `helper` variable and is safe against removal of list entry. Whole iteration must be protected using `rcu_read_lock()`, `rcu_read_unlock()` or equivalents.

`list_for_each_entry_rcu(type *iterator, list, member);`

Iterate over an RCU protected `list`. `struct list_head` is hidden inside the `type` as `member`. This macro can be viewed as union of `list_for_each()` and `list_entry`. Whole iteration must be protected using `rcu_read_lock()`, `rcu_read_unlock()` or equivalents.

`list_for_each_continue_rcu(position, list);`

Iterate over an RCU protected `list` starting (or continuing) from `position`. Whole iteration must be protected using `rcu_read_lock()`, `rcu_read_unlock()` or equivalents.

Chapter 5

Memory Allocation

5.1 Slab Memory Allocation

Linux kernel offers to kernel routines two functions similar to user space `malloc()` and `free()`. These functions are called `kmalloc()` and `kfree()` and they're based on slabs, therefore suitable only for allocating small buffers.

Mechanism used to allocate sub-page memory areas is calles slab allocator (SA). Slab allocator takes care of caching freed memory so that future allocations can be faster. The memory is allocated from physical contignuous memory and cannot be swapped out. Slabs are usually smaller than one page.

This API is defined in <`linux/slab.h`>:

```
void *kmalloc(size_t size, int flags);
```

Allocate buffer of `size`. The meaning of flag is as follows:

`GFP_KERNEL`
: Allocate the physical contignuous memory buffer in kernel space. Function may sleep and swap to free memory. This flag is allowed only in user context.

`GFP_ATOMIC`
: This flag forbids the call to sleep, but may fail in case where `GFP_KERNEL` would swap to free memory. Function with this flag may be called from interrupt context.

`GFP_DMA`
: Allocate memory in lower 16MB of memory. This is suitable for device drivers using ISA bus.

Return value is the pointer to allocated buffer or NULL if memory allocation failed.

`void *kzalloc(size_t size, int flags);`
: Allocate buffer of `size` using `kmalloc()`. The arguments are the same, but this call also zeroes allocated memory.

`void *kmalloc_node(size_t size, gfp_t flags, int node);`
: Allocate a buffer on given node (NUMA-specific call).

`void kfree(const void *buffer);`
: Free memory buffer previously allocated by `kmalloc()`.

`void ksize(const void *buffer);`
: Return the size of `buffer` previously allocated by `kmalloc`.

The power of slab allocator lies in its cache: the slabs of the same size are not freed from memory immediatelly, but they are held in cache. Slab cache is therefore suitable only for allocations of small pieces of memory.

`kmem_cache_t *kmem_cache_create(const char *name, size_t size, size_t align, unsigned long flags, void (*constructor)(void *, kmem_cache_t *, unsigned long), void (*destructor)(void *, kmem_cache_t *, unsigned long));`
: Create a cache that will allocate, free and hold objects of given `size`. `align` specifies memory alignment of allocated objects. The cache is identified by its `name` (used in `/proc/slabinfo`). Possible flags are:

  `SLAB_POISON`
  : Poison the slab with pattern `0xa5` to catch references to uninitialized memory.

  `SLAB_RED_ZONE`
  : Insert red zones around allocated memory to check for buffer underruns and overruns.

  `SLAB_NO_REAP`
  : Don't shrink memory when the system is looking for more available memory.

  `SLAB_HWCACHE_ALIGN`
  : Align the object in this cache to match hardware cacheline.

  The `constructor(void *object, kmem_cache_t *cachep, unsigned long flags)` and `destructor(void *object, kmem_cache_t *cachep, unsigned`

`long flags)` are optional. The constructor must be provided if the destructor is also given. Both callbacks may be implemented by the same function as the `constructor()` is called with flag `SLAB_CTOR_CONTRUCTOR`. The `SLAB_CTOR_ATOMIC` flag indicates that the callback may not sleep, `SLAB_CTOR_VERIFY` indicates that this call is a *verification* call. Constructor is called only for newly allocated objects, objects held in the cache and return by allocation call do not gain constructor's attention. Constructors and destructors are very rarely used.

This function returns pointer to created cache or `NULL` if creation failed.

`int kmem_cache_destroy(kmem_cache_t *cache);`
Destroy a `cache`. The cache must be empty before calling this function. The caller must take care that no allocation attempt will be made during `kmem_cache_destroy()`.

`int kmem_cache_shrink(kmem_cache_t *cache);`
Shrink cache, i.e. release as many slabs as possible. Zero return value indicates that all slabs were released.

`void *kmem_cache_alloc(kmem_cache_t *cache, gfp_t flags);`
Allocate an object from `cache`. For description of flags see `kmalloc()`.

`void *kmem_cache_alloc_node(struct kmem_cache_t *cachep, gfp_t flags, int nodeid);`
Allocate an object from `cache` on given node (NUMA-specific call).

`void kmem_cache_free(kmem_cache_t *cache, void *object);`
Release an `object`. This does not free memory, it only marks object as freed, but the memory is kept in `cache`.

`unsigned int kmem_cache_size(kmem_cache_t *cache);`
Return the size of the objects allocated by this `cache`.

`const char *kmem_cache_name(kmem_cache_t *cache);`
Return pointer to the name of the `cache`.

`kmem_cache_t *kmem_find_general_cachep(size_t size, gfp_t gfpflags);`
Find suitable cache from array of general caches. There are in fact two arrays, one for DMA memory and the other for normal allocations.

Many subsystems use of this cache API and create their own caches, the most common ones are:

`kmem_cache_t *vm_area_cachep;`
Cache for `vm_area_struct` structures.

`kmem_cache_t *files_cachep;`
Cache for `files_struct` (table of open files).

`kmem_cache_t *filp_cachep;`
Cache for `struct file` objects.

`kmem_cache_t *fs_cachep;`
Cache for `fs_struct` structures.

`kmem_cache_t *signal_cachep;`
Cache for structures (`signal_struct`).

`kmem_cache_t *sighand_cachep;`
Cache for signal handler structures (`sighand_struct`).

`kmem_cache_t *dentry_cachep;`
Cache for directory entries (`struct dentry`).

5.2 Virtual Memory Allocation

Virtual memory allocation functions (<`linux/vmalloc.h`>) offer high level API for allocating swappable memory. This API works with virtually contignuous memory as opposed to slabs. There is usually no need to use physically contignuous memory except for hardware devices, which don't understand paging and virtual memory. On the other side memory allocated using *vmalloc*-like functions is accessed through Translation Look-aside Buffer (TLB) instead of direct access, which can result in slight overhead when accessing this memory.

`void *vmalloc(unsigned long size);`
Allocate a virtually contignuous memory of given `size`. The allocated space is rounded up to whole pages.

`void *vmalloc_exec(unsigned long size);`
Allocate a virtually contignuous executable memory of given `size`. The allocated space is rounded up to whole pages.

`void *vmalloc_32(unsigned long size);`
Allocate a virtually contignuous (32bit addressable) memory of given `size`. The allocated space is rounded up to whole pages.

`void *__vmalloc(unsigned long size, int gfp_mask, pgprot_t prot);`

Allocate virtually contignuous memory buffer. The allocation takes enough pages to cover whole buffer `size`. `gfp_mask` contains mask of flags for page level allocator (see <`linux/gfp.h`>) and `prot` contains protection mask for allocated pages (see <`asm/pgtable/h`>).

`void vfree(void *addr);`

Release memory previously allocated by one of *vmalloc* functions. This function may not be called in interrupt context.

`void *vmap(struct page **pages, unsigned int count, unsigned long flags, pgprot_t prot);`

Map an array of `pages` into virtually contignuous space. `count` is the number of pages to be allocated, `prot` contains protection mask. Possible flags:

`VM_MAP`
: Pages mapped by `vmap()`.

`VM_ALLOC`
: Pages allocated by one of *vmalloc* functions.

`VM_IOREMAP`
: Mapped from hardware device.

`void vunmap(void *addr);`

Release mapping obtained by `vmap`. May not be called in interrupt context.

5.3 Page Allocation

Page allocation is the most elementar of all allocation methods. It's hardware dependent, but it's sometimes the most suitable when writing a hardware device driver.

Following interface is defined in <`linux/gfp.h`> and <`linux/mm.h`>:

`struct page *alloc_pages(unsigned int gfp_mask, unsigned int order);`

Allocate pages. `gfp_mask` is combination of following:

`GFP_USER`
: User allocation.

`GFP_KERNEL`
: Kernel allocation.

`GFP_HIGHMEM`
: Highmem allocation.

`GFP_FS`
: Don't call back into a file system. Without this option the allocator may swap out some pages to free some more physical memory.

`GFP_ATOMIC`
: Don't sleep, this also prohibits swapping out pages.

`order` is the power of two of allocation size in pages. Return the pointer to the first page's structure or NULL if allocation fails.

`void *page_address(struct page *page);`
: Return the logical address of the `page`.

`unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)`
: ; Allocate physical pages (see `alloc_pages()`) and return the logical address of the first of them.

`struct page *alloc_page(unsigned int gfp_mask);`
: Allocate one page and return pointer to it's page structure.

`unsigned long __get_free_page(unsigned int gfp_mask);`
: Allocate one page and return it's logical address.

`unsigned long get_zereod_page(unsigned int gfp_mask);`
: Allocate one zeroed page and return it's logical address.

`void __free_pages(struct page *pages,unsigned int order);`
: Free previously allocated pages.

`void free_pages(unsigned long addr, unsigned int order);`
: Free previously allocated pages.

`void free_page(unsigned long addr);`
: Free one page.

Chapter 6

Handling Interrupts

Interrupts signalize operating system about events that occured in hardware devices. CPU knows how to handle each interrupt as it has pointers to interrupt handlers stored in a table. This handlers are usually invoked with interrupts disabled[1]

Interrupt handlers need to be fast, because we don't want to lose other pending interrupts. The code is split in linux into two layers, the interrupt handler usually just queues needed routines as softIRQs or taskqueues (both will be described later) and quits. Taskqueues will be executed later with interrupts enabled[2] and this way no interrupt should be lost.

6.1 Hardware Interrupts

Hardware devices generate interrupt requests from time to time, the meaning of an IRQ is to notice software (the kernel in our case) about some event: tick of timer, arrival of packet from network... Understanding how to properly handle IRQs is essential for device driver developers.

The kernel offers interrupt handlers to handle IRQs. The handler won't be entered by the same interrupt more than once, if the same IRQ is generated, it is queued or dropped. The handler runs with interrupts disabled (this is done automatically by CPU on or IRQ controller on some architectures) so it has to run very fast and *may not* block. It's usual to set a *software interrupt* and exit the handler.

[1]Standard PCs have been using Programmable Interrupt Controller for a long time, this controller takes care of (un)masking individual interrupts. MSDOS programmers probably remember *magic sequence*, which wrote `0x20` to port `0x20`, and thus allowing PIC to report interrupts to CPU again.

[2]But taskqueues can disable interrupt while doing something critical.

General rule: don't call any blocking function waiting for IRQ handle that could block while holding a resource that may be used by IRQ handler. Avoid deadlocks.

Interrupt lines may be shared by devices if all interrupt handlers are marked with flag `SA_SHIRQ`.

Include <`linux/hardirq.h`> to access following function and macros:

`in_irq();`
: Returns true if called in a hardware IRQ.

`void synchronize_irq(unsigned int irq);`
: Wait for pending IRQ handler exit for `irq`. This function may be called with care from IRQ context.

`void disable_irq_nosync(unsigned int irq);`
: Disable `irq` without waiting. Disables and enables are nested. This function does not guarantee that any running handlers have exited before returning. This function may be called from IRQ context.

`void disable_irq(unsigned int irq);`
: Disable `irq` and wait for any handlers to complete. Disables and enables are nested. May be called from IRQ context.

`void enable_irq(unsigned int irq);`
: Enables `irq`. Enables and disables are nested. May be called from IRQ context.

`void free_irq(unsigned int irq, void *device);`
: Remove interrupt handler from `irq` for given `device`. If the interrupt line is not used by any other driver then the line is blocked. Caller must ensure that interrupt is disabled on the device prior to calling this function. The function does not wait for completion of any running handlers. It may not be called from from IRQ context.

`int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char *devname, void *device);`
: Allocate an interrupt line `irq` with a `handler`. Flags can be:

  - `SA_SHIRQ` - Interrupt is shared. If there is already any handler bound to this IRQ without this flag set, this `request_irq()` will return with error code `-EBUSY`. Otherwise the new interrupt handler is added to the end of the interrupt handler chain.

- `SA_INTERRUPT` - Local interrupts are disabled while processing.
- `SA_SAMPLE_RANDOM` - Interrupt can be used for random entropy.

Argument `devname` is the ASCII name for the claiming device and `device` is a cookie passed back to handler. `device` must be globally unique (e.g. address of the device data structure). The interrupt line is enabled after allocation. If the interrupt is shared, `device` must be non-NULL.

6.2 Softirqs and Tasklets

Servicing harware interrupt may not take too much time, so the work is split into two parts: the minimal servicing is done in the hardware IRQ handler and the rest is deffered. Previous kernel version used *bottom halves* (BH) to process the deferred which didn't take advantage in using multiple CPUs and developers decided to switch to softIRQs. Some macros still contain `bh` substring as a remainder from the past: `local_bh_enable()` and `local_bh_disable()`, though they actually disable end enable softirqs.

Softirqs are statically created and there are 6 softIRQs defined in kernel 2.6.16:

`HI_SOFTIRQ`
: high priority tasklets

`TIMER_SOFTIRQ`
: timer interrupts

`NET_TX_SOFTIRQ`
: network packet transmission

`NET_RX_SOFTIRQ`
: network packet receiving

`SCSI_SOFTIRQ`
: SCSI-related processing

`TASKLET_SOFTIRQ`
: tasklets

Softirqs can be run by multiple CPUs at once, moreover one softirq can be run by multiple CPUs at once. They're also reentrant. Synchronization is essential.

Include <`linux/interrupt.h`> to access following macros and functions:

`in_softirq();`
: Return true if in softIRQ. Warning: this macro returns true even if BH lock is held.

`in_interrupt();`
: Return true if in hardware IRQ of softIRQ. Warning: this macro returns true even if BH lock is held.

Tasklets (as opposed to softirqs) can be creating in runtime, so they'll be of bigger interest to us. They are executed from softirqs and kernel guarantees not to execute one tasklet more than once on multiple CPUs. Tasklets of the same type are serialized, but tasklets of different type may run concurrently. Tasklets are also preferred way of doing things, they're sufficient for most operations done by the drivers.

The tasklet is built aroung following struct:

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

The `next` field is pointer to next tasklet in the list. `func(data)` will be called once the tasklet is to be executed. The `state` can hold zero (meaning: tasklet not in list) or combination of the following values:

`TASKLET_STATE_SCHED`
: Tasklet is scheduled for execution. A tasklet without this flag will not start, even if it's in the tasklet list.

`TASKLET_STATE_RUN`
: Tasklet is running (SMP only).

Tasklet can be easily declared using `DECLARE_TASKLET(name, func, data)`, which declares `struct tasklet_struct` of given `name`. Disabled tasklet is provided by `DECLARE_TASKLET_DISABLED(name, func, data)`.

`void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);`

Initialize a `tasklet_struct` structure. This function is suitable for dynamically allocated or reinitialized tasklets.

`int tasklet_trylock(struct tasklet_struct *t);`

Try to lock the tasklet: try to set `TASKLET_STATE_RUN` to `t->state` using atomic operation. Returns true if the tasklet has been locked before the operation.

`void tasklet_unlock(struct tasklet_struct *t);`

Unlock the tasklet using atomic operation.

`void tasklet_unlock_wait(struct tasklet_struct *t);`

Wait for tasklet to become unlocked. Warning: busy-loop implementation, this function does not sleep.

`void tasklet_schedule(struct tasklet_struct *t);`

Activate the tasklet, i.e. put it into the beginning of tasklet list for current CPU.

`void tasklet_hi_schedule(struct tasklet_struct *t);`

Activate the tasklet, i.e. put it into the beginning of high priority tasklet list for current CPU.

`void tasklet_disable_nosync(struct tasklet_struct *t);`

Disable the tasklet, do not wait for termination of tasklet if it's currently running.

`void tasklet_disable(struct tasklet_struct *t);`

Disable the tasklet, but wait for termination of tasklet if it's currently running.

`void tasklet_enable(struct tasklet_struct *t);`

Enable the disabled, but scheduled tasklet.

`void tasklet_kill(struct tasklet_struct *t);`

Kill a tasklet. This is done by clearing the `state`'s flag `TASKLET_STATE_SCHED`.

`void tasklet_kill_immediate(struct tasklet_struct *t, unsigned int cpu);`

Kill a tasklet. This is done by clearing the `state`'s flag `TASKLET_STATE_SCHED`. Moreover, the tasklet is immediately removed from the list. The `cpu` must be in `CPU_DEAD` state.

Note that scheduled tasklets are kept in per-CPU arrays (one for normal and the second one for high priority tasklets). Before the tasklet is executed its `TASKLET_STATE_SCHED` flag is cleared. If the tasklet wants to be executed in the future, it's `TASKLET_STATE_SCHED` flag must be set.

Chapter 7

Process Management and Scheduling

Good process management is feature needed by every multitasking OS. This category contains not only process creation and termination, but also switching between processes, process state management, in-kernel preemption, kernel threads and many other things. And everything must be done with maximal efficiency.

The process (or thread or task) itself may be in one of many states:

`TASK_RUNNING`
: The task is able to run, i.e. it will run when some CPU has time to run it.

`TASK_INTERRUPTIBLE`
: This task is sleeping, but it's possible to wake it up using signals or expiration of a timer.

`TASK_UNINTERRUPTIBLE`
: This task is sleeping, but it can't be woken up by signals or timers.

`TASK_STOPPED`
: Task is stopped either due to job control or `ptrace()`.

`TASK_TRACED`
: This task is being traced.

`TASK_NONINTERACTIVE`
: This task won't be given any penalty or priority-boost due to average sleep time.

The tasks may be even frozen, depending on power management state.

`set_task_state(task, state)`
: This macro sets the `state` to given `task`. The state should not be directly assigned, this function uses memory barrier.

`set_current_state(state)`
: This macro sets state to task currently running on local CPU.

Some structures that are used in many parts of process management are described here for easier reference later.

`struct task_struct;`
: This very complicated structure holds all information concerning one process.

`struct sched_group;`
: Scheduling group, the load balancing algorithm tries to equally divide load between groups in a single domain.

`struct sched_domain;`
: Scheduling domain create a tree. Each CPU is represented by its own domain and is a leaf in the domain tree. This approach allows to make very complicated load balancing even on NUMA machines.

`struct user_struct;`
: A structure for each user, which holds number of files opened, processes running, pending signals, inotify watches etc. and `uid`.

`struct thread_struct;`
: Low level per-architecture defined structure containing copy of some CPU registers and similar non-portable things.

`struct fs_struct;`
: This structure represents a file system, it contains pointer to `root`, `pwd` (current working directory) directory entries etc.

`struct files_struct;`
: Per-process structure of opened files.

`struct dentry;`
: This structure represents directory entry.

`struct mm_struct;`
: Memory management information for a process, this may be shared by threads in one process.

`struct vm_area_struct;`

A VM area is a part of process virtual memory with a special rule for the page-fault handlers. This structure describes one area.

There is one function that is mentioned fairly often within following sections: `schedule()`. This function interrupts execution of current task and switches to other one.

7.1 Scheduler

The role of the scheduler in multitasking (and multithreading) operating system is to switch between tasks and to decide when to switch, which task will be woken up and how long will it run until next task is woken up.

Scheduler also does load balancing on SMP/NUMA architectures. The goal of load balancing is to divide computations equally to the CPU nodes.

`NICE_TO_PRIO(nice)` maps nice-value to static priority for internal use, `PRIO_TO_NICE(prio)` maps static priority back to nice-value and `TASK_NICE(p)` returns nice-value of task `p`. The classic nice-value is converted to user static priority, which can be easier to use internally in scheduler routines. However dynamic priority is used when assigning timeslices to tasks.

The minimum timeslice is `MIN_TIMESLICE` (currently 5 msecs or 1 jiffy, which one is more), default timeslice is `DEF_TIMESLICE` (100 msecs), maximum timeslice is 800 msecs. Timeslices get refilled after they expire. Tasks are executed in timeslices, when all tasks run out of their assigned time they get refilled. Dynamic priority is based on static priority, slightly modified by bonuses/penalties. Interactive tasks get bonuses (we estimate that task is interactive, if it sleeps sometimes) and batch tasks (also called cpu hogs) get penalties.

Each cpu has its own runqueue, which holds lists of tasks being run on this cpu. There are two priority arrays, one for active tasks and the other for task with already expired timeslices. When task runs out of its timeslice, we usually put it to expired array. If a task is interactive, we reinsert it in the active array after its timeslice has expired. However it wil not continue to run immediately, it will still roundrobin with other interactive tasks. For each task a `sleep_avg` value is stored, which is increased by one for each jiffy during sleep and decreased while running.

`SCALE_PRIO` scales dynamic priority values `[ -20 ... 19 ]` to time slice values. The higher a thread's priority, the bigger timeslices it gets during one round of execution. But even the lowest priority thread gets `MIN_TIMESLICE` worth of execution time. `int task_timeslice(task_t *p)` is the interface that is used by

the scheduler. Dynamic priority can be at most 5 points more or less (depends on interactivity) than static priority. [1]

The task uses its timeslice in intervals of max `TIMESLICE_GRANULARITY` msecs and then the scheduler tries to execute other tasks of equal priority from the runqueue[2]. Thus the task uses its timeslice in parts (if the timeslice is long enough) before being dropped from the active array).

Priority array contains list queue for each priority. The bitmap array keeps in mind which list is empty, `nr_active` is number of active tasks. As a locking rule is used the following: those places that want to lock multiple runqueues (such as the load balancing or the thread migration code), lock acquire operations must be ordered by ascending `&runqueue`. To make it simpler, `static void double_rq_lock(runqueue_t *rq1, runqueue_t *rq2)` and `static void double_rq_lock(runqueue_t *rq1, runqueue_t *rq2)` are provided.

Runqueue (`runqueue_t` data type) is the main per-cpu data structure holding number of running tasks, cpu load (only when compiled with `CONFIG_SMP`, used when migrating tasks between cpus to determine which cpu is heavily loaded and which one is almost idle), number of context switches, expired timestamp and the best expired priority priority (used to determine when is the time to return tasks from expired array to active array, so that interactive tasks won't starve non-interactive), number of uninterruptible tasks, timestamp of last tick, pointers to current and idle tasks, pointer to previously used `mm_struct` (used when switchig kernel threads to find out that sometimes there's no need to switch address space), number processes waiting for I/O, and the following under the assumption of `CONFIG_SMP` being defined: scheduling domain for this CPU/runqueue, whether balancing is active for this runqueue (`active_balance`), `push_cpu`, `migration_thread` and `migration_queue`.

