Architecture of the Linux kernel

Gerion Entrup Felix Herrmann Sophie Matter Elias Entrup
Matthias Jakob Jan Eberhardt Mathias Casselt

March 21, 2018

Contents

1 Introduction

On 26. August 1991 Linus Torvalds wrote in the Usenet-Newsgroup *comp.os.minix* the following message:

> Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).
>
> I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)
>
> Linus (torv...@kruuna.helsinki.fi)
>
> PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

This was the first public announcement of the Linux operating system, that is nowadays one of the biggest operating systems in the world.

The most exiting part of Linux is that it is Free Software. It is licensed under the GNU GPL and developed in such a way that everybody can study the internals of Linux, dive into the code, improve it and contribute it back. This has formed a huge community around Linux and leads to a very unique software development model – conditioned by the huge code base and amount of different developers.

This book focuses on the internals of Linux. It will present the functionality of the different subsystems and also give a quick overview about the development process.

1.1 Overview of Linux components

Linux is divided in several subsystems that handle one operating system specific task. Figure 1.1 presents an overview about the most common subsystems.

System call interface To connect with the userland, Linux uses the concept of system calls. They are defined as C-functions, that internally translate to specific assembler calls, to bring the processor in the supervisor mode. The system call interface is actually not a subsystem, but part of the whole kernel.

Process management (PM) A core concept of operating systems is the handling of processes. This subsystem defines processes and threads and how they interact with each other. The CPU is a limited resource, so a scheduler is needed to distribute it.

With the technique of interrupts, Linux can react as soon as possible to events. This comes with the cost of context switches and therefore needs some synchronization mechanisms.

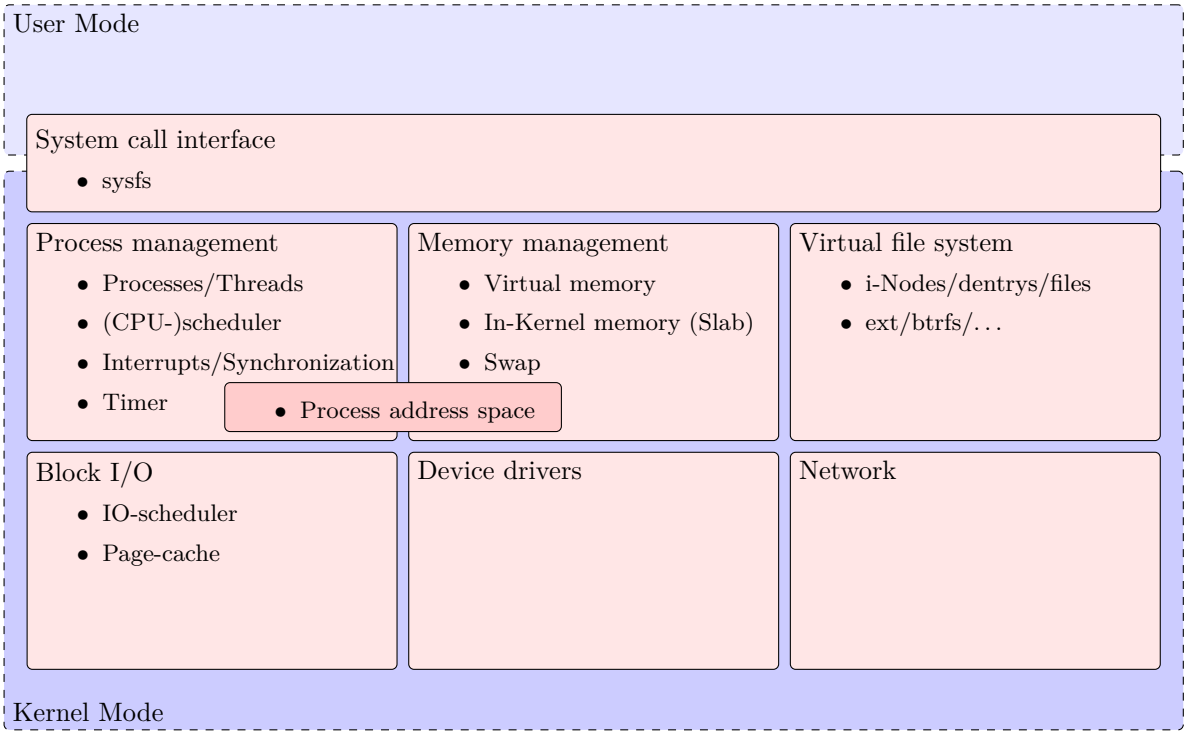


Figure 1.1: Overview of Linux subsystems

Memory management (MM) Processes and the kernel itself