Following macros are being defined for accessing runqueues: `cpu_rq(cpu)` representes runqueue of given cpu, `this_rq()` represents runqueue of this processor, `task_rq(p)` finds runqueue belonging to task `p` and finally `cpu_curr(cpu)` is currently executed task on given cpu.

One of the changes introduced in kernel 2.6 is preemption. Prior to this the task switching was possible in kernel only when process voluntarily rescheduled. With

[1] This approach is often criticised, especially the estimating whether a task is interactive. There are attempts to write better scheduler/patch the existing one (e.g. Con Kolivas's patches, http://kerneltrap.org/node/view/780). Too little time for minimal timeslice could cause many context switches and thus decreasing performance as the CPU cache is often reloaded. This approach is in common better than the one in 2.4 and older kernels.

[2] This was added because applications like audio players tend to send e.g. 50ms buffer to soundcard and then sleep, and if other task gains 100ms timeslice this led to audio dropouts

preemption it is to switch task at almost any time. Preemptions uses SMP locking mechanisms to avoid concurrency and reentrancy problems. See "Locking" chapter for details about preemption.

Next problem is protecting CPU state, e.g. FPU state: FPU should be used only with preemption disabled (the kernel does not save FPU state except for user tasks, so preemption could change the contents of FPU registers). `kernel_fpu_begin()` and `kernel_fpu_end()` should be used around areas that use FPU. Note: some FPU functions are already preempt-safe.

Externally visible scheduler statistics are in `struct kernel_stat kstat;`, which is defined per cpu. Exported symbol.

7.1.1 Priority

`static int effective_prio(task_t *p);`

Return the effective priority, i.e. the priority based on static priority but modified by bonuses/penalties. Bonus/penalty range is +-5 points, thus we use 25% of the full priority range. This ensures us that the +19 interactive tasks do not preempt 0 nice-value cpu intensive tasks and -20 tasks do not get preempted by 0 nice-value tasks.

`static void recalc_task_prio(task_t *p, unsigned long long now);`

Recalculates priority of given task. User tasks that sleep a long time are categorised as idle and will get just interactive status to stay active and to prevent them suddenly becoming cpu hogs and starving other processes. The lower the `sleep_avg` a task has the more rapidly it will rise with sleep time. Tasks with low `interactive_credit` are limited to one timeslice worth of `sleep_avg` bonus. Tasks with credit less or equal than `CREDIT_LIMIT` waking from uninterruptible sleep are limited in their `sleep_avg` rise as they are likely to be cpu hogs waiting on I/O. This code gives a bonus to interactive tasks. The boost works by updating the 'average sleep time' value here, based on `->timestamp`. The more time a task spends sleeping, the higher the average gets - and the higher the priority boost gets as well.

`void set_user_nice(task_t *p, long nice);`

Set nice value of task `p`. Lock of runqueue the task belongs in is locked during execution. Exported symbol.

`int can_nice(const task_t *p, const int nice);`

Check whether task `p` can reduce its `nice` value.

`asmlinkage long sys_nice(int increment);`

Add to the priority of the current process `increment` value. This function is available only with `__ARCH_WANT_SYS_NICE` defined.

`int task_prio(const task_t *p);`

Return the priority of a given task. Realtime tasks are offset by -200, normal tasks are arounf 0, value goes from -16 to +15, this is the priority value as seen by users in /proc

`int task_nice(const task_t *p);`

Return the nice value of a given task. Exported symbol.

`asmlinkage long sys_sched_setparam(pid_t pid, struct sched_param __user *param);`

Set/change the realtime priority of a thread to the values obtained from user-space.

`asmlinkage long sys_sched_getparam(pid_t pid, struct sched_param __user *param);`

Get the realtime priority of a thread and store it to user-space. `tasklist_lock` is read-locked during execution.

`asmlinkage long sys_sched_get_priority_max(int policy);`

Return maximum realtime priority that can be used by a given scheduling class.

`asmlinkage long sys_sched_get_priority_min(int policy);`

Return minimum realtime priority that can be used by a given scheduling class.

7.1.2 Runqueues

`static runqueue_t *task_rq_lock(task_t *p, unsigned long *flags);`

This function locks the runqueue for a given task and disables interrupts. Preemption is not disabled during execution of this function.

`static inline void task_rq_unlock(runqueue_t *rq, unsigned long *flags);`

This function unlocks the runqueue (possible locked by `task_rq_lock`)

`static runqueue_t *this_rq_lock(void);`

Lock runqueue, the same case as with `task_rq_lock()`: we need to disable

interrupts first and then lock the spinlock (due to SMP and preemption, details in [kernel/sched.c])

`static inline void rq_unlock(runqueue_t *rq);`
Unlock the runqueue.

`static void dequeue_task(struct task_struct *p, prio_array_t *array);`
Remove a task from priority array, i.e. remove it from appropriate list and (if needed) set the bit to indicate that list is empty.

`static void enqueue_task(struct task_struct *p, prio_array_t *array);`
Adds task to end of appropriate list and sets bit to indicate that the list is surely not empty.

`static inline void enqueue_task_head(struct task_struct *p, prio_array_t *array);`
Adds task to beginning of appropriate list and sets bit to indicate that the list is surely not empty. This is used by the migration code (that means migration between cpus), we pull tasks from the head of the remote queue so we want these tasks to show up at the head of the local queue.

`static inline void __activate_task(task_t *p, runqueue_t *rq);`
Move given task to given runqueue.

`static inline void __activate_idle_task(task_t *p, runqueue_t *rq);`
Move idle task to the front of runqueue.

`static void activate_task(task_t *p, runqueue_t *rq, int local);`
Move a task to the runqueue and do priority recalculation, update all the scheduling statistics stuff (sleep average calculation, priority modifiers, etc.)

`static void deactivate_task(struct task_struct *p, runqueue_t *rq);`
Remove a task from the runqueue.

`static void resched_task(task_t *p);`
`need_resched` flag, on SMP it might also involve a cross-cpu call to triger the scheduler on the target cpu. SMP version also does `preempt_disable` in the beginning and `preempt_enable` in the end.

`static runqueue_t *find_busiest_queue(struct sched_group *group);`
Find the busiest runqueue among the CPUs in group. Only on SMP.

Following functions are provided to prevent deadlock while locking runqueues' spinlocks:

`static void double_rq_lock(runqueue_t *rq1, runqueue_t *rq2);`
Safely lock two runqueues. This does not disable interrupts like `task_rq_lock`, you need to do so manually before calling. This function should be used to prevent deadlock when locking two runqueues.

`void double_lock_balance(runqueue_t *this_rq, runqueue_t *busiest);`
Lock the `busiest` runqueue, `this_rq` is locked already. During execution it may be unlocked and locked again later (to prevent deadlock). Only on SMP.

`static void double_rq_unlock(runqueue_t *rq1, runqueue_t *rq2);`
Safely unlock two runqueues, note that this does not restore interrupts.

7.1.3 Load Balancing and Migration

Following functions are intended for migration tasks between CPUs, therefore are conditionaly compiled only if `CONFIG_SMP` is defined. There are two possible types of request (`request_type`): `REQ_MOVE_TASK` (for migrating between processors) and `REQ_SET_DOMAIN` (for setting scheduling domain)

Following struct (`migration_req_t`) is used to hold migration request details. `type` keeps type of request: `REQ_MOVE_TASK` or `REQ_SET_DOMAIN`, `task` is the task that is being moved to the `dest_cpu`. The other case is setting the scheduling domain `sd`. Completion mechanism is used for executing migration request.

Scheduling domains are new approach to SMT/SMP/NUMA scheduling. Each cpu has its scheduling domain, these domains create a tree-like structure. Domain has `parent` pointer, the root domain's `parent` is `NULL`. Scheduling domains are per-cpu structures as they are locklessly updated. Each domain spans over a set of CPUs, registered in `cpumask_t span`, these CPUs belong to one or more `struct sched_group` (pointed to by `struct sched_group *groups`, which are organised as one-way circular list. The union of CPU masks of these groups must be identical to CPU mask of the domain and the intersection of any of these groups must be empty. Groups are (unlike domains) shared between CPUs, they are read-only after they've been set up.

Load balancing happens between groups. Each group has its load, which is sum of loads of CPUs that belong into this group. When the load of a group is out of balance, load balancing between groups occurs. `rebalance_tick()` is the function that is called by timer tick and checks if there is any need to do load balancing. If

is is so, balance the domain and then check the parent domain, again check for need for balancing and so forth.

Each scheduling domains can have set the following flags:

`SD_BALANCE_NEWIDLE`
: Balance when about to become idle.

`SD_BALANCE_EXEC`
: Balance on `exec()`, this is quite effective, since 'execed' task is not cache hot on any CPU.

`SD_BALANCE_CLONE`
: Balance on `clone()`.

`SD_WAKE_IDLE`
: Wake to idle CPU on task wakeup.

`SD_WAKE_AFFINE`
: Wake task to waking CPU

`SD_WAKE_BALANCE`
: Perform balancing at task wakeup.

`SD_SHARE_CPUPOWER`
: Domain members share cpu power, e.g. SMT architectures.

Migration routines:

`static int migrate_task(task_t *p, int dest_cpu, migration_req_t *req);`
: Tries to migrate task `p` to given `dest_cpu`. The task's runqueue lock must be held, returns `true` if you have to wait for migration thread.

`void kick_process(task_t *p);`
: Causes a process which is running on another CPU to enter kernel-mode, without any delay (to get signals handled).

`static inline unsigned long source_load(int cpu);`
: Returns a low guess at the load of a migration-source cpu. We want to underestimate the load of migration sources, to balance conservatively.

`static inline unsigned long target_load(int cpu);`
: Returns a high guess at the load of a migration-target cpu.

`struct sched_group *find_idlest_group(struct sched_domain *sd, struct task_struct *p, int this_cpu);`

Find and return the least busy CPU group within the domain `sd`.

`static int find_idlest_cpu(struct task_struct *p, int this_cpu, struct sched_domain *sd);`

Find the the idlest CPU i.e. least busy runqueue on SMP machine.

`static struct sched_group *find_busiest_group(struct sched_domain *sd, int this_cpu, unsigned long *imbalance, enum idle_type idle);`

Find and return the busiest cpu group within the domain. Calculate and return the number of tasks which should be moved to restore balance via the imbalance parameter. Only on SMP.

`int sched_balance_self(int cpu, int flag);`

Balance the current task running on given `cpu` in domains that have `flag` set. The least loaded group is selected. Target CPU number is returned or this CPU if no balancing is needed. Preemption must be disabled prior to calling this function.

`static void sched_migrate_task(task_t *p, int dest_cpu);`

If `dest_cpu` is allowed for this process, migrate the task to it. This is accomplished by forcing the `cpu_allowed` mask to only allow `dest_cpu`, which will force the cpu onto `dest_cpu`. Then the `cpu_allowed` mask is restored. Only on SMP.

`void sched_exec(void);`

Find the highest-level, exec-balance-capable domain and try to migrate the task to the least loaded CPU. `execve()` is a good opportunity, because at this point the task has the smallest effective memory and cachce footprint. Only on SMP machines.

`static inline void pull_task(runqueue_t *src_rq, prio_array_t *src_array, task_t *p, runqueue_t *this_rq, prio_array_t *this_array, int this_cpu);`

Move a task from a remote runqueue to the local runqueue, both runqueues must be locked. Only on SMP.

`static inline int can_migrate_task(task_t *p, runqueue_t *rq, int this_cpu, struct sched_domain *sd, enum idle_type idle);`

May the task `p` from runqueue `rq` be migrate onto `this_cpu` ? We can't migrate tasks that are running or tasks that can't be migrated due to `cpus_allowed` or are cache-hot on their current cpu. Only on SMP.

`static int move_tasks(runqueue_t *this_rq, int this_cpu, runqueue_t *busiest, unsigned long max_nr_move, struct sched_domain *sd, enum idle_typeidle);`

Try to move up to `max_nr_move` tasks from busiest to `this_rq`, as part of a balancing operation within domain. Returns the number of tasks moved. Must be called with both runqueues locked, no unlocking is being done. Only on SMP.

`static int load_balance(int this_cpu, runqueue_t *this_rq, struct sched_domain *sd, enum idle_type idle);`

Check `this_cpu` to ensure it is balanced within domain. Attempt to move tasks if there is an imbalance. Must be called with `this_rq` unlocked. Locks of `this_rq` and busiest runqueue in the domains may be locked during executions, but they are finally unlocked. Only on SMP.

`static int load_balance_newidle(int this_cpu, runqueue_t *this_rq, struct sched_domain *sd);`

Check `this_cpu` to ensure it is balanced within domain. Attempt to move tasks if there is an imbalance. Called from schedule when `this_rq` is about to become idle (`NEWLY_IDLE`). Beware: `this_rq` is being locked and not unlocked upon return. Only on SMP.

`static inline void idle_balance(int this_cpu, runqueue_t *this_rq);`

This function is called by `schedule()` if `this_cpu` is about to become idle. It attempts to pull tasks from other CPUs on SMP, on UP does nothing.

`static void active_load_balance(runqueue_t *busiest, int busiest_cpu);`

This function is run by migration threads. It pushes a running task off the cpu. It can be required to correctly have at least 1 task running on each physical CPU where possible, and not have a physical / logical imbalance. Must be called with busiest locked, however the lock is a few times relocked to ensure there'll be no deadlock.

`static void rebalance_tick(int this_cpu, runqueue_t *this_rq, enum idle_type idle);`

This function will get called every timer tick, on every CPU on SMP. Check each scheduling domain to see if it is due to be balanced, and initiates a balancing operation if so. Balancing parameters are set up in `arch_init_sched_domains`. On uniprocessors this function does nothing.

`task_t *idle_task(int cpu);`
Return idle task for given `cpu`.

`static void __setscheduler(struct task_struct *p, int policy, int prio);`
Change priority, runqueue lock must be held.

`asmlinkage long sys_sched_getscheduler(pid_t pid);`
Get the policy (scheduling class) of a thread. `tasklist_lock` is read-locked during execution.

`static int setscheduler(pid_t pid, int policy, struct sched_param __user *param);`
Change the scheduling policy and/or realtime priority of a task to the values obtained from user-space. During execution are disabled irqs, read-locked `tasklist_lock`, locked runqueue lock of appropriate task.

`asmlinkage long do_sched_setscheduler(pid_t pid, int policy, struct sched_param __user *param);`
Set/change the scheduler policy and realtime priority to the values obtained from usermode.

`asmlinkage long sys_sched_setscheduler(pid_t pid, int policy, struct sched_param __user *param);`
Wrapper for `do_sched_setscheduler()`.

`asmlinkage long sys_sched_setaffinity(pid_t pid, unsigned int len, unsigned long __user *user_mask_ptr);`
Set the cpu affinity of a process. Values are obtained from user-space.

`cpumask_t cpu_present_map;`
This represents all cpus present in the system. In systems capable of hotplugging, this map could dynamically grow as new cpus are detected in the system via any platform specific method. Exported symbol.

On SMP following cpu mask maps (`cpumask_t`) are declared: `cpu_online_map`, which is map of all online CPUs (initally `CPU_MASK_ALL`), `cpu_possible_map`, which

is map of all possible CPUs (initially `CPU_MASK_ALL`) and `nohz_cpu_mask`, which is map of CPUs with switched-off Hz timer (initally and on systems that do not switch off the Hz timer `CPU_MASK_NONE`)

`long sched_getaffinity(pid_t pid, cpumask_t *mask)`
: Get the cpu affinity of a process. Cpu hotplugging is locked and `tasklist_lock` is read-locked.

`asmlinkage long sys_sched_getaffinity(pid_t pid, unsigned int len, unsigned long __user *user_mask_ptr);`
: Wrapper for `sched_getaffinity()`. The mask is copied to userspace.

`int set_cpus_allowed(task_t *p, cpumask_t new_mask);`
: Change a given task's CPU affinity. Migrate the thread to a proper CPU (if needed i.e. the current task's cpu is not enabled in the `new_mask`) and schedule it away if the CPU it's executing on is removed from the allowed bitmask. The caller must have a valid reference to the task, the task must not `exit()` & deallocate itself prematurely. The call is not atomic, no spinlocks may be held. Task's runqueue lock is being held during execution and irqs are disabled. Exported symbol, SMP only.

`static void __migrate_task(struct task_struct *p, int src_cpu, int dest_cpu);`
: Move (not current) task off this cpu, onto dest cpu. We're doing this because either it can't run here any more (`set_cpus_allowed()` away from this cpu, or cpu going down when using hotplug), or because we're attempting to rebalance this task on exec (`sched_balance_exec`). Both source and destination CPUs' runqueue-locks are held. SMP only.

`static int migration_thread(void * data);`
: This is a high priority system thread that performs thread migration by bumping thread off cpu then 'pushing' onto another runqueue. `data` runqueue's lock is held during execution. SMP only.

`static void move_task_off_dead_cpu(int dead_cpu, struct task_struct *tsk);`
: This function moves task `tsk` from CPU `dead_cpu` that went offline.

`static void migrate_nr_uninterruptible(runqueue_t *rq_src);`
: Adds `nr_uninterruptible` statistics from (`rq_src`) to any online CPU. Sometimes `nr_uninterruptible` is not updated for performance reasons and this

way the sum of numbers of `TASK_UNINTERRUTIBLE` tasks on all CPUs is correct. IRQs are disabled and runqueues' locks are locked while updating statistics.

`static void migrate_live_tasks(int src_cpu);`
Run through task list on `src_cpu` CPU and migrate tasks from it. IRQs are blocked and `tasklist_lock` is being held during this operation.

`void sched_idle_next(void);`
Schedule idle task to be the next runnable task on current CPU. It's done by boosting its priority to highest possible and adding it to the front of runqueue. Used by cpu offline code. Current cpu's runqueue lock is held during execution and interrupts are disabled. SMP only.

`void idle_task_exit(void);`
Ensures that the idle task is using `init_mm` right before its cpu goes offline.

`static void migrate_dead(unsigned int dead_cpu, task_t *tsk);`
Migrate exiting task from dead CPU. The runqueue lock must be held before calling this function and released afterwards. This lock is dropped around migration.

`static void migrate_dead_tasks(unsigned int dead_cpu);`
Migrate exiting tasks from dead CPU.

`int migration_call(struct notifier_block *nfb, unsigned long action, void *hcpu);`
This is a callback that gets triggered when a CPU is added or removed. Migration thread for this new CPU is started and stopped here and tasks are migrated from dying CPU.

`static int migration_call(struct notifier_block *nfb, unsigned long action, void *hcpu);`
Callback that gets triggered when a CPU is added. Here we can start up the necessary migration thread for the new CPU. When a new cpu goes up, new migration thread is created and bind to that cpu. That cpu's rq gets locked for a while. When a new cpu goes online, wake up migration thread. If cpu's going up was canceled (this could happen only with cpu hotplugging) stop the migration thread. When cpu dies, migrate all tasks somewhere else and stop migration thread. Dead cpu's runqueue gets locked for a while. SMP only.

`int __init migration_init(void);`
Start one migration thread for boot cpu and register cpu migration notifier.

This thread has the highest priority so that task migration `migrate_all_tasks` happens before everything else.

`static int sd_degenerate(struct sched_domain *sd);`
Return true if scheduling domain is not degenerate. Degenerate domain either contains only one CPU, or if it supports internal load balancing and has only one group, or if it has set any balacing flag concerning waking processes.

`static int sd_parent_degenerate(struct sched_domain *sd, struct sched_domain *parent);`
Check whether the `parent` is not degenerated and compare CPU sets of parent and `sd` and flags to find degeneration.

`void cpu_attach_domain(struct sched_domain *sd, int cpu);`
Attach the scheduling domain to given `cpu` as its base domain. CPU hotplug gets locked during execution, so does given `cpu`'s runqueue lock and irqs are disabled. Only on SMP.

`void init_sched_build_groups(struct sched_group groups[], cpumask_t span, int (*group_fn)(int cpu));`
Build a circular linked list of the groups covered by the given span.

`int __init migration_cost_setup(char *str);`
This function sets up a migration cost table: `unsigned long long migration_cost[MAX_DOMAIN_DISTANCE];` This table is a function of 'domain distance', i.e. the number of steps (when not implicitly given on command line, this table is boot-time calibrated).

`int __init setup_migration_factor(char *str);`
Set migration factor, i.e. retrieve the option given on kernel command line. The migration factor is given in percents and measures the cost of migration.

`unsigned long domain_distance(int cpu1, int cpu2);`
Estimated distance of two CPUs. Return value is the number of domains on way between CPUs.

`void touch_cache(void *__cache, unsigned long __size);`
Dirty a big buffer, so that L2 cache will miss and filled with values from this buffer.

`static unsigned long long measure_one(void *cache, unsigned long size, int source, int target);`
This function returns the cache-cost of one task migration in nanoseconds.

`unsigned long long measure_cost(int cpu1, int cpu2, void *cache, unsigned int size)`
; Measure the cache-cost multiple times for different cache sizes using `measure_one()` between `cpu1` and `cpu2`. Then measure the cost on each CPU itself and the difference is the extra cost that the task migration brings.

`unsigned long long measure_migration_cost(int cpu1, int cpu2);`
Measure migration cost between two given CPUs.

`void calibrate_migration_costs(const cpumask_t *cpu_map);`
This function calibrates migration costs for a set of CPUs.

`int find_next_best_node(int node, unsigned long *used_nodes);`
Find the next node to include in a given scheduling domain.

`cpumask_t sched_domain_node_span(int node);`
Get a CPU mask for a node's scheduling domain.

`static void __init arch_init_sched_domains(void);`
Set up domains for each cpu and groups for each node. Assumptions: `CONFIG_SMP` and not `ARCH_HAS_SCHED_DOMAIN`

If `ARCH_HAS_SCHED_DOMAIN` is not defined then `struct sched_group sched_group_cpus[NR_CPUS]` contains schedule group information for each cpu and for each CPU is declared `struct sched_domain cpu_domains`. Since these structures are almost never modified but they are fairly often accessed, they're copied for each CPU. If `CONFIG_NUMA` defined and `ARCH_HAS_SCHED_DOMAIN` undefined then `struct sched_group sched_group_nodes[MAX_NUMNODES]` contains schedule groups for each node and the `struct sched_domain` contains schedule domains for each cpu in `node_domains`.

7.1.4 Task State Management and Switching

`unsigned long nr_running(void);`
Return number of running processes.

`unsigned long nr_uninterruptible(void);`
Return number of uninterruptible processes.

`unsigned long long nr_context_switches(void);`
Return number of context switches since bootup.

`unsigned long nr_iowait(void);`

Return number of processes waiting for I/O.

`void wait_task_inactive(task_t * p);`

Waits for a thread to unschedule. The caller must ensure that the task will unschedule sometime soon, else this function might spin for a long time. This function can't be called with interrupts off, or it may introduce deadlock with `smp_call_function()` if an interprocessor interrupt is sent by the same process we are waiting to become inactive.

`static int wake_idle(int cpu, task_t *p);`

This function (conditonally compiled under assumption `ARCH_HAS_SCHED_WAKE_IDLE` being defined) is useful on SMT architectures to wake a task onto an idle sibling if we would otherwise if we would otherwise wake it onto a busy sibling inside a scheduling domain. Returns the cpu we should wake onto. The other version (`ARCH_HAS_SCHED_WAKE_IDLE` undefined) returns just cpu from argument.

`static int try_to_wake_up(task_t * p, unsigned int state, int sync);`

Wakes up a process. The arguments are: thread to be woken up (`p`), the mask of task states that can be woken (`state`) and whether awakening should be synchronous (`sync`). Put it on the run-queue if it's not already there. The current thread is always on the run-queue (except when the actual re-schedule is in progress), and as such you're allowed to do the simpler `current->state=TASK_RUNNING` to mark yourself runnable without the overhead of this. Returns failure only if the task is already active.

`int fastcall wake_up_process(task_t * p);`

Just a wrapper to wake up a process using call to `try_to_wake_up()` without synchronisation, state mask to be woken allows to wake stopped, interruptible or uninterruptible processes. This symbol is exported.

`int fastcall wake_up_state(task_t *p, unsigned int state);`

Simple wrapper to call `try_to_wake_up(p, state, 0);`

`void fastcall sched_fork(task_t *p);`

Performs scheduler related setup for a newly forked process `p`. Process `p` is forked by current process. Timeslice of current proces is split in two, one half goes to forked process `p`. The remainder of the first timeslice might be recovered by the parent if the child exits early enough.

Local IRQs are being disabled inside this function and in the end enabled, preemption can also be disabled sometimes. This function also locks `spin_lock_init(&p->switch_lock)` but it doesn't unlock it.

`void fastcall wake_up_new_task(task_t *p, unsigned long clone_flags)`
Put the `p` task to runqueue, wake it up and do some initial scheduler statistics housekeeping that must be done for every newly created process. Locks `task_rq_lock(current, &flags)` and unlocks it in the end.

`void fastcall sched_exit(task_t * p);`
Retrieves timeslices from exiting `p`'s children back to parent. This way the parent does not get penalized for creating too many threads. Disables local irqs and locks runqueue of `p`'s parent during execution.

`void prepare_task_switch(runqueue_t *rq, task_t *next);`
Prepare to switch tasks from `rq` runqueue to `next`: locking and architecture specific things. This should be called with runqueue lock held and interrupts disabled. When the switch is finished, `finish_task_switch()` must be called.

`void finish_task_switch(runqueue_t *rq, task_t *prev);`
This function must be called after the context switch with runqueue lock held and interrupts disabled. Locks locked by `prepare_task_switch()` are unlocked and architecure specific things are done.

`void finish_task_switch(task_t *prev);`
Does some clean-up after task-switch. Argument prev is the thread we just switched away from. We enter this with the runqueue still locked, and `finish_arch_switch()` (done in this function) will unlock it along with doing any other architecture-specific cleanup actions. Note that we may have delayed dropping an mm in `context_switch()`. If so, we finish that here outside of the runqueue lock by calling `mmdrop()` (Doing it with the lock held can cause deadlocks, see `schedule()` for details.)

`void schedule_tail(task_t *prev);`
This is the first thing a freshly forked thread must call, the argument prev is the thread we just switched away from. Calls `finish_task_switch()`.

`static inline task_t *context_switch(runqueue_t *rq, task_t *prev, task_t *next);`
Switch to the new MM and the new process's register state. Return previous task.

`static inline int wake_priority_sleeper(runqueue_t *rq);`

If an SMT sibling task has been put to sleep for priority reasons reschedule the idle task to see if it can now run.

`void update_cpu_clock(task_t *p, runqueue_t *rq, unsigned long long now);`

Update task's time since the last tick/switch.

`unsigned long long current_sched_time(const task_t *tsk);`

Return time since last tick/switched that wasn't added to `p->sched_time`.

`void account_user_time(struct task_struct *p, cputime_t cputime);`

Account user cpu time to a process.

`void account_system_time(struct task_struct *p, int hardirq_offset, cputime_t cputime);`

Account system cpu time to a process.

`void account_steal_time(struct task_struct *p, cputime_t steal);`

Account for involuntary wait time.

`void scheduler_tick(int user_ticks, int sys_ticks);`

This function gets called by the timer code, with HZ frequency. We call it with interrupts disabled. It also gets called by the fork code, when changing the parent's timeslices. This CPU's runqueue lock is locked during execution.

`void wakeup_busy_runqueue(runqueue_t *rq);`

If an SMT runqueue `rq` is sleeping due to priority reasons wake it up.

`unsigned long smt_slice(task_t *p, struct sched_domain *sd);`

Return number of timeslices lost to process running on multi-thread sibling execution unit. (SMT CPU's threads don't run always at same speed, e.g. shared FPU...)

`static inline void wake_sleeping_dependent(int cpu, runqueue_t *rq);`

Only on SMT architectures, otherwise dummy function. The function reschedules idle tasks on cpus within domain of given runqueue. If an SMT sibling task is sleeping due to priority reasons wake it up now.

`static inline int dependent_sleeper(int cpu, runqueue_t *rq, task_t *p);`

Only on SMT architectures, otherwise dummy. If a user task with lower static

priority than the running task on the SMT sibling is trying to schedule, delay it till there is proportionately less timeslice left of the sibling task to prevent a lower priority task from using an unfair proportion of the physical cpu's resources. Reschedule a lower priority task on the SMT sibling, or wake it up if it has been put to sleep for priority reasons.

`asmlinkage void __sched schedule(void);`

This is the main scheduler function. First we manage here gathering interactive credit of task currently being unscheduled, then we check if we've not run out tasks in our runqueue, if it is so do balance. If all tasks assigned this CPU are in expired array then switch expired and active array and set best expired priority to hightest (`rq->best_expired_prio = MAX_PRIO;`) so that high priority tasks will be executed first (that also means interactive tasks have some advantage due to higher dynamic priority). Note: We must be atomic when calling this function, preemption is being disabled inside and this CPU runqueue's lock is held. Exported symbol.

`asmlinkage void __sched preempt_schedule(void);`

Only with preemptible kernels. This is is the entry point to `schedule()` from `preempt_enable()` if there is need to reschedule. Jump out if irqs are disabled or `preempt_count()` is non-zero. Exported symbol.

`asmlinkage void __sched preempt_schedule_irq(void);`

This is improved `preempt_schedule()`, which enables local IRQs before calling `schedule()` and disables them afterwards.

`int default_wake_function(wait_queue_t *curr, unsigned mode, int sync, void *key);`

Try to wake up current task from `curr` waitqueue. Exported symbol.

`void fastcall complete(struct completion *x);`

Wake up task from `x->wait` and increase `x->done`. `x->wait.lock` gets locked (and irqs disabled) before and unlocked (and irqs restored) after waking up a task. Exported symbol.

`void fastcall complete_all(struct completion *x);`

Wake up task from `x->wait` and increase `x->done` by `UINT_MAX/2` value. `x->wait.lock` gets locked (and irqs disabled) before and unlocked (and irqs restored) after waking up a task. Exported symbol. Exported symbol.

`void fastcall __sched wait_for_completion(struct completion *x)`
If completion is not 'done' then create waitqueue (with `WQ_FLAG_EXCLUSIVE`) and chain `x->wait` to the tail of this new waitqueue. Set current task's state to `TASK_UNINTERRUPTIBLE` and call schedule until we're done. `x->wait.lock` is held inside but unlocked before each `schedule()` call and locked upon return. Also unlocked when exiting function. Exported symbol.

`unsigned long fastcall __sched wait_for_completion_timeout(struct completion *x, unsigned long timeout);`
Improved version of `__sched wait_for_completion()` that uses `schedule_timeout()`.

`unsigned long fastcall __sched wait_for_completion_interruptible(struct completion *x);`
Improved version of `__sched wait_for_completion()` that uses interruptible sleep.

`unsigned long fastcall __sched wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout);`
Improved version of `__sched wait_for_completion()` that uses interruptible sleep and `schedule_timeout()`.

`void fastcall __sched interruptible_sleep_on( wait_queue_head_t *q );`
Create new waitqueue, add current task, chain with `q`, call `schedule()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

`long fastcall __sched interruptible_sleep_on_timeout( wait_queue_head_t *q, long timeout);`
Create new waitqueue, add current task, mark it as `TASK_INTERRUPTIBLE`, chain with `q`, call `schedule_timeout()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

`void fastcall __sched sleep_on(wait_queue_head_t *q);`
Create new waitqueue, add current task, mark it as `TASK_UNINTERRUPTIBLE`, chain with `q`, call `schedule()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

`long fastcall __sched sleep_on_timeout(wait_queue_head_t *q, long timeout);`

Create new waitqueue, add current task, mark it as `TASK_UNINTERRUPTIBLE`, chain with `q`, call `schedule_timeout()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

`asmlinkage long sys_sched_yield(void);`

Yield the current processor to other threads by moving the calling thread to the expired array. Realtime tasks will just roundrobin in the active array. If there are no other threads running on this CPU then this function will return. This CPU's runqueue spinlock is locked during execution.

`void __sched __cond_resched(void);`

Sets current task's state to `TASK_RUNNIG` and calls `schedule()`. Exported symbol.

`int cond_resched_lock(spinlock_t *lock);`

If a reschedule is pending, drop the given lock, `schedule()`, and on return reacquire the lock. Exported symbol.

`int __sched cond_resched_softirq(void);`

Conditionally reschedule with SoftIRQs enableed. Exported symbol.

`void __sched yield(void);`

Yield the current processor to other threads. It is done via marking thread runnable and calls `sys_sched_yield()`. Exported symbol.

`void __sched io_schedule(void);`

Mark this task as sleeping on I/O. Increment `rq->nr_iowait` so that process accounting knows that this is a task in I/O wait state. Don't do that if it is a deliberate, throttling IO wait (this task has set its `backing_dev_info`: the queue against which it should throttle). Exported symbol.

`long __sched io_schedule_timeout(long timeout);`

The same as `io_schedule` but limited with `timeout`.

`asmlinkage long sys_sched_rr_get_interval(pid_t pid, struct timespec __user *interval);`

Return the default timeslice od a process into the user-space buffer. A value of `0` means infinity. `tasklist_lock` is locked during execution.

`static inline struct task_struct *eldest_child(struct task_struct *p);`

Return the eldest child of a task.

`static inline struct task_struct *older_sibling(struct task_struct *p);`

Return older sibling of a task.

`static inline struct task_struct *younger_sibling(struct task_struct *p);`

Return younger sibling of a task.

`static void show_task(task_t * p);`

Print (via printk) some information about given task: pid, parent's pid, siblings, eldest child process...

`void show_state(void);`

Calls show_task on all threads of all processes. `tasklist_lock` is being held during execution.

`void __devinit init_idle(task_t *idle, int cpu);`

Move `idle` task to given cpu and make it idle (however the task is still marked `TASK_RUNNING`). Then the task si rescheduled. Both source and destination runqueues are locked and irqs are disabled.

`void __might_sleep(char *file, int line);`

Helper funtion that writes warning about call to sleeping function from invalid context and dumps stack if IRQs are disabled and preemption disabled and kernel not locked. Simple timeout is used to prevent message flooding. Exported symbol. Only when `CONFIG_DEBUG_SPINLOCK_SLEEP` defined.

`void __init sched_init_smp(void);`

Call `arch_init_sched_domains()` and `sched_domain_debug()` on SMP, otherwise dummy function.

`void __init sched_init(void);`

Initialize runqueues and the first thread. The boot idle thread does lazy MMU switching as well

`int in_sched_functions(unsigned long addr);`

Return true if `addr` is in memory range belonging to scheduler functions and false otherwise.

`task_t *curr_task(int cpu);`

Return the `current` task for given `cpu`

`void set_curr_task(int cpu, task_t *p);`
Set the `current` task for a given `cpu`.

`signed long __sched schedule_timeout(signed long timeout);`
Sleep `current` task until `timeout` expires.

`signed long __sched schedule_timeout_interruptible(signed long timeout)`
Mark task as `TASK_INTERRUPTIBLE` and sleep with `timeout`.

`signed long __sched schedule_timeout_uninterruptible(signed long timeout)`
Mark task as `TASK_UNINTERRUPTIBLE` and sleep with `timeout`.

`int idle_cpu(int cpu);`
Is a given cpu idle currently ? Exported symbol.

`inline int task_curr(const task_t *p);`
Is this task currently being executed on a CPU ?

7.2 Process Forking

Creation of new processes under unix-like operating systems is done by `fork()` system call. Linux also supports `clone()` syscall and since 2.6.16 also `unshare()` to *unshare* some parts of process context shared since `clone()`.

The simplest operation is `fork()`, which forks one process into two processes. Child receives zero return value and parental process receives the PID of the child.

The `clone()` syscall offers more options: the processes may share memory, file descriptors, signal handlers... complete list is in <`linux/sched.h`>.

Almost everything that can be shared using `clone()` can be later unshared by `unshare()` syscall, but some parts are not yet implemented.

Memory for task structures is allocated from slab caches: task_struct, signal_struct, sighand_struct, files_struct, fs_struct, vm_area_struct and mm_struct to speed up memory allocations.

`int nr_threads;`
Number of threads, the idle ones do not count. Protected by `write_lock_irq(&tasklist_lock)`.

`int max_threads;`
The maximal number of threads. Should be at least 20 (to be safely able to

boot), however the default maximum is so big that the thread structures can take up to half of memory.

`unsigned long total_forks;`
Total number of forks done since boot.

`unsigned long process_counts=0;`
Defined per CPU, holds number of processes for each CPU[3].

`rwlock_t tasklist_lock;`
Read-write lock protecting access to task list and `nr_threads`. Exported symbol.

`int nr_processes(void);`
Counts total number of processes (the sum of process_counts for each CPU).

`void *alloc_task_struct()`
Allocate memory for `task_struct`. Return value is pointer to allocated memory or NULL pointer in case of failure.

`void free_task(struct task_struct *tsk);`
Free memory allocated by `task_struct` and its `thread_info`.

`void __put_task_struct_cb(struct task_struct *tsk);`
This is the RCU callback to free `tsk`. Warn when task is dead, zombie or current.

`void put_task_struct(struct task_struct *t);`
Decrease reference counter and free task if counter has dropped to zero.

`void __init fork_init(unsigned long mempages);`
Create a slab on which `task_struct`s can be allocated and set maximum number of threads so that thread structures can take up to half of memory, but at least 20 threads. Set the resource current and maximal resource limit to `max_threads/2`

`static struct task_struct *dup_task_struct(struct task_struct *orig);`
Duplicate `task_struct` and its `thread_info`. Return `NULL` if memory allocation fails. If needed save FPU state and disable preemption while preparing to copy. Increase `task_struct->usage` by 2 (one for us, one for whoever does the `release_task()` (usually parent)).

[3]As far as I found this is not changed when process migrates from between CPUs, for exact numbers look at the runqueues

`static inline int dup_mmap(struct mm_struct *mm, struct mm_struct *oldmm);`

Write-lock `oldmm->mmap_sem` semaphore, duplicate `mm_struct` and initialize some fields and memory region descriptors of a `mm_struct`. Add it to the `mmlist` after the parent (`mmlist_lock` spinlock held). If `VM_DONTCOPY` is set then don't copy anything. Otherwise copy `mem_policy`, page table entries etc[4]. Locks `mm->page_table_lock` and `->vm_file->f_mapping->i_mmap_lock`. Return zero on success. Note: this function is fully implemented only with `CONFIG_MMU`, otherwise return 0.

`static inline int mm_alloc_pgd(struct mm_struct *mm);`

Allocate page directory structure, fill it with zeroes and if something fails return `-ENOMEM` otherwise zero. Note: this function is fully implemented only with `CONFIG_MMU`, otherwise return 0.

`static inline void mm_free_pgd(struct mm_struct *mm);`

Free page directory structure. Note: this function is fully implemented only with `CONFIG_MMU`, otherwise it is an empty macro.

`void *allocate_mm();`

Allocate a `struct mm_struct` from slab cache `mm_cachep`.

`void free_mm(struct mm_struct *mm);`

Free previously allocated `struct mm_struct`.

`static struct mm_struct *mm_init(struct mm_struct *mm);`

Initialize `mm_struct`: set number of users to one, unlock locks, and try to allocate page directory. If the last fails free `mm` memory (note: this memory is not allocated in this function) and return `NULL` otherwise `mm`.

`struct mm_struct *mm_alloc(void);`

Allocate, fill with zeros and initialize an `mm_struct`. Return NULL on failure and pointer to `mm_struct` on success.

`void fastcall __mmdrop(struct mm_struct *mm);`

Called when the last reference to the `mm` is dropped: either by a lazy thread or by `mmput`. Free the page directory and the `mm`. Warn when `mm` is `init_mm`, i.e. do not try to free the anchor of `mmlist` list.

[4]Too complex and too internal to spent more time on this function

`void mmput(struct mm_struct *mm);`

Decrement the use count and release all resources for an `mm`, exit all asynchronous I/O operations and release all mmaps. Spinlock `mmlist_lock` is held while `mm` is removed and `mmlist_nr` decreased.

`struct mm_struct *get_task_mm(struct task_struct *task);`

Return a `struct mm_struct` for given `task`. This function also increases reference counter for return `mm_struct`. `NULL` value is returned if `flags` contain `PF_BORROWED_MM` or if the task has no `mm`. The proper way to release a `mm_struct` acquired by this function is `mmput()`. `task->alloc_lock` is used to ensure mutual exclusion inside this function.

`void mm_release(struct task_struct *tsk, struct mm_struct *mm);`

This function is called after a `mm_struct` has been removed from the current process. Notify parent sleeping on `vfork()` (i.e. complete `tsk->vfork_done` completion, this disables IRQs and locks `tsk->vfork_done->lock` while completing).

`struct mm_struct *dup_mm(struct task_struct *task);`

Allocate a new `mm_struct` and copy from `task`'s `mm`.

`static int copy_mm(unsigned long clone_flags, struct task_struct *tsk)`

Copy `mm_struct` from current task to task `tsk`. If flag `CLONE_VM` is set then just set pointers `tsk->mm` and `tsk->active_mm` to `current->mm` and return 0.

If `CLONE_VM` si not set, then allocate memory for new `mm_struct` and copy the old one to it. Then call `mm_init()` on the new copy, initialize new context and call `dup_mmap()`. If anything of these fails, return non-zero value otherwise return zero.

`static inline struct fs_struct *__copy_fs_struct(struct fs_struct *old)`

Allocate memory for new `struct fs_struct`, increase use count, set lock as `RW_LOCK_UNLOCKED` and copy umask. The following is done with lock held for reading: copy info about root mount point and current working directory. If alternate root mount point is set, then copy it else make it `NULL`. Return pointer to newly allocated structure. If allocation failed in the beginning then return `NULL`.

`struct fs_struct *copy_fs_struct(struct fs_struct *old);`

Just a wrapper to call `__copy_fs_struct()`. Exported symbol.

`static inline int copy_fs(unsigned long clone_flags, struct task_struct *tsk);`

Copy `struct fs_struct` from current task to `task->fs`. If `CLONE_FS` is given then just increase usage counter in `current->fs->count`.

`static int count_open_files(struct files_struct *files, int size);`

Counts the number of open files by `files_struct`.

`struct files_struct *alloc_files();`

Allocate a new structure `files_struct`, set it's reference counter to one and initialize some of structure's members: next file descriptor, maximal count of FDs... and RCU head.

`struct files_struct *dup_fd(struct files_struct *oldf, int *errorp`

Allocate a new `files_struct` and copy contents from `oldf`.

`static int copy_files(unsigned long clone_flags, struct task_struct *tsk);`

If `clone_flags` contains `CLONE_FILES` then return with zero. Otherwise allocate memory, set next file descriptor to `0` and spinlock state to `SPIN_LOCK_UNLOCKED`. Also set maximal number of file descriptors, copy relevant data fields and increase usage counter for file descriptors. Spinlock `current->files->lock` is held inside while copying.

`int unshare_files(void);`

This is helper to call `copy_files()`, which isn't exported. Exported symbol.

`void sighand_free_cb(struct rcu_head *rhp);`

This is callback to free `struct sighand_struct` containing given `rcu_head`.

`void sighand_free(struct sighand_struct *sp);`

Free RCU-protected `struct sighand_struct`.

`static inline int copy_sighand(unsigned long clone_flags, struct task_struct *tsk);`

If flags contain either `CLONE_SIGHAND` or `CLONE_THREAD` increase usage counter and return, otherwise allocate memory for `sighand_struct`, initialize spinlock, set usage counter to `1` and copy action from `current->sighand->action`.

`static inline int copy_signal(unsigned long clone_flags, struct task_struct *tsk);`

If flags contain `CLONE_THREAD` increase usage counter and return, otherwise

allocate memory for `struct signal_struct` and fill initial values, then copy process group, tty and session. If the memory allocation fails return `-ENOMEM`, otherwise return zero.

`static inline void copy_flags(unsigned long clone_flags, struct task_struct *p);`
Clear `PF_SUPERPRIV` and set `PF_FORKNOEXEC` to `p->flags`. If `CLONE_PTRACE` is not set, then set `p->ptrace` to zero to disable ptrace for `p`.

`long sys_set_tid_address(int __user *tidptr);`
Set pointer to thread ID. This should be called from user space.

`struct task_struct *copy_process(unsigned long clone_flags, unsigned long stack_start, struct pt_regs *regs, unsigned long stack_size, int __user *parent_tidptr,int __user *child_tidptr);`
This creates a new process as a copy of the old one, but does not actually start it yet. It copies the registers, and all the appropriate parts of the process environment (as per the clone flags). The actual kick-off is left to the caller. In the beginning check corectness of the flags, such as that thread groups must share signals, detached threads can only be started from within group, if signal handlers are shared then VM must be implicitly shared too etc. Duplicate task struct and then check whether we're not going to cross resource limit (max. number of processes, usually set so that `task_struct` take no more than half of memory, but never less than 20). Then increase appropriate counters. Mark the process as yet not executed and copy flags. If flag `CLONE_IDLETASK` is not set (note: this is only for kernel only threads) then allocate a pid, if needed (flag `CLONE_PARENT_SETTID`)set tid in the parent. Initialize list of children and list of siblings and reset timers. Then copy `mempolicy` and afterwards via `security_task_alloc()` and `audit_alloc()` (system call auditing) check the operation. If it is ok then copy semaphores, files, fs, signals, memory structs, namespace and thread information. Set TID and whether clear TID on `mm_release`. Child also inherits execution from parent process. Now perform scheduler related setup for our newly forked process. Lock `tasklist_lock` for writing and check for `SIGKILL` signal (e.g. it is possible here to get `SIGKILL` from OOM kill). If `CLONE_PARENT` flag is set then reuse parent's parent as our parent. If `CLONE_THREAD` flag is set then lock `current->sighand->siglock` and check for group exit. Copy thread group ID from parent and also group leader and unlock `current->sighand->siglock`. Attach PID (if the process is a thread group leader, also attach TGID, PGID

and SID). Increase number of threads running `nr_threads` and finally release write-locked `tasklist_lock`. On success return zero, otherwise free what we allocated and return corresponding value.

`struct pt_regs *idle_regs(struct pt_regs *regs);`
Zero whole structure and return pointer to it.

`task_t *fork_idle(int cpu);`
Clone the process with zeroed registers and no stack start and set it as an idle process for given `cpu`. Then the process is unhashed from list of processes. This function is used only during SMP boot.

`static inline int fork_traceflag(unsigned clone_flags);`
Extract trace flags from argument.

`long do_fork(unsigned long clone_flags, unsigned long stack_start, struct pt_regs *regs, unsigned long stack_size, int __user *parent_tidptr, int __user *child_tidptr);`
This is the main fork routine. Check the trace flags modify if needed. Copy the task, but check the pid for quick exit (OOM kill, etc). If `CLONE_VFORK` is set then initialize completion and set `vfork_done` to this completion for the new task. If task is traced or stopped, start with immediate `SIGSTOP`. If task is not stopped, check whether it is a thread or a task and wake it up using appropriate call. In case of stopped task get CPU (this disables preempt) and set its internal state to `TASK_STOPPED`, move to cpu we get and put the cpu back (i.e. enable preempt). Increase number of forks done (for statistical reasons). If task is traced, notify the tracer. If the fork was initiated by the `vfork()` syscall, wait for completion of task and notify the tracer when the task has exited. If it was not `vfork()` then set thread flag to reschedule. If everything went all right, return pid.

`void __init proc_caches_init(void);`
Initialize `signal_cachep`, `sighand_cachep`, `files_cachep`, `fs_cachep`, `vm_area_cachep` and `mm_cachep`.

`void check_unshare_flags(unsigned long *flags_ptr);`
Check constraints on flags passed to `unshare()` syscall: unsharing a thread implies unsharing VM, unsharing VM implies unsharing signal handler. Unsharing signal handlers from task created using `CLONE_THREAD` implies unsharing a thread. Finally unsharing a namespace implies unsharing filesystem information.

`int unshare_thread(unsigned long unshare_flags);`
Return `-EINVAL` no `CLONE_THREAD` flag and zero otherwise. Unsharing of tasks created with `CLONE_THREAD` is not supported yet.

`int unshare_fs(unsigned long unshare_flags, struct fs_struct **new_fsp);`
If `CLONE_FS` flag is set, unshare file system information (represented by `struct fs_struct`).

`int unshare_namespace(unsigned long unshare_flags, struct namespace **new_nsp, struct fs_struct *new_fs);`
Unshare namespace if it's requested by `CLONE_NEWNS` flag.

`int unshare_sighand(unsigned long unshare_flags, struct sighand_struct **new_sighp);`
Unshare `sighand_struct` (signal handlers) if `CLONE_SIGHAND` is set.

`int unshare_vm(unsigned long unshare_flags, struct mm_struct **new_mmp);`
Unshare VM if `CLONE_VM` is set.

`int unshare_fd(unsigned long unshare_flags, struct files_struct **new_fdp);`
Unshare file descriptors if `CLONE_FILES` is set.

`int unshare_semundo(unsigned long unshare_flags, struct sem_undo_list **new_ulistp);`
This function will unshare System V semaphore once it will be implemented. The flag for this operation is `CLONE_SYSVSEM`.

`long sys_unshare(unsigned long unshare_flags);`
This is implementation of the `unshare()` syscall, which allows a process to 'unshare' part of its context that was originally shared since `clone()`. This function modifies `task_struct *current` and lock its `current->alloc_lock`.

7.3 Process Termination

Whenever a process calls `exit()` syscall, multiple things occur: The kernel must decide what to do with *orphaned* children processes, whom to inform about death, whether to something with allocated memory before releasing it etc. All references to process must be removed from all kernel structures.

`static void __unhash_process(struct task_struct *p);`

Detach PID from given task `p` and decrease number of threads in system. If this thread was thread group leader then detach `PGID` and `SID`. `tasklist_lock` should be held before call to this function.

`void release_task(struct task_struct * p);`

Release entry in `/proc`, structures for signal handling and unhash process. Stop any `ptrace`. If this thread is the last non-leader member of a thread group and the leader is zombie, then notify the group leader's parent process if it wants notification. Locks used: `tasklist_lock` and `p->proc_lock`.

`void unhash_process(struct task_struct *p);`

Only used for SMP init, unhashes proc entry and process and locks `p->proc_lock` and `tasklist_lock` (together with disabling IRQs).

`int session_of_pgrp(int pgrp);`

This function checks the process group, but falls back on the PID if no satisfactory process group is found. `tasklist_lock` is held for reading.

`static int will_become_orphaned_pgrp(int pgrp, task_t *ignored_task);`

Determine if a process group is orphaned. Orphaned process groups are not to be affected by terminal-generated stop signals. Newly orphaned process groups are to receive a `SIGHUP` and a `SIGCONT`.

`int is_orphaned_pgrp(int pgrp);`

Lock `tasklist_lock`, check whether process group will become orphaned, unlock the lock and return appropriate value (gotten from `will_become_orphaned_pgrp()`).

`static inline int has_stopped_jobs(int pgrp);`

Return non-zero value if there exists any thread in process group that is not `TASK_STOPPED` or it's being ptraced and its `exit_code` is not one of following: `SIGSTOP`, `SIGTSTP`, `SIGTTIN` or `SIGTTOU`.

`void reparent_to_init(void);`

Reparent the calling kernel thread to the init task (i.e. stop ptracing of this process, set `parent` and `real_parent` to `child_reaper`, which should represent `init_task`, set the exit signal to SIGCHLD so we signal init on exit, set nice value to `0` (only with `SCHED_NORMAL`), copy resource limit from `init_task`, switch uid to `INIT_USER` and increase reference count for this structure.

If a kernel thread is launched as a result of a system call, or if it already exits, it should generally reparent itself to init so that it is correctly cleaned up on exit. The various task state such as scheduling policy and priority may have been inherited from a user process, so we reset them to sane values here.

Note: the `reparent_to_init()` gives the caller full capabilities. `tasklist_lock` locked for writing while reparenting, corectness of the operation is checked with `security_task_reparent_to_init()`.

`void __set_special_pids(pid_t session, pid_t pgrp);`

If task process's SID is not `session` then detach from the old session and attach to `session`. If current task's process group is not `pgrp` then detach from the old one and attach to `pgrp`. This function does not acquires any locks or tampers with IRQs, see `set_special_pids()`.

`void set_special_pids(pid_t session, pid_t pgrp);`

Lock
`tasklist_lock` for writing with disabling IRQs, call `__set_special_pids()` and restore IRQs.

`int allow_signal(int signal);`

Allow given `signal`: the signal is enabled by exluding from set of blocked signals (`current->blocked`). Kernel threads handle their own signals, so prevent dropping this signal or converting it to `SIGKILL`.

This operation is protected with `current->sighand->siglock` spinlock and IRQs are disabled. Return value is zero if the `sig` is a valid signal number, `-EINVAL` is reported otherwise. Exported symbol.

`int disallow_signal(int sig);`

Disallows a signal by including its number `sig` into set of blocked signals (`current->blocked`). Spinlock `current->sighand->siglock` is held and IRQs are disable, protecting the body of the function. Return value is zero if the `sig` is a valid signal number, otherwise `-EINVAL` is returned. Exported symbol.

`void daemonize(const char *name, ...);`

Detach thread from userspace. If we were started as result of loading a module, close all of the user space pages. We don't need them, and if we didn't close them they would be locked in memory. Then block and flush all signals. Clone `fs_struct` from `init`, increase it's usage counter and close all open file

descriptors and steal from `init`. Finally reparent thread to `init()`. This puts everything required to become a kernel thread without attached user resources in one place where it belongs. Locking is done only in calls to subfunctions. Exported symbol.

`static inline void close_files(struct files_struct * files);`
Enumerate through all open files in `files` and close all of them.

`struct files_struct *get_files_struct(struct task_struct *task);`
With `task->alloc_lock` held increase `task->files->count` reference counter. Return `task->files`.[5]

`void fastcall put_files_struct(struct files_struct *files);`
Give up using `files`: atomicaly decrease and test `files->count`. If the usage counter dropped to zero, close all file descriptors, free file descriptor arrays and other memory used by this structures. Exported symbol.

`static inline void __exit_files(struct task_struct *tsk);`
Give up using `tsk->files` and set it to `NULL` while holding lock `task->alloc_lock`. This function calls `put_files_struct()` to decrement reference counter and eventually free memory.

`void exit_files(struct task_struct *task);`
Just a wrapper to call `__exit_files(task)`.

`static inline void __put_fs_struct(struct fs_struct *fs);`
Decrement reference counter of `fs` and eventually free memory structures used by it. If we're the last user and decremented it to zero, we don't need to lock anything.

`void put_fs_struct(struct fs_struct *fs);`
Wrapper to call `__put_fs_struct(fs)`.

`static inline void __exit_fs(struct task_struct *task);`
Set `task->fs` to `NULL` value, decrease reference counter to it and potentially free the memory. `task->alloc_lock` is held while `task->fs` is being modified.

`void exit_fs(struct task_struct *task);`
Wrapper to call `__exit_fs(tsk)`. Exported symbol.

[5]This function is not an exported symbol, but the following `void fastcall put_files_struct()` is.

`static inline void __exit_mm(struct task_struct *task);`

Stop using `task->mm`, set it to `NULL`. This function serializes with any possible pending coredump. `mm->mmap_sem` and `task->alloc_lock` locks are used. Reference counter to `task->mm` is decremented and memory is potentially released if we were the last user. Turn us into a lazy TLB process if we aren't already.

Warning: this function may sleep, it's implementation uses semaphores.

`void exit_mm(struct task_struct *tsk);`

Wrapper to call `__exit_mm(tsk)`. Exported symbol.

`static inline void choose_new_parent(task_t *p, task_t *reaper, task_t *child_reaper);`

Reparent task `p` to parent `child_reaper` if `p` is `reaper` of `reaper` is zombie, otherwise reparent to `reaper`.

`static inline void reparent_thread(task_t *p, task_t *father, int traced);`

Move the child from its dying parent to the new one, modify appropriate structures needed for ptracing and if we've notified the old parent about this child's death (`state == TASK_ZOMBIE`, thread group empty...) then also notify the new one. If `p` is in other process group than `father`, and it was the only connection outside, the `p`'s process group is now orphaned: if there are any stopped jobs, send `SIGHUP` and `SIGCONT` to the process group

Note: this function is called from `forget_original_parent()`, which is called from `exit_notify()` with `tasklist_lock` write-locked.

`static inline void forget_original_parent(struct task_struct *father, list_head *to_release);`

When we die, we re-parent all our children. Try to give them to another thread in our thread group, and if no such member exists, give it to the global child reaper process (`child_reaper` i.e. `init_task`).

`static void exit_notify(struct task_struct *tsk);`

Make init inherit all the child processes and check to see if any process groups have become orphaned as a result of our exiting, and if they have any stopped jobs, send them a `SIGHUP` and then a `SIGCONT`. If something other than our normal parent is ptracing us, then send it a `SIGCHLD` instead of honoring `exit_signal`, which has special meaning only to real parent. Possibly used

locks: `tasklist_lock` and `tsk->sighand->siglock`, IRQs get enabled in the end.

`asmlinkage NORET_TYPE void do_exit(long code);`

This function never returns as its purpose is to exit task. First check and warn, if we're trying to kill interrupt handler, idle task or init. If the task has I/O context, release it. Do profiling and process accounting if needed and free resources. If we're session leader then dissociate with current tty. On NUMA also free memory policy. As the last thing call `schedule()`. Exported symbol.

`NORET_TYPE void complete_and_exit(struct completion *comp, long code);`

Complete completion `comp` and exit current task using `do_exit()`. Exported symbol.

`asmlinkage long sys_exit(int error_code);`

Wrapper to call `do_exit()`.

`task_t fastcall *next_thread(const task_t *p);`

Return next thread in group. Exported symbol.

`NORET_TYPE void do_group_exit(int exit_code);`

Take down every thread in the group. This is called by fatal signals as well as by `sys_exit_group()`. This kills every thread in the thread group. Note that any externally `wait4()`-ing process will get the correct exit code, even if this thread is not the thread group leader. If `current->signal->group_exit` is non-zero, the exit code used is `current->signal->group_exit_code` instead of `exit_code` argument. Used locks: `tasklist_lock`, `current->sighand->siglock`.

`asmlinkage void sys_exit_group(int error_code);`

Wrapper to call `do_group_exit()`.

`static int eligible_child(pid_t pid, int options, task_t *p);`

This function is used in `sys_wait4()` to check whether process `pid` is eligible to wait for if given options. We process negative/zero `pid` according to description of `sys_wait4()`.

Wait for all children (clone[6] or not) if `__WALL` is set, otherwise, wait for clone children only if `__WCLONE` is set, otherwise wait for non-clone children only.

[6]Note: A clone child here is one that reports to its parent using a signal other than `SIGCHLD`.

`int wait_noreap_copyout(task_t *p, pid_t pid, uid_t uid, int why, int status, struct siginfo __user *infop, struct rusage __user *rusagep);`

Fill `struct siginfo` and `struct rusage` in user space. This is useful for `wait3()`, `wait4()` and `waitid()` syscalls. Return zero on success.

`static int wait_task_zombie(task_t *p, unsigned int __user *stat_addr, struct rusage __user *ru);`

`static int wait_task_zombie(task_t *p, int noreap,struct siginfo __user *infop, int __user *stat_addr, struct rusage __user *ru);`

This function handles `sys_wait4()` work for one task in state `TASK_ZOMBIE`. We hold `read_lock(&tasklist_lock)` on entry. If we return zero, we still hold the lock and this task is uninterestingi and we have filled user space structures.

If we return nonzero, we have released the lock and the system call should return. We try to move the task's state to `TASK_DEAD`, if we fail this task is not interesting to us. Then check whether we're not in a race with ptraced thread dying on another processor, if it is so then jump out. In this phase we are sure that this task is interesting, and no other thread can reap it because we set its state to `TASK_DEAD`. We can unlock `tasklist_lock` now and check the state of resources.

If `p->real_parent != p->parent` then write lock `tasklist_lock`, unlink ptrace for task `p` and set `p`'s state to `TASK_ZOMBIE`.

If this is not detached task, notify the parent. If it's still not detached after that, don't release it now. Finally unlock the `tasklist_lock`. And done the releasing potentially we needed after notifying parent before unlocking.

`static int wait_task_stopped(task_t *p, int delayed_group_leader, unsigned int __user *stat_addr, struct rusage __user *ru);`

Handle `sys_wait4()` work for one task in state `TASK_STOPPED`. We hold `read_lock(&tasklist_lock)` on entry.

If we return zero, we still hold the lock and this task is uninteresting and the user space structures have been filled with data.

If we return nonzero, we have released the lock and the system call should return.

If group stop is in progress and `p` is the group leader return zero. Now we are pretty sure this task is interesting. Make sure it doesn't get reaped out from

under us while we give up the lock and then examine it below, so increase its usage using `get_task_struct(p)`. We don't want to keep holding onto the `tasklist_lock` while we try to get resource usage and acquire data from user space as this could possibly cause page faults. Relock `task` from read to write lock and try to update `p`'s state with `TASK_STOPPED`.

If we failed the task resumed and then died, so let the next iteration catch it in `TASK_ZOMBIE`[7].

We check during next step whether `exit_code` is zero, if it's sok then other function got it first or it resumed or it resumed and then died. So unlock `tasklist_lock`, decrease `p`'s usage counter and read-lock `tasklist_lock` and return with return value of zero. As the next step we move to end of parent's list to avoid starvation and unlock `tasklist_lock`.

`static int wait_task_continued(task_t *p, int noreap, struct siginfo __user *infop, int __user *stat_addr, struct rusage __user *ru);`
This function handles `do_wait()` work for one task in a live, non-stopped state. This function must be called with `read_lock(&tasklist_lock)` acquired. Zero return value indicates that this task is not interesting and lock is still acquired.

Non-zero return value means that the lock has been released and the syscall should return to user space (this non-zero return value is the pid).

`int my_ptrace_child(struct task_struct *p);`
Return true if task `p` is ptraced by someone and the structures are currently being modified.

`long do_wait(pid_t pid, int options, struct siginfo __user *infop, int __user *stat_addr, struct rusage __user *ru);`
This suspends execution of the current process until a child as specified by the `pid` argument has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function.

Implementation of this function declares its own waitqueue, which is added to `current->wait_chldexit` and change current's task state to `TASK_INTERRUPTIBLE`.

For each child of current task check if the child is eligible to wait for, if it is not so, go on with next child. Based on the state of this child decide to

[7] `exit_code` might already be zero here if it resumed and did `_exit(0)`. The task itself is dead and it won't touch `exit_code` again, other processors in this function are locked out.

use either `wait_task_stopped()` for `TASK_STOPPED` or `wait_task_zombie()` for `TASK_ZOMBIE`.

If the waiting in one of these two functions finished successfully (i.e. return value is non-zero), the `tasklist_lock` is unlocked and we can remove our waitqueue from the `current->wait_chldexit` and return the PID of the child that exited as return value. If we didn't find any eligible child, we try to look for one in list of ptraced children. If we suceeded then unlock `tasklist_lock` and based on `options` decide whether we found the child we were looking for or whether we need to restart the whole search, in which case we call `schedule()` before restarting the search.

Return value is the PID of the process we waited for or `-ECHILD` to indicate that there is no child process we can wait for. `-ERESTARTSYS` means that this syscall woulb block and `WNOHANG` option has been set.

`asmlinkage long sys_waitid(int which, pid_t pid, struct siginfo __user *infop, int options, struct rusage __user *ru);`
Implementation of `waitid()` syscall. `do_wait()` is called after a batch of checks. See `man 2 waitid`.

`asmlinkage long sys_wait4(pid_t pid, int __user *stat_addr, int options, struct rusage __user *ru);`
Implementation of `waitid()` syscall. `do_wait()` is called after a batch of checks. See `man 2 wait4`.

`asmlinkage long sys_waitpid(pid_t pid, unsigned __user *stat_addr, int options);`
Wrapper to call `sys_waitpid4()`. `sys_waitpid()` remains for compatibility. `waitpid()` should be implemented by calling `sys_wait4()` from `libc`. See `man 4 wait`.

7.4 Waitqueues

Sometimes a task gets into state that forbids later execution until some event happens. Implementation using busy-loops is not effective[8], so the task is inserted into structure called `wait_queue_t` and delayed for later execution. Developer can

[8]It's true that busyloops are used in kernel, e.g see spinlock implementation. However spinlocks are held only for a short time, delaying task using waitqueue can cause bigger overhead: the need for context switching, invalidating cpu cache etc.

decide whether to use one of the already declared wait queues or define his own using macro `DECLARE_WAIT_QUEUE(name)`. This macro also declares variable of name `name`, in case a variable was declared before assign to that variable macro `__WAITQUEUE_INITIALIZER(name, tsk)`.

The items held in wait queue are represented by structure `wait_queue_head_t`, which can be declared using macro `DECLARE_ WAIT_QUEUE_HEAD(name)` or initialized using `__WAIT_QUEUE_HEAD_INITIALIZER(name)`. Another option is to initialize `wait_queue_head_t` using function `static inline void init_waitqueue_head(wait_queue_head_t *q)`, functionality is the same. `wait_queue_t` can be initialized with `static inline void init_waitqueue_entry(wait_queue_t *q, struct task_struct *p)`, this will use the default wake function – for alternative wake function use `static inline void init_waitqueue_func_entry(wait_queue_t *q, wait_queue_func_t func)`.

`static inline int waitqueue_active(wait_queue_head_t *q);`

Check whether task list is empty and if it is so return true, otherwise return false.

`void fastcall add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);`

Clear exclusive flag (`WQ_FLAG_EXCLUSIVE`) from `wait` and add wait to the wait queue `q`. Spinlock `q->lock` is held and IRQs disabled while adding to q. Exported symbol.

`void fastcall add_wait_queue_exclusive(wait_queue_head_t *q, wait_queue_t *wait);`

Set exclusive flag (`WQ_FLAG_EXCLUSIVE`) to `wait` and add `wait` to the wait queue `q`. The exclusive flag means that the process will be woken alone and no other process will be woken up. Spinlock `q->lock` is held and IRQs disabled while adding `q`. Exported symbol.

`void fastcall remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);`

Remove `wait` from wait queue `q`, `q`'s lock is held and IRQs disabled while removing. Exported symbol.

`static inline void __add_wait_queue(wait_queue_head_t *head, wait_queue_t *new);`

Add `new` to the wait queue head `head`, to compare with `add_wait_queue`: this function does not manipulate flags and does not make us of any spinlock.

`static inline void __add_wait_queue_tail(wait_queue_head_t *head, wait_queue_t *new);`

Add `new` to the tail of `head`, no locks are used or flags changed. Used for wake-one threads.

`static inline void __remove_wait_queue(wait_queue_head_t *head, wait_queue_t *old);`

Remove `old` from the wait queue `head`.

`int default_wake_function(wait_queue_t *curr, unsigned mode, int sync, void *key);`

Try to wake up current task from `curr` wait queue. Exported symbol.

`static void __wake_up_common(wait_queue_head_t *q, unsigned int mode, int nr_exclusive, int sync, void *key);`

The core wakeup function. Non-exclusive wakeups (`nr_exclusive == 0`) just wake everything up. If it's an exclusive wakeup (`nr exclusive == small + ve number`) then we wake all the non-exclusive tasks and one exclusive task. There are circumstances under which we can try to wake a task which has already started to run but is not in state `TASK_RUNNING`. `try_to_wake_up()` returns zero in this rare case, and we handle it by continuing to scan the queue.

`void fastcall __wake_up(wait_queue_head_t *q, unsigned int mode, int nr_exclusive, void *key);`

Wake up threads blocked on a waitqueue `q`. Lock `q->lock` is locked and irqs distabled during execution. Exported symbol.

`void fastcall __wake_up_locked(wait_queue_head_t *q, unsigned int mode);`

Same as `__wake_up()` but called with the spinlock in `wait_queue_head_t` held.

`void fastcall __wake_up_sync(wait_queue_head_t *q, unsigned int mode, int nr_exclusive);`

Wake up threads blocked on a waitqueue. The sync wakeup differs that the waker knows that it will schedule away soon, so while the target thread will be woken up, it will not be migrated to another CPU - ie. the two threads are synchronized with each other. This can prevent needless bouncing between

CPUs. On UP it can prevent extra preemption. Lock `q->lock` is locked during execution. Exported symbol.

Some of the following macros are using previous function for waking up:

`wake_up(x)`
Wake up one exclusive `TASK_UNINTERRUPTIBLE` or `TASK_INTERRUPTIBLE` task.

`wake_up_nr(x, nr)`
Wake up `nr` number of exclusive `TASK_UNINTERRUPTIBLE` or `TASK_INTERRUPTIBLE` tasks.

`wake_up_all(x)`
Wake up all `TASK_UNINTERRUPTIBLE` or `TASK_INTERRUPTIBLE` tasks.

`wake_up_all_sync(x)`
Synchronously wake up all `TASK_UNINTERRUPTIBLE` or `TASK_INTERRUPTIBLE` tasks.

`wake_up_interruptible(x)`
Wake up one exclusive `TASK_INTERRUPTIBLE` task.

`wake_up_interruptible_nr(x, nr)`
Wake up nr number of exclusive `TASK_INTERRUPTIBLE` tasks.

`wake_up_interruptible_all(x)`
Wake up all `TASK_INTERRUPTIBLE` tasks.

`wake_up_locked(x)`
Wake up one `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` task with spin-lock in `x->lock` held and IRQs disabled.

`wake_up_interruptible_sync(x)`
Synchronously wake up one `TASK_INTERRUPTIBLE` task.

These are the new interfaces to sleep waiting for an event, the older ones are documented later since they're still in use, but developers should use these new ones.

`DEFINE_WAIT(name)`
This macro declares a wait queue of given `name`, initializes it and sets its wake function to `autoremove_wake_function()`.

`init_wait(wait);`

This macro initializes `wait_queue_t` and sets its wake function to `autoremove_wake_function()`.

`__wait_event(wq, condition)`

Declare a wait with autoremove function (i.e. remove task from queue if task was successfully finished) and add `wq` to wait queue marked as `TASK_UNINTERRUPTIBLE` and wait for finishing `wq` while `condition` is not accomplished.

`wait_event(wq, condition)`

First check the `condition`, if it is true then jump out else continue with `__wait_event(wq, condition)`.

`__wait_event_interruptible(wq, condition, ret)`

Declare a wait with autoremove function (i.e. remove task from queue if task was successfully finished) and add `wq` to wait queue marked as `TASK_UNINTERRUPTIBLE` and wait for finishing `wq` while `condition` is not accomplished. Right after checking the `condition` check for any pending signals. If there are none try the cycle again otherwise return with `-ERESTARTSYS`. Note that if we have successfuly fulfilled the `condition` we don't change the `ret` value, so it should not be `-ERESTARTSYS` in the beginning.

`wait_event_interruptible(wq, condition)`

Declare int `__ret = 0;` and check `condition`. If the `condition` is no true, continue with `__wait_event_interruptible(wq, condition, __ret)`, i.e. this misses the possibility of bug `-ERESTARTSYS` in the beginning mentioned in previous macro.

`__wait_event_interruptible_timeout(wq, condition, ret)`

Declare a wait with autoremove function (i.e. remove task from queue if task was successfully finished) and add `wq` to wait queue marked as `TASK_UNINTERRUPTIBLE` and wait for finishing `wq` while `condition` is not accomplished. Right after checking the `condition` check for any pending signals. If there are none, check the timeout (if we returned after timeout jump out) and try the cycle again otherwise return with `-ERESTARTSYS`. Note that if we have successfuly fulfilled the `condition` we set the ret to timeout value.

`wait_event_interruptible_timeout(wq, condition, timeout)`

Declare `int __ret`

and initialize it to timeout and check `condition`. If the `condition` is not true, continue with `__wait_event_interruptible_timeout(wq, condition, __ret)`, i.e. this misses the possibility of bug `-ERESTARTSYS` in the beginning mentioned in previous macro (the timeout should be a positive value).

`__wait_event_interruptible_exclusive(wq, condition, ret)`
: Declare an exclusive wait with autoremove function (i.e. remove task from queue if task was successfully finished) and add `wq` to wait queue marked as `TASK_UNINTERRUPTIBLE` and wait for finishing `wq` while `condition` is not accomplished. Right after checking the `conditio` check for any pending signals. If there are none try the cycle again otherwise return with `-ERESTARTSYS`. Note that if we have successfuly fulfilled the condition we don't change the `ret` value, so it should not be `-ERESTARTSYS` in the beginning.

`wait_event_interruptible_exclusive(wq, condition)`
: Declare `int __ret = 0;` and check `condition`. If the `condition` is not true, continue with `__wait_event_interruptible_exclusive(wq, condition, ret)`, i.e. this misses the possibility of bug `-ERESTARTSYS` in the beginning mentioned in previous macro.

```
static inline void __add_wait_queue_exclusive_locked(wait_queue_head_t
    *q, wait_queue_t *wait);
```
: Mark `wait` as exclusive wake and add wait to the tail of `q`, no locks are used, so must be called with the spinlock `q->lock` held.

```
static inline void remove_wait_queue_locked(wait_queue_head_t *q,
    wait_queue_t *wait);
```
: Remove `wait` for wait queue `q`, must be called with the spinlock `q->lock` held.

These are the old interfaces to sleep waiting for an event. I enlisted them just for simplifying analysis of kernel 2.6. They can cause race condition. DO NOT use them, use the `wait_event*` interfaces above. It's rumoured that this interfaces are planned to be removed during 2.7:

```
void fastcall __sched interruptible_sleep_on( wait_queue_head_t *q);
```
: Create new waitqueue, add current task, chain with `q`, call `schedule()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

```
long fastcall __sched interruptible_sleep_on_timeout(
    wait_queue_head_t *q, long timeout);
```

Create new waitqueue, add current task, mark it as `TASK_INTERRUPTIBLE`, chain with `q`, call `schedule_timeout()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

`void fastcall __sched sleep_on(wait_queue_head_t *q);`
Create new wait queue, add current task, mark it as `TASK_UNINTERRUPTIBLE`, chain with `q`, call `schedule()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

`long fastcall __sched sleep_on_timeout(wait_queue_head_t *q, long timeout);`
Create new wait queue, add current task, mark it as `TASK_UNINTERRUPTIBLE`, chain with `q`, call `schedule_timeout()` and remove this new runqueue from `q`. `q`'s lock is held while manupulating `q`. Exported symbol.

Wait queues which are removed from the waitqueue head at wakeup time:

`void fastcall prepare_to_wait(wait_queue_head_t *q, wait_queue t *wait, int state);`
Clear exclusive flag and if list `wait->task_list` is empty add it to `q` and set current task's state to `state`, everything while holding `q` lock and IRQs disabled. Exported symbol.

`void fastcall prepare_to_wait_exclusive(wait_queue_head_t *q, wait_queue_t *wait, int state);`
Set exclusive flag and if list `wait->task_list` is empty add it to `q` and set current task's state to `state`, everything while holding `q->lock` and IRQs disabled. Exported symbol.

`void fastcall finish_wait(wait_queue_head_t *q, wait_queue_t *wait);`
Set current task's state to `TASK_RUNNING`, check if list `wait->task_list` is empty and if it is not: lock wait queue spinlock and disable IRQs, delete head, reinitialize `wait` as new, unlock wait queue spinlock and restore previous IRQ state. Note: Checking for emptiness is done without any locking using `list_empty_careful()`. Exported symbol.

`int autoremove_wake_function(wait_queue_t *wait, unsigned mode, int sync, void *key);`
Try to wake up `wait`, if successful then delete head of `wait` and reinitialize it as new. On success return true, otherwise false. Exported symbol.

7.5 Workqueues and Kernel Threads

Work queues are generic mechanism for defining kernel helper threads for running arbitrary tasks in process context. Work queues are based on wait queues, but they are not as frequently used as wait queues.

The most basic structure `struct work_struct` represents work to be done with following attributes: `pending` is non-zero if work is currently pending, `entry`, pointer to function `func` that will be called with parameter `data`, pointer to `cpu_workqueue_struct` this work belongs to and `timer` to be used with delayed work.

Another basic structure is `cpu_workqueue_struct`. The contents is protected by its member spinlock `lock`. The `worklist` keeps list of work that need to be done. `run_depth` is used in `run_workqueue()` to check whether recursion is not too deep[9]

Workqueues are tied together in structure `workqueue_struct`, which contains per-CPU `cpu_workqueue_struct cpu_wq[NR_CPUS]`, `name` of the workqueue and `list` of all per-CPU workqueues in the system.[10]

Reasonable API to these functions and macros is provided by <`linux/workqueue.h`> and `kthread.h`.

All the per-CPU workqueues on the system are protected by single spinlock `workqueue_lock` and they're held in `workqueues` list. This list is modified on the fly as the CPUs come and go.

There
are three functions defined in <`linux/workqueue.h`>: `__WORK_INITIALIZER(n, f, d)`, which initializes `struct work_struct n` and sets callback function to `f` with argument `d`. The next is `DECLARE_WORK(n, f, d)`, which does the same, but also declares the variable `n`. And the last one is `PREPARE_WORK(_work, _func, _data)`, which only sets callback function `_func` and its argument `_data`.

`static inline int is_single_threaded(struct workqueue_struct *wq);`
: Returns true if `workqueue_struct` is singlethreaded.

`static void __queue_work(struct cpu_workqueue_struct *cwq, struct work_struct *work);`
: Add `work` to the tail of queue `cwq` and save `cwq` to `work->wq_data` so that we can easily find which `cpu_workqueue_struct` we belong to. Increase counter of 'works' in queue (the `insert_sequence`), and try to wake up some work from

[9] The stack available in kernel is limited, 4 or 8 kB on i386

[10] Note that this list is empty if the workqueues are single-threaded.

queue. Body of this function is executed with IRQs disabled and `cwq->lock` held. Preemption should be disabled before calling this function.

`int fastcall queue_work(struct workqueue_struct *wq, struct work_struct *work);`

Queue `work` on a workqueue `wq`. Return non-zero if it was successfully added. The work is queued to the CPU it was submitted by (except when `wq` is singlethreaded), but there is no guarantee that it will be process by that CPU. Preemption is disabled during the execution, the work is added if there is no work pending on `wq`. If `wq` is singlethreaded, add work to the first CPU, otherwise to current one. Exported symbol.

`static void delayed_work_timer_fn(unsigned long __data);`

Queue work `(struct work_struct *)__data` to workqueue belonging to current CPU (`(struct work_struct *) __data->wq_data->cpu_wq[smp_processor_id()]`). If the workqueue is singlethreaded, add to the first CPU as usual. This is timer callback and shouldn't be called directly.

`int fastcall queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay);`

If there is no work/timer pending, store in `work->wq_data` the workqueue we want to insert the work into. Then set the timer to expire at `jiffies+delay` time, set the callback to be `delayed_work_timer_fn` and add timer. This way the `work` will get enqueued in `wq` as soon as at least `delay` jiffies pass. Non-zero return value means success. Exported symbol.

`static inline void run_workqueue(struct cpu_workqueue_struct *cwq);`

Increase `run_depth` in `cpu_workqueue_struct` and check whether the recursion is not too deep (i.e. more than 3, if it is so bug user and dump stack). Otherwise cycle through `cwq->worklist`, remove `work_struct`s one by one, clear `pending` bit and call `work_struct->func` on each. After completing one increase the `remove_sequence` counter and wake up `work_done`. Then repeat if there is more work to be done. Spinlock `cpu_workqueue_lock` is held together with IRQs disabled.

`static int worker_thread(void *__cwq);`

Set `PF_NOFREEZE` flag to current process and renice it to `-10`. Then block and flush all signals and set mark state of current thread as `TASK_INTERRUPTIBLE`.

While `kthread_should_stop()`[11] is false, we stay inside the cycle and try to perform work enqueued in current CPU's runqueue. If the list is empty, we call `schedule()`. While we're doing some work, current thread is marked as `TASK_RUNNING`. In the end (i.e. `kthread_should_stop()` is true) mark current process as `TASK_RUNNING`. Return value is always zero.

`static void flush_cpu_workqueue(struct cpu_workqueue_struct *cwq);`

In the first place we check whether the keventd is not trying to flush its own queue. If it is so, simply call `run_workqueue(cwq)` to avoid deadlock. In the other case lock `cwq->lock` and disable irqs. Prepare `cwq->work_done` to wait and (after unlocking spinlock and restoring IRQs) call `schedule()` so that waiting task could be waken up, after return from `schedule()` again lock the spinlock and disable IRQs. Repeat this `cwq->insert_sequence - cwq->remove_sequence` times to do all work enqueued here. When done reinitialize the list (`cwq->work_done`) to be empty, unlock spinlock and restore IRQs.

`void fastcall flush_workqueue(struct workqueue_struct *wq);`

Ensure that any scheduled work has run to completion: if workqueue is singlethreaded then continue with `flush_cpu_workqueue()` for CPU 0, otherwise lock cpu hotplug and for each online CPU call `flush_cpu_workqueue()`. When done unlock CPU hotplug and return. This function forces execution of the workqueue and blocks until completion (typically used in driver shutdown handlers). It will sample each workqueue's current `insert_sequence` number and will sleep until the head sequence is greater than or equal to that. This means that we sleep until all works which were queued on entry have been handled, but we are not livelocked by new incoming ones. Exported symbols.

`static struct task_struct *create_workqueue_thread(struct workqueue_struct *wq, int cpu);`

Create kthread to handle `worker_thread` for given `cpu`. Also set `insert_sequence` and `remove_sequence` to zeros and init `more_work` and `work_done`. Return newly created kthread's `task_struct` on success (otherwise `NULL`).

`struct workqueue_struct *__create_workqueue(const char *name, int singlethread);`

Create workqueue: allocate memory for `workqueue_struct`, clear it with zeros

[11] see `kthread_stop()` for details

and set its name (set pointer, no string copying, don't free the string memory too early). Lock the cpu hotplug, we don't need race condition. In case we want single-threaded processing of workqueues we just init list `wq->list` and call `create_workqueue_thread()` and wake up newly created kthread. In case of multi-thread processing of workqueues we add `workqueues` to the `list` and for each online CPU create one kthread, bind it to CPU and wake it up. When we're done with all CPUs, we can unlock cpu hotplug. Return value on success is the `workqueue_struct` we just created else `NULL` in the other case. Exported symbol. Note: <linux/workqueue.h> defines macro `create_workqueue(name)` as `__create_workqueue((name), 0)` and macro `create_singlethread_workqueue(name)` as `__create_workqueue((name), 1)`.

`static void cleanup_workqueue_thread(struct workqueue_struct *wq, int cpu);`

Stop kthread belonging to workqueue of `cpu`. Locks that workqueue's spinlock and disables IRQs while retrieving pointer to `task_struct` (and setting it to `NULL`)

`void destroy_workqueue(struct workqueue_struct *wq);`

Flush workqueues, lock cpu hotplug and clean workqueue thread for each cpu up. If using multi-threaded processing of workqueues, hold lock `workqueue_lock` while deleting `wq->list`[12]. Unlock cpu hotplug and free memory used by `wq`. Exported symbol.

`static struct workqueue_struct *keventd_wq;`

Keventd's workqueue.

`int fastcall schedule_work(struct work_struct *work);`

Queue `work` to `keventd_wq`. Exported symbol.

`int fastcall schedule_delayed_work(struct work_struct *work, unsigned long delay);`

Queue `work` to `keventd_wq` with `delay` jiffies delay. Exported symbol.

`int fastcall schedule_delayed_work_on(int cpu, struct work_struct *work, unsigned long delay);`

Queue a `work` on given `cpu` with given `delay` (in jiffies).

[12]Remember that for single-threaded processing is this list always empty.

`int schedule_on_each_cpu(void (*func)(void *info), void *info);`
Schedule given function on each cpu.

`void flush_scheduled_work(void);`
Flush `keventd_wq` workqueue. Exported symbol.

`void cancel_rearming_delayed_workqueue(struct workqueue_struct *wq, struct work_struct *work);`
Kill `work` and if it resisted then flush the waitqueue.

`void cancel_rearming_delayed_workqueue(struct work_struct *work);`
Kill `work` or flush keventd workqueue.

`int keventd_up(void);`
Return true if `keventd_wq` is up.

`int current_is_keventd(void);`
Return non-zero if current process on this CPU is keventd.

`static void take_over_work(struct workqueue_struct *wq, unsigned int cpu);`
Take the work from `cpu` that went down in hotplug. Lock of workqueue assigned to incriminated is held and IRQs are disabled protecting the body of function. Only with `CONFIG_HOTPLUG_CPU`

`static int __devinit workqueue_cpu_callback(struct notifier_block *nfb, unsigned long action, void *hcpu);`
This is callback for handling CPU hotplug events. Prepare workqueue threads for CPUs that are preparing to go up, wake thier worker threads when the CPU goes online or when going up was canceled bind the kthread to current CPU and clean its workqueue up. When a CPU dies, take its work to current CPU. Only with `CONFIG_HOTPLUG_CPU`

`void init_workqueues(void);`
Register callback for cpu hotplug events and create keventd's waitqueue (`keventd_wq`) with name `events`.

`static inline int cancel_delayed_work(struct work_struct *work);`
Kill off a pending `schedule_delayed_work()`. Note that the work callback function may still be running on return from `cancel_delayed_work()`. Run `flush_scheduled_work()` to wait on it. The function returns whether it has deactivated a pending timer or not.

`struct task_struct *kthread_create(int (*threadfn)(void *data), void *data, const char namefmt[], ...);`

This helper function creates and names a kernel thread. The thread will be stopped: use `wake_up_process()` to start it. See also `kthread_run()` and `kthread_create_on_cpu()`. When woken, the thread will run `threadfn()` with `data` as its argument. `threadfn` can either call `do_exit()` directly if it is a standalone thread for which noone will call `kthread_stop()`, or return when `kthread_should_stop()` is true (which means `kthread_stop()` has been called). The return value should be zero or a negative error number: it will be passed to `kthread_stop()`. Returns a `task_struct` or `ERR_PTR(-ENOMEM)`. The `namefmt` argument contains a printf-style name for the thread. Exported symbol.

`define kthread_run(threadfn, data, namefmt, ...)`

This macro is wrapper for `kthread_create()` followed by `wake_up_process()`. Returns the kthread's `task_struct`, or `ERR_PTR(-ENOMEM)`. Arguments are identical with those of `kthread_create()`.

`void kthread_bind(struct task_struct *k, unsigned int cpu);`

Bind a just-created kthread `k` to given `cpu`. This function is equivalent to `set_cpus_allowed()`, except that `cpu` doesn't need to be online, and the kthread `k` must be stopped (ie. just returned from `kthread_create()`. Exported symbol.

`int kthread_stop(struct task_struct *k);`

Stop a kthread `k` created by `kthread_create()`: set `kthread_should_stop()` for `k` to return true, wakes it, and waits for it to exit. The `threadfn()` must not call `do_exit()` itself if you use this function. This can also be called after `kthread_create()` instead of calling `wake_up_process()`: the thread will exit without calling `threadfn()`. The return value is the result of `threadfn()`, or `-EINTR` if `wake_up_process()` was never called.[13] Exported symbol.

`int kthread_stop_sem(struct task_struct *k, struct semaphore *s);`

Stop a thread created by `kthread_create()`. The `up` operation on semaphore `s` is used to wake up `k`.

`int kthread_should_stop(void);`

When someone calls `kthread_stop()` on your kthread, it will be woken and

[13]Thread stopping is done by setting `struct kthread_stop_info kthread_stop_info`, mutex `kthread_stop_lock` serializes multiple `kthread_stop()` calls.

this will return true. Your kthread should then return, and your return value will be passed through to `kthread_stop()`. Exported symbol.

`static void kthread_exit_files(void);`
Gives up `files_struct` and `fs_struct` and adopt ones from `init_task`.

`static int kthread(void *_create);`
Exit files, block and flush all signals and mark as to be allowed to run on any CPU by default. Then change the state to `TASK_INTERRUPTIBLE` and wait for either stop or wake-up. After rescheduling check if someone stopped us, or if the thread has exited on its own without `kthread_stop()` and deal with this situation (the return value is zero in this case).

`static void keventd_create_kthread(void *_create);`
Create new kernel thread via arch-specific call `kernel_thread()` and following completion, uses `kthread()`.

7.6 PIDs

Processes are identified by PIDs, which are unique numbers. No two processes at any moment can share PID. On the other side since simple data types are always able to hold exactly one value from a finite set, PIDs need to be reused. If we can guarantee that the set is greater than the biggest possible number of running processes, we can use any free pid at any time.

There are currently four types of PIDs (from <`linux/pid.h`>):

`PIDTYPE_TGID`
is the process PID as defined in POSIX.1: it's the ID of whole group of threads inside one process. Threads inside one process share this ID.

`PIDTYPE_PID`
is the same for standalone processes and for leader threads, it's different for threads, i.e. each of six `xmms`[14] threads has it's own PID.

`PIDTYPE_PGID`
is the process group PID. Process groups are used for distribution of signals, e.g. processes *joined* by pipe on a command line will belong to one process group.

[14]Xmms is media player application

`PIDTYPE_SID`
: is the session ID. When a tty is lost all processes belonging to affected session receive `SIGHUP` followed by `SIGCONT`.

Following functions are defined in `[kernel/pid.c]`:

`fastcall void free_pidmap(int pid);`
: Mark bit as free in pidmap table.

`int alloc_pidmap(void);`
: Try to allocate PID from pidmap, on success return new PID value, on failure return -1.

`fastcall struct pid *find_pid(enum pid_type type, int nr);`
: Find `struct pid` for given `type` and number `nr` of pid.

`int fastcall attach_pid(task_t *task, enum pid_type type, int nr);`
: Attach `task` to PID `nr` of given `type`. Always return 0.

`static inline int __detach_pid(task_t *task, enum pid_type type);`
: Detach `task` from PID it belongs to. Return number of PID.

`void fastcall detach_pid(task_t *task, enum pid_type type);`
: Detach PID via `__detach_pid()` and check if there is any type of PID asociated with the number of PID from which we released the `task`. If there isn't any then free pid via `free_pidmap()`.

`task_t *find_task_by_pid(int nr);`
: Return task that belongs to given PID `nr`. Exported symbol.

`void switch_exec_pids(task_t *leader, task_t *thread);`
: This function switches the PIDs if a non-leader thread calls `sys_execve()`. This must be done without releasing the PID, which a `detach_pid()` would eventually do.

`void __init pidhash_init(void);`
: Initialize `pid_hash`, `pidhash_shift` and `pidhash_size`. The pid hash table is scaled according to the amount of memory in the machine. From a minimum of 16 slots up to 4096 slots at one gigabyte or more.

`void __init pidmap_init(void);`
: Allocate memory for `pidmap_array->page` and set PID 0 as used (so that the first PID used will be 1) and accordingly decrease number of free PIDs. Finally

associate current task with PID 0 (remember that this is done at boot and the only thread is the one doing all initializations)

The API (<linux/pid.h> contains prototypes for attach_pid(), detach_pid(), find_pid(), alloc_pidmap(), free_pidmap() and switch_exec_pids(). Moreover later mentioned macros are declared.

pid_task(pids, type) return pointer to struct task_struct if elem is pids[type]->pid_list inside task_struct.

Enumeration through all PIDs of some type can be done using these two macros: do_each_task_pid(who, type, task) and while_each_task_pid(who, type, task). Arguments to these macros should be the same:

```
do_each_task_pid(tty->session, PIDTYPE_SID, p) {
    // do something with whole PID session
} while_each_task_pid(tty->session, PIDTYP_SID, p);
```

This code was inspired by [drivers/char/tty_io.c], because TTY layer is the biggest user of PIDs.

7.7 Process Accounting for Linux

Process accounting is a way of keeping (recording) accounting information for processes regarding (in version 2) user/group id, real user/group id, control terminal, process creation time, user time, system time, elapsed time, average memory usage, characters transfered, blocks read or written, minor and major pagefaults and number of swaps, exitcode, command name, elapsed time and various flags. Process accounting is somewhat different from BSD.

Possible flags are AFORK (process forked but didn't exec), ASU (superuser privileges), ACORE (core dumped), AXSIG (killed by signal) and big/little endianness.

Changes between versions consist mainly of new binary format and parent process id was added between version 2 and 3. Later the developers added up to 6 possible logging formats, but without any additions to accounted items.

Process accounting is controlled by acct() syscall and by amount of free space. Initially it is suspended whenever there is less then 2% of free space and it is resumed again when the free space crosses 4%. The check is performed every 30 seconds (these values are controllable by acct_parm).

```
struct acctglbs {
spinlock_t lock;
```

```
volatile int active;
volatile int needcheck;
struct file *file;
struct timer_list timer;
};
```

Accounting globals are protected by spinlock `acct_glbs->lock`, accounting is active when `acct->active` holds nonzero value. The accoung information is being written to `acct->file`.

`static void acct_timeout(unsigned unused);`
: This function is being called whenever timer says to check free space. It's internal function consists of setting `acct_globals.needcheck` to 1, which is being checked in following function:

`static int check_free_space(struct file *file);`
: This funcion first checks if free space needs to be checked (see `acct_timeout`) and if it is so it performs accordingly. Finally it deletes old timer and registers new one using timeout from `ACCT_TIMEOUT` (actually this macro refers to `acct_param[2]`, which can be changed in runtime).

`void acct_file_reopen(struct file *file);`
: Close the old accounting file, if any open, and open a new one (if file is non-NULL). If there is an old file open, it deletes timer and resets the rest of `acct_glbs` to initial values. If there is a new file open, it sets it as accounting file in `acct_globals`, sets the accounting to active state and starts new timer. In the end it does one last accounting and closes old file. `acct_globals.lock` must be held on entry and exit, altough it is unlocked before last accounting to old file, closing old file and afterwards it is locked again.

`int acct_on(char *filename);`
: Open a file `filename` for appending and check whether it's ok to write to this file. Finally call `acct_file_reopen()` to close old file and use new one. This function returns either 0 if no error occured or `-EACCESS` if the permissions do not allow accounting to write to this file or `-EIO` in case of any other error.

`asmlinkage long sys_acct(const char __user *name);`
: This system call starts accounting (or stops in case of `NULL` parameter). If the name argument is not `NULL` the file is opened in write and append

mode. Then the `security_acct(file)` is used to check permission to account. If we don't have permission to do accounting, the file is closed. Finally`acct_file_reopen(file)` is being called with `acct_globals.lock` is being held.

`void auto_close_mnt(struct vfsmount *m);`

Turn off accounting if it's done on `m` file system.

`void acct_auto_close(struct super_block *sb);`

If the accounting is turned on for a file in the filesystem pointed to by sb, turn accounting off. The `acct_globals.lock` lock is being held during call to `acct_file_reopen((struct file *)NULL)`.

`static comp_t encode_comp_t(unsigned long value);`

Encodes an `unsigned long`to `comp_t`.

`static comp2_t encode_comp2_t(u64 value);`

If `ACCT_VERSION` is defined as 1 or 2 then this function encodes unsigned 64-bit integer into `comp2_t`.

`static u32 encode_float(u64 value);`

If `ACCT_VERSION` is defined as 3 then this function encodes unsigned 64-bit integer into 32-bit IEEE float.

`static void do_acct_process(long exitcode, struct file *file);`

This function does the real work. Caller holds the reference to `file`. It starts with checking, whether there's enough free space to continue the accounting. If it is so, fill the structure with the needed values as recorded by the various kernel functions. Then the accounting structure is written to file; resource limit limiting filesize (`FSIZE`) is disabled during this operation.

`void acct_process(long exitcode);`

Handle process accounting for an exiting file. This function is just a wrapper for `do_acct_process()` with some checks done first.

`void acct_update_integrals(struct task_struct *tsk);`

Update `tsk->mm` integral fields: RSS and VM usage.

`void acct_clear_integrals(struct task_struct *tsk);`

Clear the `tsk->mm` integral fields.

7.8 Thread API

To enumerate through the all threads use following construction:

```
do_each_thread(g, p) {
    // do something with 'p'
} while_each_thread(g, p);
```

To exit from forementioned loop use `goto` and not `break`, because these macros work as two loops: the outer one iterate through the list of tasks and the inner one iterates through the threads for each task.

Thread flags (based on i386 architecture):

`TIF_SYSCALL_TRACE`
: Syscall trace is active.

`TIF_NOTIFY_RESUME`
: Resumption notification is requested.

`TIF_SIGPENDING`
: A signal is pending.

`TIF_NEED_RESCHED`
: Rescheduling is necessary.

`TIF_SINGLESTEP`
: Restore singlestep on return to user mode. This is useful for ptrace.

`TIF_IRET`
: Return with `iret` instruction.

`TIF_SYSCALL_EMU`
: Syscall emulation is active. Useful for ptrace.

`TIF_SYSCALL_AUDIT`
: Syscall auditing is active.

`TIF_SECCOMP`
: Secure computing is active, i.e. thread is allowed to do only some syscalls: read, write, close...[15] `TIF_RESTORE_SIGMASK` Restore the signal mask in

[15]This feature will allow to create `inetd`-like daemon that would accept connectionis, `fork()` and then limit abilities to those few allowed in secure computing mode. Then the server itself will be executed to serve the request.

`do_signal()`. `TIF_POLLING_NRFLAG` True if `poll_idle()` is polling on `TIF_NEED_RESCHED`. This is power consuming and should be used carefully. `TIF_MEMDIE` This thread is freeing some memory so don't kill other thread in OOM-killer.

Following methods use these flags:

`void set_tsk_thread_flag(struct task_struct *tsk, int flag);`
Set thread flag (`TIF_...`) in structure of other task.

`void clear_tsk_thread_flag(struct task_struct *tsk, int flag);`
Clear thread flag (`TIF_...`) in structure of other task.

`int test_and_set_tsk_thread_flag(struct task_struct *tsk, int flag);`
Set thread flag and return previous value.

`int test_and_clear_tsk_thread_flag(struct task_struct *tsk, int flag);`
Clear thread flag and return previous value.

`int test_tsk_thread_flag(struct task_struct *tsk, int flag);`
Return value of a flag.

`void set_tsk_need_resched(struct task_struct *tsk);`
Set `TIF_NEED_RESCHED` flag.

`void clear_tsk_need_resched(struct task_struct *tsk);`
Clear `TIF_NEED_RESCHED` flag.

`int signal_pending(struct task_struct *p);`
Return true if a signal is pending.

`int need_resched(void);`
Return true if `TIF_NEED_RESCHED` is set.

7.9 Various Other Process Management Functions

`int on_sig_stack(unsigned long sp);`
Return true if we are on alternate signal stack.

`void mmdrop(struct mm_struct *mm);`
Decrease `mm`'s reference counter and it it drops to zero then free the memory.

`next_task(p)`
: This macro returns the next task to task `p`.

`prev_task(p)`
: This macro return the previous task to task `p`.

`for_each_process(p)`
: Enumerate through list of all processes.

`task_struct *thread_group_leader(task_struct *p);`
: Return true if `p` is leader of a thread group.

`int thread_group_empty(task_t *p);`
: Return true if `p` has empty thread group.

`void task_lock(struct task_struct *p);`
: Lock the `p->alloc_lock` and protect `fs`, `files`, `mm`, `ptrace`, `group_info`, `comm` and `cpuset` member variables of `p`. This lock also synchronizes with `wait4()`.

`void task_unlock(struct task_struct *p);`
: Unlock the `p->alloc_lock`, see `task_lock()` for details.

`struct thread_info *task_thread_info(struct task_struct *task);`
: Return pointer to `task`'s thread info. The `struct thread_info` is on the stack.

`unsigned long *task_stack_page(struct task_struct *task);`
: Return pointer to thread's stack page. The `struct thread_info` is on the stack, so the stack is calculated using this structure.

`void setup_thread_stack(struct task_struct *p, struct task_struct *original);`
: Copy `struct thread_info` from `original` to `p`, but maintain correct `struct task_struct::task`.

`unsigned long *end_of_stack(struct task_struct *p);`
: Return pointer to end of stack.

`int need_lockbreak(spinlock_t *lock);`
: This macro returns true if a critical section we're currently in needs to be broken because of another task waiting.

`int lock_need_resched(spinlock_t *lock);`

Return one if critical section needs to be broken either due to another task waiting or preemption is to be performed.

`unsigned int task_cpu(const struct task_struct *p);`

Current CPU for given task `p`.

`void set_task_cpu(struct task_struct *p, unsigned int cpu);`

Assign task `p` to given `cpu`. This only sets `struct task_info::cpu`, but does not migrate anything.

`int frozen(struct task_struct *p);`

Check if a thread has been frozen (due to system suspend).

`int freezing(struct task_struct *p);`

Check whether thread `p` is to be frozen now.

`void freeze(struct task_struct *p);`

Freeze the thread `p`. Warning: It's forbidden on SMP to modify other thread's flags. This could be fixed in future versions.

`int thaw_process(struct task_struct *p);`

Wake up a frozen thread. This should be OK even on SMP, because frozen thread isn't running. Return zero if process was not frozen before calling this function.

`void frozen_process(struct task_struct *p);`

The process is frozen now, change status from freezing to frozen.

`int try_to_freeze(void);`

Return true if the process is trying to freeze.

`pid_t process_group(struct task_struct *task);`

Return process group of given `task`.

`int pid_alive(struct task_struct *p);`

Check whether a task is not zombie. If this test fails then pointers within the `task_struct` are stale and may not be referenced.

`int freezeable(struct task_struct *p);`

Return true if task `p` is freezable. This means that the task is not `current`, zombie, dead, does not have `PF_NOFREEZE` flag set and may be neither stopped nor traced.

`void refridgerator();`

This function with interesting name freezes `current` task and enters the following cycle: set task's state to `TASK_UNINTERRUPTIBLE` and call `schedule()`. This is repeated until task is *unfrozen* by other task (`thaw_process()`). Exported symbol.

`int freeze_processes(void);`

Iterate through all processes and their threads and try to freeze them. Return number of processes we could not freeze.

`void thaw_processes()`

Thaw frozen processes.

Chapter 8

New Driver Model in 2.6

The Linux kernel driver model introduced in 2.6 aims to unite various driver models used in the past. The idea behind is that devices have something in common: they're connected to some type of bus, they often support power management... The driver model adheres to tradition of displaying hardware in tree-like structure.

This model offers better communication with userspace and management of object lifecycles (the implementation hidden behind does reference counting, and it acts like object oriented).

Both buses and devices have their common operations and properties: power management, plug and play, hot-plug support. The common ones were moved from specific bus/device structures to `bus_type` or `struct device`.

The third primitive used by driver mode is class. Let's explain relations between these three entities:

Devices like network card or sound card are usually connected to PCI bus. Most computers have only one PCI bus[1], so this two devices share a bus. The PCI bus is able to do power management together with compliant devices.

Another exemplary bus is USB, with it's support for plug-and-play, power management and hot-plugging. When you unload module for your USB optical mouse, its sensor stops shining and the mouse is without electric power.

These devices have something in common, but they all belong to different classes: `network`, `input` (mouse), `sound`, `pci_bus` and `usb`.

This hierarchy of hardware devices is represented by new file-system also introduced during 2.5 development cycle: the `sysfs`. This file-system (usually mounted at `/sys/`) is completely virtual and contains at its root directories like `[bus/]`, `[class/]`, `[devices/]`, which contain other items based on the HW structure of the system (or logically sorted in `class`).

[1] AGP acts like second logical PCI bus with only device connected to it: the graphics adapter

The graphics adapter connected to AGP would be `[class/graphics/]`, but it will also appear in somewhere `[class/pci_bus/...]`. There are symbolic links to the devices deep in the directory `[class/]`.

The API is defined in <`linux/device.h`>. A device is controlled by a driver, that knows how to talk to this particular type of hardware. The device is implemented in `struct device`, its driver in `struct device_driver` and the class is in `struct class`.

8.1 From Kobject to Sysfs

Kobject is the base type for whole driver model, all classes, devices, drivers, ksets and subsystems are base on kobject. It uses reference counted life cycle management. Kobjects can create hierarchical structures, or they can be embedded into ksets (and further into subsystems). This whole hierarchy is represented by directory tree in `sysfs` and thus exporting the hierarchy to userspace.

8.1.1 Kref API

Whole kobject infrastructure works thanks to reference counting, implemented in `struct kref`. This structure has only one member variable: `atomic_t refcount`, which is the reference counter itself. The API to manipulate this data type (defined in <`linux/kref.h`>) is simple:

`void kref_init(struct kref *kref);`
: Initialize reference counter.

`void kref_get(struct kref *kref);`
: Increment reference counter.

`int kref_put(struct kref *kref, void (*release)(struct kref *kref));`
: Decrement reference counter. Whne the reference counter drops to zero, the `release()` callback is used to clean up object. This function returns 1 if the object was released.

8.1.2 Kobject API

A `struct kobject` defined in <`linux/kobject.h`> is basic object of driver model. It's not used directly, but it's embedded in larger objects[2].

The internals of `struct kobject` are following:

[2]Recall `container_of()` macro mentioned in *Common routines* chapter.

`const char *k_name;`
Name of kobject if longer than `KOBJ_NAME_LEN`.

`char name[KOBJ_NAME_LEN];`
Name of kobject if shorter than `KOBJ_NAME_LEN`.

`struct kref kref;`
Reference counter.

`struct kobject *parent;`
The parent of kobject.

`struct kset *kset;`
The kset into which does this kobject belong. If there's no `parent` for this kobject, then the `kset` is used in `sysfs` hierarchy.

`struct kobj_type *ktype;`
The type of the kobject.

`struct dentry *dentry;`
`sysfs` directory entry.

A kobject basic API is as follows:

`void kobject_init(struct kobject *kobj);`
Initialize a kobject (reference is set to 1 and initialize some of its internals).

`int kobject_set_name(struct kobject *kobj, const char *fmt, ...);`
Set name for the kobject, the arguments are in `printf()`-like format.

`const char *kobject_name(const struct kobject *kobj);`
Get the name of kobject.

`struct kobject *kobject_get(struct kobject *kobj);`
Increase reference counter.

`void kobject_put(struct kobject *kobj);`
Decrease reference counter and potentially release the object (if the reference counter has falled to 0).

`void kobject_cleanup(struct kobject *kobj);`
Free object resources.

`int kobject_add(struct kobject *kobj);`
Add object to the `sysfs` hierarchy.

`void kobject_del(struct kobject *kobj);`
Delete object from the `sysfs` hierarchy.

`int kobject_rename(struct kobject *kobj, const char *new_name);`
Change the name of the object.

`int kobject_register(struct kobject *kobj);`
Initialize and add an object.

`void kobject_unregister(struct kobject *kobj);`
Remove from hierarchy and decrease reference count.

`char *kobject_get_path(struct kobject *kobject, gfp_t gfp_mask);`
Return path associated with given `kobject`. The buffer is allocated using `gfp_mask`[3] The result must be freed with `kfree()`.

Each kobject has associated type (represented by `struct kobj_type` with it. The type take care of sysfs file operations, destructor (when freeing an kobject) and holds attributes.

Kobject is represented by subdirectory in `sysfs`. This directory is a subdirectory of directory that belongs to parent of this kobject. All kobjects in kernel are exported to userspace via `sysfs`. A kobject whose `parent` member variable has `NULL` value is toplevel object in `sysfs` hierarchy.

8.1.3 Kset API

Kobject can be gathered into sets (named ksets, `struct kset`). Kset is also a kobject of some and it is able keep kobject of one type. The kset belongs to subsystem (which is also kset, but with a semaphore, as will be explained later). The kset is internally protected by spinlock.

Kobjects are used to simulate hierarchies of objects, either using `parent` pointer or ksets. The ksets are further part of the subsystems.

`void kset_init(struct kset *k);`
Initialize a kset for use (also the internal spinlock).

`int kset_add(struct kset *k);`
Add a kset `k` to hierarchy.

`int kset_register(struct kset *k);`
Initialize and add kset.

[3]See *Memory allocation* chapter for `gfp_t`.

`void kset_unregister(struct kset *k);`

Remove from hierarchy and decrease reference counter.

`struct kset *to_kset(struct kobject *kobjectj)`

Return pointer to kset in which is the `kobject` embedded.

`struct kset *kset_get(struct kset *k);`

Increate reference counter.

`void kset_put(struct kset *k);`

Decrease pointer and potentially release kset.

`struct kobj_type *get_ktype(struct kobject * k);`

Return the type of the object. If the object is in a set and the set has type set, return this one. Otherwiser return kobject's type.

`struct kobject *kset_find_obj(struct kset *k, const char *name);`

Find a kobject in kset by its name. Lock internal spinlock and iterate over a list of kobjects. If a kobject is found, increase its reference counter and return pointer to it. The lock is unlocked before return.

`set_kset_name(str);`

This macro is an initializer, when name is the only kset's field to be initialized.

8.1.4 Subsystem API

Subsystem (`struct subsystem`) contains only two member variables, a `kset` and rw semaphore `rwsen`. Subsystem can contain attribute (`struct subsys_attribute`), which are controllable from userspace. Each `subsys_attribute` contains two callbacks: `show()` to display and `store()` to store attribute. These attributes are exposed to user space through `sysfs`.

Subsystem operations are mostly wrapped kset ones:

`decl_subsys(name, type, uevent_ops);`

Declare a subsystem of *name*`_subsys` name, with given `type` and `uevent_ops`.

`kobj_set_kset_s(obj, subsystem);`

This macro sets kset for embedded kobject `obj` to be the same as is set in `subsystem`. `obj` is pointer to kobject, but subsystem is not a pointer.

`kset_set_kset_s(obj, subsystem);`

This macro sets kset for embedded kset. `obj` is pointer to kobject, but subsystem is not a pointer.

`subsys_set_kset(obj, subsysten);`
: This macro sets kset for subsystem. `obj` is pointer to kobject, but subsystem is not a pointer.

`void subsystem_init(struct subsystem *subsys);`
: Initialize internal rw semaphore and kset.

`int subsystem_register(struct subsystem *subsys);`
: Register subsystem, this makes kset point back to subsystem

`void subsystem_unregister(struct subsystem *subsys);`
: Unregister internal kset.

`struct subsystem *subsys_get(struct subsystem *s);`
: Increase reference counter.

`void subsys_put(struct subsystem *s);`
: Decrease reference counter and if reference count drops to zero, release subsystem's kset.

`int subsys_create_file(struct subsystem *s, struct subsys_attribute *a);`
: Export `sysfs` attribute file.

`void subsys_remove_file(struct subsystem *s, struct subsys_attribute *a);`
: Remove `sysfs` attribute file.

8.1.5 Kernel to User Space Event Delivery

Kobject

`void kobject_uevent(struct kobject *kobj, enum kobject_action action);`
: Notify user space by sending an event. The `action` is one of following:

  `KOBJ_ADD`
  : An object has been added (usable exclusively by kobject core).

  `KOBJ_REMOVE`
  : An object has been added (usable exclusively by kobject core).

  `KOBJ_CHANGE`
  : Device state has changed.

`KOBJ_MOUNT`
: Mount event for block devices. Comment in 2.6.16 source code says it's broken.

`KOBJ_UMOUNT`
: Umount event for block devices. Comment in 2.6.16 source code says it's broken.

`KOBJ_OFFLINE`
: Device is offline.

`KOBJ_ONLINE`
: Device is online.

If there is netlink socket opened, the notification is first sent through it. Moreover if `uevent_helper` is set then it's executed with `subsystem` as it's first argument. The environmental variables are set to describe the event.

```
int add_uevent_var(char **envp, int num_envp, int *cur_index, char
    *buffer, int buffer_size, int *cur_len, const char *format,
    ...);
```

Helper for creating environmental variables. `envp` is pointer to table of environmental variables, `num_envp` is the number of slots available, `cur_index` is the pointer to index into `envp` (i.e. points to the first free slot in `envp`. This index should be initialized to 0 before first call to `add_uevent_var()`). `buffer` points to buffer for environmental variables as passed into `uevent()` method. `cur_len` is pointer to current length of space used in `buffer`. `format` is `printf()`-like format.

This helper return 0 if environmental variable was added successfully or `-ENOMEM` if there wasn't enough space available.

8.2 Bus

The bus API is accessible through <`linux/device.h`>. The basic type is the `bus_type` structure, which contains the `name` of the bus e.g. "pci", list of devices and drivers and attributes, but the most interesting are its operations:

```
int (*match)(struct device *dev, struct device_driver *drv);
```

This callback is called whenever a new device is added to the bus. It is called multiple times and each time it compares device (`dev`) with one driver (`drv`).

If they match, i.e. the driver is able to manage the device, this callback should return non-zero value.

`int (*uevent)(struct device *dev, char **envp, int num_envp, char *buffer, int buffer_size);`
This callback is useful for sending events to user space, for example to load some module, configure the device etc. See kobject API.

`int (*probe)(struct device *dev);`
Probe for a device.

`int (*remove)(struct device *dev);`
Remove device from the bus.

`void (*shutdown)(struct device *dev);`
Shutdown the device.

`int (*suspend)(struct device *dev, pm_message_t state);`
Power management: suspend device.

`int (*resume)(struct device * dev);`
Resume the device. For example on PCI this first resumes PCI config space, then enable the device and finally if the device has been bus master before make it bus master again.

The API concerning bus management is as follows:

`int bus_register(struct bus_type * bus);`
Register a bus with the system.

`void bus_unregister(struct bus_type * bus);`
Unregister the child subsystems and the bus itself.

`void bus_rescan_devices(struct bus_type * bus);`
This functions scans the bus for devices without assigned drivers and tries to match the devices with existing drivers. If any device/driver pair matches, `device_attach()` attaches device to driver.

`struct bus_type *get_bus(struct bus_type * bus);`
Increment reference counter for `bus` object.

`void put_bus(struct bus_type *bus);`
Decrement reference counter for `bus` object.

`struct bus_type *find_bus(char *name);`
Find bus by given bus `name`, return pointer to found `bus_type` or NULL if the bus was not found.

`int bus_for_each_dev(struct bus_type *bus, struct device *start, void *data, int (*fn)(struct device *, void *));`
Iterate over a `bus`'s list of devices starting from `start`. `fn(device, data)` callback is called for each device. Return value of `fn()` is checked after each call, non-zero value breaks iteration and this function will return that value. Reference counter for device that broke iteration is not incremented, it must be done in callback if it's needed. If the iteration is successfully finished, the return value is 0.

`struct device * bus_find_device(struct bus_type *bus, struct device *start, void *data, int (*match)(struct device *, void *));`
This function is similar to `bus_for_each()`, but it tries to find particular device. The `match(device, data)` callback returns non-zero when the device is found and this function exits with return value pointing to `struct device` of found device. If no suitable device was found, return value is NULL.

`int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void * data, int (*fn)(struct device_driver *, void *));`
Iterate over a `bus`'s driver list. Return value of callback `fn(driver, data)` is checked after each call and if it's zero the iteration is broken and this return value is also return by `bus_for_each_drv()`. Otherwise the return value is zero. If the caller needs to access `struct device_driver` that caused iteration to fail, its reference counter must be manually increased in callback.

`int bus_create_file(struct bus_type bus*, struct bus_attribute *attribute);`
Create an attribute file for given `attribute` of a `bus` in codesysfs.

`void bus_remove_file(struct bus_type *bus, struct bus_attribute *attribute);`
Remove attribute file from `sysfs`.

An attribute is a `struct bus_attribute`, it's a mechanism to provide some interface to user space for setting/getting values. The `bus_attribute` supports two operations: `show()` to show value kept inside and `store()` to store a new value. New attribute can be declarede using `BUS_ATTR(name, mode, show, store)`, which declares `struct bus_attribute` of name `bus_attr_`*`name`*`.`

8.3 Drivers

Driver (represented in our model by `struct device_driver`) has a `name` and is connected to some particular `bus`. It keeps pointer to module in which it's implemented. The driver has its operations and attributes, which can be (as in other object in this model) exported to user space. The driver uses `struct completion unloaded` to unload itself only if reference count has dropped to zero.

The device offers following operations for probing, removing, suspending and resuming devices (see bus_type operations):

```
int (*probe)(struct device *dev);
int (*remove)(struct device *dev);
void (*shutdown)(struct device *dev);
int (*suspend)(struct device *dev, pm_message_t state);
int (*resume)(struct device *dev);
```

Driver attributes contain reader and writer routines to `show()` the value to user space or retrieve a new one using and `store()` it back to attribute. Driver attribute can be declared using macro `DRIVER_ATTR(name, mode, show, store)`, the variable will be named `driver_attr_`*`name`*.

Driver API is following:

`extern int driver_register(struct device_driver *driver);`
: Register a driver with a bus and initialize completion.

`extern void driver_unregister(struct device_driver * drv);`
: Remove driver from the system: the completion will block us until reference count drops to zero.

`extern struct device_driver * get_driver(struct device_driver *driver);`
: Increase reference counter of given `driver`.

`extern void put_driver(struct device_driver *drv);`
: Decrease reference counter of given `driver`.

`extern struct device_driver *driver_find(const char *name, struct bus_type *bus);`
: Find a driver on a bus by its `name`.

`int driver_create_file(struct device_driver *driver, struct driver_attribute *attribute);`
: Create a file in `sysfs` for given `attribute`.

`void driver_remove_file(struct device_driver *driver, struct driver_attribute *attribute);`
Remove a file from `sysfs` for given `attribute`.

`int driver_for_each_device(struct device_driver *driver, struct device *start, void *data, int (*fn)(struct device *, void *));`
Iterate through devices bound to `driver`, starting from `start` device. `fn(device, data)` callback will be called for each device found.

`struct device * driver_find_device(struct device_driver *driver, struct device *start, void *data, int (*match)(struct device *, void *));`
Iterate throiugh devices bound to `driver` similarly as in `driver_for_each_device()`. The difference is in callback: `match(device, data)` returns zero for device that doesn't match and non-zero for a matching device. The matching device is returned as the return value of this function (the first match also cancels iteration). If no device was matched, return `NULL`.

`void driver_attach(struct device_driver *drv);`
Walk through the device list of the bus this driver is attached to and try to match the driver with each device. If `driver_probe_device()` returns 0 and `device->driver` is set, a compatible pair was found and the driver will now manage the device.

8.4 Classes

A device class provides higher level view of a device. Classes allow user space software to work with devices based on 'what they are' or 'what they do' rather than 'which bus are they connected to'. Each class is also a subsystem.

Basic data type representing class is `struct class`, which contains:

`const char *name;`
The name of the class.

`struct module *owner;`
Pointer to `struct module` which owns the class.

`struct subsystem subsys;`
Subsystem for this class.

`struct semaphore sem;`
Semaphore that locks both the children and the interface lists.

`struct class_attribute *class_attrs;`
Pointer to array of class attributes. The last attribute must have empty `name`.

`struct class_device_attribute *class_dev_attrs;`
Pointer to array of class device attributes. The last attribute must have empty `name`.

`int (*uevent)(struct class_device *dev, char **envp, int num_envp, char *buffer, int buffer_size);`
This callback is useful for sending events to user space.

`void (*release)(struct class_device *device);`
This callback is called when a `device` is going to be released from this device class.

`void (*class_release)(struct class *class);`
This callback will be called on class release.

The API for device class manipulation is following:

`int class_register(struct class *class);`
Register a new device `class`, create attribute files in `sysfs` and create a subsystem for this `class`.

`void class_unregister(struct class *class);`
Unregister `class` and its subsystem and remove corresponding files.

`struct class *class_create(struct module *owner, char *name);`
Create a `struct class` with given `owner` and `name`. This is used to create a class that can be used in calls to `class_device_create()`. The newly created class is also registered. The `release()` and `class_release()` callback are initialized.

This structure should be destroyed using `class_destroy()` call.

`void class_destroy(struct class *class);`
Destroy a class previously created with `class_create()`

`struct class *class_get(struct class *class);`
Increment reference counter for given `class` and return pointer to it.

`void class_put(struct class *class);`

Decrement reference counter for given `class` and release it if the counter has dropped to zero.

Each class can have an array of accompanying attributes. Each attribute is represented by `struct class_attribute` with `show()` and `store()` callbacks to get and set the attribute value:

```
ssize_t (*show)(struct class *, char *);
ssize_t (*store)(struct class*, const char *buf, size_t count);
```

Class attribute can be declared using `CLASS_ATTR(name, mode, show, store)` macro, which declares and initializes variable `class_attr_`*`name`*. `show()` and `store()` are attribute access functions as described before.

`int class_create_file(struct class *class, const struct class_attribute *attribute);`

Create a file representing given `attribute`.

`void class_remove_file(struct class *class, const struct class_attribute *attribute);`

Remove file for `attribute`.

8.5 Class Devices

Class devices are devices belonging to particular class. `struct class_device` contain following member variables:

`struct class *class;`

Pointer to the parent class for this class device. Required in all instances.

`struct class_device_attribute *devt_attr;`

For internal use by the driver core only.

`struct class_device_attribute uevent_attr;`

Attribute that represents events for user space.

`struct device *device;`

Points to `struct device` of this `device`, this is represented by symlink in `sysfs` hierarchy.

`void *class_data;`
: Class device specific data, driver developer may use this field for anything. Use `class_get_devdata()` and `class_set_devdata()` to access this field.

`struct class_device *parent;`
: Parent of this device if there is any. If NULL then this device will show up at this class' root in `sysfs` hierarchy.

`void(*release)(struct class_device *dev);`
: This callback is used during release of `struct class_device`. If it's set, it will be called instead of the class specific `release()` callback. The developer should use in cases like releasing nested `class_device` structures.

`int(*uevent)(struct class_device *dev, char **envp, int num_envp, char *buffer, int buffer_size);`
: This callback is used to send event notifications to user space. If this pointer to function is not set, the class-specific `uevent()` callback will be used.

`char class_id[BUS_ID_SIZE];`
: Unique ID for this class.

The API for `struct class_device` manipulation is similar to other kobject APIs:

`void *class_get_devdata(struct class_device *dev);`
: Return pointer to class specific data.

`void class_set_devdata(struct class_device *dev, void *data);`
: Set pointer to class specific data.

`void class_device_initialize(struct class_device *dev);`
: Initialize the class device. See `class_device_register()` for most cases.

`int class_device_register(struct class_device *dev);`
: Initalize the class device and add it to it's class. (see `class_device_add()`).

`void class_device_unregister(struct class_device *dev);`
: Unregister the class device and if the reference counter for this structure has is zero, release the structure.

`int class_device_add(struct class_device *device);`
: Add `device` to concerned class and configure user space event notification. Create files for attributes. At the end an event `KOBJ_ADD` is signalled to user space.

`void class_device_del(struct class_device *dev);`

`class_id`, `parent_class` and `dev` member variables must be set before this call is made.

`int class_device_add(struct class_device *device);`

Remove `device` from parent class, remove symlinks and attribute files and issue `KOBJ_REMOVE` event to user space.

`int class_device_rename(struct class_device *dev, char *name);`

Rename the class device.

`struct class_device *class_device_get(struct class_device *dev);`

Increment the reference counter for this structure.

`void class_device_put(struct class_device *dev);`

Decrement the reference counter and release the structure when it reaches zero.

`struct class_device *class_device_create(struct class *cls, struct class_device *parent, dev_t devt, struct device *device, char *fmt, ...);`

Create a class device and register it with `sysfs` hierarchy. `cls` is the pointer to `class` that this device should be registered to. `parent` class device may be NULL pointer. `dev_t devt` is for character device to be added, `device` points to `struct device` associated with this class device. `fmt` and following arguments represent `printf`-like description of this class device's name.

This function can be used by character device classes, `struct class_device` will be allocated, initialized an registered inside this function.

`void class_device_destroy(struct class *cls, dev_t devt);`

Remove a class device that was created with `class_device_create()`. `cls` is the class to which the class device belongs and `devt` identifies the device.

Class device attribute (`struct class_device_attribute`) offers the same functionality as other attributes in this driver model: `show()` and `store()` operations. The attribute can be conveniently declared using `CLASS_DEVICE_ATTR(name, mode, show, store)` macro, which declares `class_device_attr_`*`name`* variable and initializes it's operations to given callbacks.

`int class_device_create_file(struct class_device *dev, const struct class_device_attribute *attribute);`

Create a file in `sysfs` hierarchy representing given `attribute`.

`void class_device_remove_file(struct class_device *dev, const struct class_device_attribute *attribute);`
Remove a file representing an `attribute`.

`int class_device_create_bin_file(struct class_device *dev, struct bin_attribute *attribute);`
Create a file in `sysfs` hierarchy representing given binary `attribure`

`void class_device_remove_bin_file(struct class_device *dev, struct bin_attribute *attribute);`
Remove a file associated with `attribute`.

A binary attribute is similar to forementioned attributes, but it stores binary data and it's operations are as follows:

`ssize_t (*read)(struct kobject *kobj, char *buffer, loff_t offset, size_t size);`
Read `size` chars from `offset` of binary attribute to `buffer`. `kobj` is the object to which this attribute belongs.

`ssize_t (*write)(struct kobject *kobj, char *buffer, loff_t offset, size_t size);`
Write `size` chars from `buffer` to binary attribute from given `offset` and `size`. `kobj` is the object to which this attribute belongs.

`int (*mmap)(struct kobject *kobj, struct bin_attribute *attr, struct vm_area_struct *vma);`
Map binary attribute to user space memory, `vma` is the VM area to be used, `attribute` is the attribute belonging to `kobj` object.

8.6 Class Interface

The last structure introduced in class section is `struct class_interface`, which contains four interesting pointers:

`struct list_head node;`
This instance's entry into list of observers of concerned `class`.

`struct class *class;`
Points to class to which is this interface bound and on which we monitor adding and removing of devices.

`int (*add) (struct class_device *dev, struct class_interface *interface);`
`add` points to function which will be called when some device is added to observed class. `dev` is the device that is to be added.

`void (*remove) (struct class_device *device, struct class_interface *interface);`
This pointer points to function that will be called upon device removal. `dev` is the device that will be removed.

This structure represents observer that gets notification about device adding and removal, the only API for manipulation with observer are functions for registering an unregistering an interface.

`int class_interface_register(struct class_interface *observer);`
Add an `observer` to chain of observers. The class which will be observed is `observer->class`.

`void class_interface_unregister(struct class_interface *iface);`
Remove an observer.

8.7 Devices

Let's describe `struct device`:

`struct device *parent;`
This is the parent device, i.e. the device to which is this device attached, usually a bus. A `NULL` parent means top-level device.

`struct kobject kobj;`
The `kobj` keeps this device in the `sysfs` hierarchy.

`char bus_id[BUS_ID_SIZE];`
Unique string identifying this device on the bus.

`struct device_attribute uevent_attr;`
Attribute, that has `store()`

`struct semaphore sem;`
A semaphore to synchronize calls to driver.

`struct bus_type *bus;`
The bus this device is connected on.

`struct device_driver *driver;`
The driver managing this device.

`void *driver_data;`
Private driver data.

`struct dev_pm_info power;`
Power state of the device.

`void (*release)(struct device *dev);`
This method releases the device, when the reference counter drops to zero.

The common driver API is following:

`int device_register(struct device *device);`
Register a device with the `sysfs` hierarchy, first initialize the device and then add it to system (i.e. to the bus). This function also sets many internal member variables inside `struct device`.

`void device_unregister(struct device *device);`
Release the device from all subsystems and then decrement the reference count. If it has dropped to zero, the device is released by `device_release()`. Otherwise the structure will be hanging around in memory until its reference count drops to zero.

`void device_initialize(struct device *device);`
Initialize `struct device` and prepare device to be woken up. See `device_add()` and note about `device_register()`.

`int device_add(struct device *device);`
Adds the device to `sysfs` hierarchy (bus etc). This function is usually preceded by `device_initialize()`, which is what `device_register()` exactly does. `device_register()` is the preferred way.

`void device_del(struct device *device);`
Remove the device from lists it's kept in, notify power management and platfom dependent functions about removal. This should be called manually only if `device_add()` was also called manually, see `device_unregister()`.

`int device_for_each_child(struct device *parent, void *data, int (*fn)(struct device *, void *));`
Iteratate over a list of children of `parent`al device and call `fn(child_device,`

`data)` on each one. If the `fn()` returns non-zero value, break the iteration and return this value.

`void device_bind_driver(struct device *dev);`
Manually attach a driver: `dev->driver` must be set. This function does not manipulate bus semaphore nor reference count. This function must be called with `dev->sem` held for USB drivers.

`void device_release_driver(struct device *dev);`
Manually release driver from device. This function does not manipulate bus semaphore nor reference count. This function must be called with `dev->sem` held for USB drivers.

`int device_attach(struct device *dev);`
Walk the list of the drivers attached to bus on which is the device connected and call `driver_probe_device()` for each one. If a compatible pair is found, break iteration and bind the driver to this device. This function returns 1 if the driver was bound to this device, 0 if no matching device was found or error code otherwise.

`struct device *get_device(struct device *dev);`
Increase the reference counter.

`void put_device(struct device *dev);`
Decrease the reference counter.

`void device_shutdown(void);`
Shut down all devices. System devices are shut down as the last ones. This calls `shutdown()` operation on each device.

`void *dev_get_drvdata (struct device *dev);`
Get driver specific data for this device.

`void dev_set_drvdata(struct device *dev, void *data);`
Set driver specific data for this device.

`int device_is_registered(struct device *dev);`
Return whether a device is registered.

Each device also has its set of attributes (just like bus, subsystem...) for importing an exporting values. Device attribute (`struct device_attribute`) is declared

using `DEVICE_ATTR(name, mode, show, store)` macro, which declares and initializes variable named `dev_attr_`*`name`*`. show()` and `store()` operations are available for getting/setting values from user space.

`int device_create_file(struct device *device, struct device_attribute *entry);`
Create a file in `sysfs` that will represent given attribure.

`void device_remove_file(struct device *device, struct device_attribute *attr);`
Remove file in `sysfs` that representing given attribure.

8.8 Extended structures

Existing buses have their own `bus_type` variables: `pci_bus_type` (<`linux/pci.h`>), `mca_bus_type` (<`linux/mca.h`>), `ide_bus_type` (<`linux/ide.h`>), `eisa_bus_type` (<`linux/eisa.h`>) and `usb_bus_type` (<`linux/usb.h`>).

A driver is tied tightly to a bus and most of existing buses use extended versions of `struct device_driver`, e.g. `struct usb_driver`, `struct pci_driver`, `struct mca_driver`, `struct ide_driver_s` (or `ide_driver_t`) and `struct eisa_driver`.

We'll look more closely on some member variables of `struct usb_driver`:

`const char *name;`
This is the driver name, it should be unique among USB drivers and it also should be the same as the module name.

`int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);`
This callback is called to check whether the driver is willing to manage a particular interface on a device.

If it is so, zero is returned and `dev_set_drvdata()` from the device API to associate driver specific data with the interface.

It the driver is not willing to manage the device, it must return negative error value.

`void (*disconnect) (struct usb_interface *intf);`
This callback is called when an interface is no longer available: either the device has been unplugged or the driver is being unloaded.

`int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);`
Handler for `usbfs` events, this provides more ways to provide information to user space.

`int (*suspend) (struct usb_interface *intf, pm_message_t message);`
Called when the device is going to be suspended (power management).

`int (*resume) (struct usb_interface *intf);`
Called when the device is going to be resumed (power management)¿

`struct device_driver driver;`
This is the device driver's structure for driver model.

Wary reader surely noticed `struct usb_interface`. This structure contains both `struct device` and `struct class_device` and some other internals.

It's common for <`linux/`*`name_of_a_bus`*`.h`> files to offer API for that particular type of a bus, developer should look inside those files as detailed description is behind scope of this work.

Chapter 9

Common Routines and Helper Macros

9.1 String Manipulation

Following routines accessible through <linux/kernel.h>:

`int printk(const char* format, ...);`

Output the message to console, dmesg and syslog daemon. For formatting options see `man 3 printf`. The output should contain priority, which can be added in the following manner:

```
printf(KERN_WARNING "%s: module license '%s' taints kernel.\n",
    mod->name, license);
```

These priorities are defined also in [kernel.h]:

KERN_EMERG
: Defined as <0>, means "system is unusable".

KERN_ALERT
: Defined as <1>, means "action must be taken immediately".

KERN_CRIT
: Defined as <2>, means "critical conditions".

KERN_ERR
: Defined as <3>, means "error conditions".

KERN_WARNING
: Defined as <4>, means "warning conditions".

`KERN_NOTICE`
Defined as `<5>`, means "normal but significant condition".

`KERN_INFO`
Defined as `<6>`, means "informational".

`KERN_DEBUG`
Defined as `<7>`, means "debug-level messages".

This function internally calls vprintk, which uses 1024 byte buffer and does not check for overruns.

int printk(const char *format, va_list args); This is the kernel mode equivalent of `vprintf`, first see `man 3 vprintf` and then explanation of `printk()` above.

sprintf(char *buf, const char* format, ...); The kernel mode equivalent of usual user space `sprintf()`.

vsprintf(char *buf, const char* format, va_list args); The kernel mode equivalent of usual user space `vsprintf()`.

snprintf(char *buf, size_t size, const char* format, ...); The kernel mode equivalent of usual user space `snprintf()`.

vsnprintf(char *buf, size_t size, const char* format, va_list args); The kernel mode equivalent of usual user space `vsnprintf()`.

sscanf(const char *buf, const char *format, ...); The kernel mode equivalent of usual user space `sscanf()`.

vsscanf(const char *buf, const char *format, va_list args); The kernel mode equivalent of usual user space `vsscanf()`.

`int get_option(char **string, int *value);`
Parse an integer `value` from an option `string`. A subsequent comma (if any) is skipped and `string` pointer is modified to point behind the parsed int. Return 0 if there's no integer in string, 1 if int was found without comma and 2 if comma was found too.

`char *get_options(const char *string, int nints, int *ints);`
Parse a string containing comma-separated list of integers into a list of integers. `string` is the string to be parsed, `nints` is the size of integer array `ints`. Integers parsed are stored into `ints` from index 1, `ints[0]` contains the number of integers successfully parsed. Return value is the character in

the string which caused end of parsing. If it's null then the string was parsed completely

`char *kstrdup(const char *s, gfp_t gfp);` Allocate space to hold duplicate of strin `s` and copy the string. `gfp` is the mask of flags used in `kmalloc()` call. Prototype is in <`linux/string.h`>

9.2 Various Helper Macros

might_sleep() This macro works as an annotation for function that can sleep, it prints a stack trace if it's executed in an atomic context (spinlock, irq_handler...). It's intended for debugging purposes to help find out whether the function (which shouldn't sleep) sleeps.

might_sleep_if(condition) First evaluate the `condition` and if it's true then evaluate `might_sleep` macro described above.

`min(x,y)`, `max(x,y)`, `min_t(type, x,y)`, `max_t(type, x,y)` Macros that return minimal or maximal value of given pair. `min()` and `max()` also does strict type checking. `min_t()` and `max_t()` allow developer to specify a `type`, both `x` and `y` will be type-casted to it.

`container_of(pointer, type, member)`
If `pointer` points to `member` variable inside the structure of given `type`, return the pointer to the structure itself. See `list_entry()` and list implementation for an example.

`typecheck(type, variable)`
Check at compile time that `variable` is of given `type`. Always evaluates to value 1.

`typecheck_fn(type, function)`
Check at compile time that `function` is of a certain `type` or a pointer to that type. `typedef` must be used to specify for function type.

`BUG()`
This macro (with `CONFIG_BUG` defined) prints message about bug and it's location and then kernel panics. This macro can be used to check various conditional branches that shouldn't be executed.

`BUG_ON(condition)`
This macro first checks the `condition` and if it's true then calls `BUG()`.

`WARN_ON(condition)`
: This macro (with `CONFIG_BUG`) prints message about it's location in file and function on which line it's been placed.

`likely(condition)` and `unlikely(condition)`
: These macros advise compiler which branch is prefered, so the resulting assembly coded will be optimized for this branch.

9.3 User Space Memory Access

Kernel code sometimes needs to access data in user space memory. The file <`asm/uaccess.h`> provides all necessary inline function, macros or prototypes:

`access_ok(type, addr, size)`
: Verify whether the memory area specified by `addr` and `size` is accessible for given `type` of access: `VERIFY_READ` for reading and `VERIFY_WRITE` for writing. Writing is superset of reading. Returns true if the memory is valid. This macro is only valid in user context.

`get_user(kernel_variable, umode_ptr)`
: Copy a simple variable (char, int...) from user space memory to given `kernel_variable`. Returns true on success and `-EFAULT` on error. This macro may sleep and it's valid only in user context.

`put_user(kernel_variable, umode_ptr)`
: Copy a simple variable (char, int...) from `kernel_variable` to address at given user space memory. Returns true on success and `-EFAULT` on error. This macro may sleep and it's valid only in user context.

`__get_user(kernel_variable, umode_ptr)`
: Copy a simple variable (char, int...) from user space memory to given `kernel_variable` with less checking. Returns true on success and `-EFAULT` on error. This macro may sleep and it's valid only in user context.

`__put_user(kernel_variable, umode_ptr)`
: Copy a simple variable (char, int...) from `kernel_variable` to address at given user space memory with less checking. Returns true on success and `-EFAULT` on error. This macro may sleep and it's valid only in user context.

```
unsigned long __copy_to_user_inatomic(void __user *to, const void
    *from, unsigned long n);
```

Copy a block of data into user space. `to` is the destination in user space, `from` is source in kernel space, `n` is number of bytes to copy. This function is valid only in user context. Caller must check the specified block with `access_ok()` before calling this function. Returns the number of bytes that cannot be copied, i.e. zero on success. This function might sleep.

`unsigned long __copy_from_user_inatomic(void *to, const void __user *from, unsigned long n);`

Copy a block of data from user space. `to` is the destination in kernel space, `from` is source in kernel space, `n` is number of bytes to copy. This function is valid only in user context. Caller must check the specified block with `access_ok()` before calling this function. Returns the number of bytes that cannot be copied, i.e. zero on success. This function might sleep.

`unsigned long __copy_from_user(void *to, const void __user *from, unsigned long n);`

Wrapper that calls `__copy_from_user_inatomic()`.

`unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);`

Copy data from kernel space to user space. Valid in user context only. This function may sleep. Return value is the number of bytes that could not be copied, zero on success.

`unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);`

Copy data from user space to kernel space. Valid in user context only. This function may sleep. Return value is the number of bytes that could not be copied, zero on success.

`long strncpy_from_user(char *dst, const char __user *src, long count);`

Copy a NULL terminated string from user space to kernel space, destination `dst` must be at least `count` bytes long. Return value is lenght of the string (NULL omitted), otherwise `-EFAULT`. If `count` is too small, copy `count` bytes and return `count`. This function might sleep.

`long strnlen_user(const char __user *s, long n);`

Get the size of the string `s` in user space. `n` is the maximum valid length. Return value is the lengh of the string or zero in case of error. If string is too long, the return value is `n`.

`strlen_user(str);`

Get the size of the string `s` in user space. Return value is the lengh of the string or zero in case of error. This is a macro evaluates to `strnlen_user(str,MAXINT)`.

`unsigned long clear_user(void __user *mem, unsigned long len);`

Zero a block in user memory. `mem` is destination, `len` is the length of the block in bytes. Return the number of bytes that could not be cleared, zero on success.

Chapter 10

Modules

A module is an object file (but with extension `.ko` instead of `.o`), which can be dynamically inserted to (or removed from) the kernel. The module is dynamically linked with the rest of the kernel and other modules at the insertion time. This implies that modules have dependencies between themselves.

Each module goes through three phases while it's loaded in kernel: initialization, normal operation and clean-up. Initialization phase is usually implemented by simple function which sets up some hooks, which will be used during normal operation. These hooks are unhooked during clean-up phase.

10.1 An Exemplary Module

Let's build an exemplary module named `mousepad.ko`, which will be driver for mouse pad:

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

static char mpad_init[] __initdata="initialized";
static char mpad_exit[] __exitdata="exiting";

static int __init mousepad_init()
{
    printk(KERN_INFO "Mouse pad %s.\n", mpad_init);
    return 0;
}

static void __exit mousepad_exit()
```

```
{
    printk(KERN_INFO "Mouse pad %s.\n", mpad_exit);
}

module_init(mousepad_init);
module_exit(mousepad_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Jaroslav Soltys");
MODULE_DESCRIPTION("Mouse Pad driver");
MODULE_SUPPORTED_DEVICE("mousepad");
```

This driver contains two functions: `mousepad_init()` is used in the initialization phase. Our example just prints string `"Mouse pad initalized."` (probably to `[/var/log/messages]`). The `module_init()` macro selects which function is called during initialization. Remember, if initializing function does not return zero, the module is unloaded.

The other function is used before unloading of module from kernel.[1] It simply prints `"Mouse pad exiting."`. This exit-function is selected by `module_exit()`.

Drivers which made it to official kernel can be either selected to be compiled as modules or they can be built-in. A driver that is built-in cannot be unloaded, therefore exit function is never used. To save memory `__exit` macro was introduced, which decides at compile time whether to include exit function or not.

Similar memory-saving is done with init functions: `__init` macro marks functions used only during initialization phase and then the memory is freed. This is the cause for message

```
Freeing unused kernel memory:  4486k freed
```

during kernel initialization. Once the initialization is completed, the functions marked with `__init` and variables marked with `__initdata` are freed from memory. This is accomplished by putting `__init` functions to `.init.text` section inside object file, `__initdata` to `.init.data` section, `__exit` puts the functions to `.exit.data` section and `__exitdata` puts variables used only from within exit context to `.exit.data`.

[1] You can think of these init and exit functions as constructors and destructors in OOP.

10.2 Kernel Purity

Kernel 2.4 introduced tainted kernels. A tainted kernel is a kernel containing code, which is neither under GPL license nor under any 'compatible' free/open software licenses.

The license of a module is specified by `MODULE_LICENSE()` macro. Kernel purity is indicated by following licenses:

`"GPL"`
: The module is under General Public license version 2 or later.

`"GPL v2"`
: The module is under GPL version 2 license.

`"GPL and additional rights"`
: GNU GPL version 2 rights and more.

`"Dual BSD/GPL"`
: GNU GPL version 2 or BSD license.

`"Dual MPL/GPL"`
: GNU GPL version 2 or Mozilla license code.

Proprietary modules can either use `MODULE_LICENSE("Proprietary")` or anything other than forementioned GPL strings is considered proprietary. Once a kernel becomes tainted, it remains tainted until reboot.

Kernel may be tainted by loading proprietary module, forcing loading or unloading of module, using SMP with CPUs not designed for SMP, machine check experience or bad page. The purity status of a kernel is exported to user space using procfs. `/proc/sys/kernel/tainted` returns 1 for tainted kernel.

10.3 Module Properties

Module properties (such as `MODULE_LICENSE`) can be seen using `modinfo` command. `MODULE_AUTHOR` declares module's author, `MODULE_DESCRIPTION` describes what a module does.

`MODULE_SUPPORTED_DEVICE("device")` is not yet implemented, but it might be used in the future to say that this module implements `[/dev/device]`. However developers should use this macro for documentation purposes.

10.4 Module Parameters

Modules used to take parameters since time of ISA bus, because earlier versions of this bus did not support plug and play operation. It was necessary to specify e.g. port, DMA channel and IRQ for a sound card driver. Modern hardware is usually able to operate without any parameters, but these interface still provides the possibility to send parameters to a module.

Let's augment our existing code:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>
#include <linux/moduleparam.h>

static char mpad_init[] __initdata="initialized";
static char mpad_exit[] __exitdata="exiting";

static int height=15;
module_param(height, int, S_IRUSR|S_IWUSR);

static int width=20;
module_param(width, int, S_IRUSR|S_IWUSR);

#define COLOR_STR_LEN 80
static char color[COLOR_STR_LEN]="deep blue";
module_param_string(color, color, COLOR_STR_LEN, S_IRUSR|S_IWUSR);

#define VER_LEN 3
static int version[VER_LEN]={-1, -1, -1};
static int ver_len=VER_LEN;
module_param_array(version, int, &ver_len, S_IRUSR|S_IWUSR);

static int __init mousepad_init()
{
    int i;

    printk(KERN_INFO "Mouse pad %s.\n",mpad_init);
```

```
    printk(KERN_INFO "The mouse pad is version ");
    for(i=0; i<ver_len; i++)
        printk("%i.", version[i]);
    printk("\n" KERN_INFO "It's size is %ix%i.\n", width, height);
    printk(KERN_INFO "Mouse pad colour: %s\n", color);

    return 0;
}

static void __exit mousepad_exit()
{
    int i;

    printk(KERN_INFO "The mouse pad is version ");
    for(i=0; i<ver_len; i++)
        printk("%i.", version[i]);
    printk("\n" KERN_INFO "It's size is %ix%i.\n", width, height);
    printk(KERN_INFO "Mouse pad colour: %s\n", color);

    printk(KERN_INFO "Mouse pad %s.\n", mpad_exit);
}

module_init(mousepad_init);
module_exit(mousepad_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Jaroslav Soltys");
MODULE_DESCRIPTION("Mouse Pad driver");
MODULE_SUPPORTED_DEVICE("mousepad");

MODULE_PARM_DESC(height, "Mouse pad height");
MODULE_PARM_DESC(width, "Mouse pad width");
MODULE_PARM_DESC(color, "Mouse pad color");
MODULE_PARM_DESC(version, "Version of a mouse pad");
```

We introduced parameters `height` and `width` to hold size of a mouse pad, `color` of the pad and `version` number that consists of up to three dot-separated numbers. Each parameter can be set at command line:

`insmod mousepad.ko height=15 width=20` `color=red version=1,2,3` `mousepad_init()` writes values of these variables to `/var/log/messages`. If there are only two numbers in `version=1,2` parameter, then `var_len` is modified accordingly. Module with parameter `version` more than three numbers can't be loaded into kernel. Similarly `color` is string with limited length, if you try to load module with longer string, the module will fail to load.

Let's describe the simplest type firstr. `int height` is declared as ordinary variable, but it's made accessible as module parameter thanks to `module_param(name)` macro.

This macro takes three arguments: `name` is the first one, it's the name of the variable to be used as module parameter and it's also the name of the parameter. The second one is `type`, which can be one of simple types like `byte`, `short`, `ushort` (unsigned short), `int`, `uint` (unsigned int), `long`, `ulong` (unsigned long), `charp` (char *), `bool` and `invbool` (inverted boolean), more complex types will be described later. The third argument describes file permission in `sysfs`.

Sysfs has a directory called `/sys/module/mousepad/parameters` for our `mousepad.ko` module, which offers virtual file for each parameter. These parameter may be read or written, all of our parameters are RW for owner (root). `cat version` prints version, while `echo 4,5 > version` sets first and second numbers of `version` array and also sets `ver_len` to 2. You must always set at least one number in array, and you can't skip any number.

To export an array of simple types as module parameter, use `module_param_array()` macro with four parameters: `name`, `type` of a single item inside an array, `array_length` (this must be pointer to `int` variable, as this value can be decreased when shorter array is sent from user space) and again file permissions.

The last type described is string `color`. The strings are exported using `module_param_string()`, which uses four parameters: name, string, buffer size (to prevent buffer overrun, kernel won't allow you to load module with longer parameter or write to belonging file more bytes than this limit allows) and finally file permissions.

For example `echo red > color` starts internal kernel mechanism that stores zero-terminated string `"red"` into `color` buffer.

Each parameter has a description assigned to it, `MODULE_PARM_DESC(version, "Version of a mouse pad");` says what's the meaning of a parameter. Use `/sbin/modinfo` to display this information.

10.5 Building Modules

It's convenient to include a `Makefile` with your module's source code. Kbuild in 2.6 makes building modules using makefiles easier then ever:

```
obj-m+=mousepad.o
KDIR=/lib/modules/$(shell uname -r)/build

all:
    make -C $(KDIR)  M=$(PWD) modules

clean:
    make -C $(KDIR)  M=$(PWD) clean

install:
    make -C $(KDIR)  M=$(PWD) modules_install
```

If you're not building module for current kernel, change KDIR variable to point it to relevant kernel source.

Kbuild offers ready-to-use makefiles, which take following variables (only those interesting are listed):

`obj-m`
: List of modules you want to build (`[mousepad.o]` in our case).

modulename`-y`
: List of object files that will be linked to create the module, if the module is split into multiple files.

`obj-y`
: List of modules that will be *built-in* in the kernel. (this is not interesting for external modules).

`obj-n`
: List of modules that won't be built.

`INSTALL_MOD_DIR`
: The relative path inside the kernel's module collection and selects where will be the module located. If this variable is not set then `[extra/]` is used as default.

The most important make targets are:

`modules`
: Build modules.

`clean`
: Clean target `.ko` and temporary files.

`modules_install`
: Install the module into `[/lib/modules/ùname -r/extra/]`. See `INSTALL_MOD_DIR` to place module somewhere else than into `[extra/]` directory.

See `[Documentation/kbuild/]` for more in-depth information.

10.6 Exporting Symbols

When a module is loaded into kernel it needs to resolve references to some kernel functions. These symbols can be exported using one of following macros:

`EXPORT_SYMBOL(name)`
: This macro exports symbol of given `name`. This symbol can be used by any module. This macro also for documentation purposes marks the symbol as external.

`EXPORT_SYMBOL_GPL(name)`
: This macro exports symbol of given `name` to GPL-complatibly licensed modules. This symbol is internal for documentation purposes. Modules under proprietary licenses cannot access this module.

Chapter 11

Conclusion

My goal was to describe Linux kernel 2.6 internals. There were no books available and only a few resources describing 2.6 at the time I started working on this thesis. There was even no stable kernel 2.6 released. Studying Linux kernel is a hard nut to crack in the beginning, and this thesis aims to fill some of the gaps between theoretical knowledge of operating systems' inner working and practical expertise of Linux kernel.

Writing about software under heavy development is like shooting at the moving target. Some of the internal interfaces mentioned here were changed after I described them for the first time. Some other interfaces became deprecated in favor of other ones, other ones evolved in some way.

The assignment proved to be too general, yet I have describe task scheduler and process management deeper than just the API. Inner working of scheduler can be of great interest for people studying optimization of parallel algorithms.

I put strong emphasis on locking issues and therefore I delved deeper into various techniques provided.

Description of other subsystems should be sufficient to understand basics of how the subsystem works and the API needed for module developers to use the functionality provided by particular subsystem. Interesting and comprehensive is the description of driver infrastructure.

This thesis however does not drain every possible information from Linux kernel, there are plenty of opportunities for deeper studies: networking infrastructure, virtual file system or memory management internals would make a theses by themselves.

I learned many things while I was studying the kernel and I hope that this thesis will help me to share this knowledge with others. Time spent with kernel source, the view of changes, new features and performance improvements introduced to kernel

have convicted me that Linux has very bright future.

Bibliography

[1] Daniel P. Bovet and Marco Cesati. *Understanding Linux Kernel*. O'Reilly Media, Inc., 2nd edition edition, December 2002. This book covers kernel 2.4.

[2] Daniel P. Bovet and Marco Cesati. *Understanding Linux Kernel*. O'Reilly Media, Inc., 3rd edition edition, November 2005. This book covers kernel 2.6.

[3] Ladislav Lhotka. *Server v Internetu*. Kopp, 1st edition edition, 2006. Published in czech language.

[4] Dirk Louis, Petr Mejzlík, and Miroslav Virius. *Jazyk C a C++ podle normy ANSI/ISO*. Grada, 1st edition edition, 1999. Published in czech language.

[5] Michael Lucas. *Síťový operační systém FreeBSD*. Computer Press, 1st edition edition, 2003. Published in czech language.

[6] Linux Kernel Mailing List <linux-kernel@vger.kernel.org>, archive is accessible on http://lkml.org/.

[7] Linux Mailing List <linux@lists.linux.sk>, Slovak Linux mailing list, archive accessible on http://lists.linux.sk/.

[8] http://wiki.kernelnewbies.org/.

[9] http://www.kernelnewbies.org/.

[10] http://kerneltrap.org/.

[11] http://kniggit.net/wwol26.html.

[12] http://lse.sf.net/.

[13] http://lwn.net/.

[14] http://lwn.net/Kernel/Index/.

[15] http://lwn.net/Kernel/LDD3/.

[16] http://lxr.linux.no/.

[17] http://bugzilla.kernel.org/.

[18] http://ftp.kernel.org/pub/linux/libs/security/linux-privs/kernel-1.4/capfaq-0.2.txt.

[19] http://oprofile.sf.net/.

[20] http://people.redhat.com/drepper/futex.pdf.

[21] http://people.redhat.com/drepper/nptl-design.pdf.

[22] http://procps.sf.net/.

[23] http://tech9.net/rml/procps/.

[24] http://thomer.com/linux/migrate-to-2.6.html.

[25] http://www.csn.ul.ie/~mel/projects/vm/guide/.

[26] http://www.geocities.com/marco_corvi/games/lkpe/index.htm.

[27] http://www.gnugeneration.com/mirrors/kernel-api/book1.html.

[28] http://www.hpl.hp.com/research/linux/kernel/o1.php.

[29] http://www.informit.com/articles/article.asp?p=336868.

[30] http://www.kernel.org/pub/linux/kernel/people/bcrl/aio/.

[31] http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html.

[32] http://www.linuxjournal.com/article/5755.

[33] http://www.linux.org.uk/~davej/docs/post-halloween-2.6.txt.

[34] http://www.scs.ch/~frey/linux/.

[35] http://www.sf.net/projects/linuxquota/.

[36] http://www.skynet.ie/~mel/projects/vm/guide/html/understand/.

[37] http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html.

[38] http://www.tldp.org/LDP/khg/HyperNews/get/khg.html.

[39] http://www.tldp.org/LDP/tlk/tlk.html.

[40] http://www-users.cs.umn.edu/~bentlema/unix/advipc/ipc.html.

[41] http://www.uwsg.indiana.edu/hypermail/linux/kernel/0306.3/1647.html.

[42] http://www.wikipedia.org/.

[43] http://www.xmailserver.org/linux-patches/nio-improve.html.

[44] http://www.kroah.com/log/.

[45] http://www.linux-mm.org/.

[46] http://www.kroah.com/linux/talks/suse_2005_driver_model/index.html.

[47] IRC channel #kernelnewbies on server irc.oftc.net.

[48] IRC channel #slackware on server irc.oftc.net.

Glossary

- `i386` - Processor architecture compatible with Intel 80386 processor.
- `ACL` - Access Control List
- `AIO` - Asynchronous Input/Output
- `API` - Application Program Interface
- `BSD` - Originally Berkeley Software Distribution, this abbreviation refers either to one of many BSD derivates of Unix or to BSD license.
- `CPU` - Central Processing Unit
- `DCC` - Direct Client-to-Client connection used on IRC network either to transfer files or to chat directly (i.e. not through the server).
- `FD` - File Descriptor
- `FTP` - File Transfer Protocol
- `GDT` - Global Descriptor Table, i386-specific table used to access segments of memory
- `HAL` - Hardware Abstraction Layer
- `HEL` - Hardware Emulation Layer
- `ICMP` - Internet Control Message Protocol
- `IFS` - Installable File System, an IFS-kit enables developer to write a new filesystem for Windows family of operating systems.
- `I/O` - Input/Output
- `IP` - Internet Protocol
- `IPIP` - IP in IP tunneling

- `IPX` - Internetwork Packet eXchange
- `IRC` - Internet Relay Chat
- `IRQ` - Interrupt ReQuest
- `ISP` - Ineternet Service Provider
- `KISS` - Keep It Simple, Small
- `LDT` - Local Descriptor Table
- `MMU` - Memory Management Unit, a part of processor that takes care of pagination and segmentation of memory, access right etc.
- `NAT` - Network Address Translation
- `NFS` - Network File System
- `NUMA` - Non-Uniformed Memory Access - rather specific set of multiprocessing machines that has shared memory, but the memory is *closer* to some processors than to other and thus more optimization in scheduler is needed.
- `OOM` - Out-Of-Memory killer kills a process if OS runs out of memory. While some OSes prefer to stop all processes and panic, others simply crash, Linux kills processes using OOM-killer since 2.4.
- `OS` - Operating System
- `PF` - Packet Filter, an engine that evaluates set of rules on each packet it receives and choses either to pass the packet further or to drop or reject it.
- `PGD` - Page Global Directory, i.e. the first of three levels of table structures used in memory pagination.
- `PIC` - Programmable Interrupt Controller
- `PMD` - Page Middle Directory
- `POSIX` - Portable Operating System Interface. The X signifies the Unix heritage of the API.
- `PPTP` - Point-to-Point-Tunneling-Protocol used for creating VPNs
- `qdisc` - Queueing discipline for Linux network traffic shaping

- `RAM` - Random Access Memory, type of physical memory that can be read from and written to.
- `RSS` - Resident Set Size - The part of the process' virtual memory that is currently in RAM, i.e. not swapped out.
- `RT` - Real Time, computing with dead-line constraints to response
- `RTOS` - Real Time Operating System
- `RW` - Read/Write
- `SCP` - Secure Copy Protocol, substitute for rcp implemented in OpenSSH
- `SFTP` - Secure FTP, substitute for ftp implemented in OpenSSH
- `SMP` - Symmetric Multi-Processing, i.e. multi-processing in which no memory is *closer* to one processor that to other as in NUMA.
- `SSH` - Secure SHell
- `syscall` - A way to call certain function implemented in kernel.
- `TCP` - Transmission Control Protocol
- `TID` - Thread ID
- `UDP` - User Datagram Protocol
- `UP` - Uniprocessing, architecture or machine with only one processor (one processor core)
- `VFS` - Virtual File System
- `VM` - Virtual Memory
- `VPN` - Virtual Private Network
- `WebDAV` - Web-based Distributed Authoring and Versioning, an extended HTTP protocol that allows users to collaboratively edit and manage files.
- `x86` - Processor architecure/family compatible with Intel 8086 processor

Abstract in Slovak Language

Práca sa zaoberá vnútorným fungovaním jadra operačného systému Linux verzie 2.6. Snaží sa zdôrazniť rozdiely oproti jadru 2.4 a uviesť do problematiky aj človeka znalého iba všeobecných princípov operačných systémov.

Popísaný je prehľad subsystémov jadra, hlbšie rozobraté sú dôležité stavebné kamene jadra ako napr. rôzne zámky, semafory a iné synchronizačné mechanizmy, vhodné dátové štruktúry, obsluha prerušení, atomické operácie, rozhranie novej architektúry ovládačov zariadení a alokácia pamäte.

Veľmi podrobne preskúmaná je aj správa procesov a nový plánovač s dobrou podporou viacprocesorových a NUMA architektúr a preemptívneho prepínania procesov. Jadro práce uzatvára popis tvorby jadrových modulov aj s ukážkovým modulom.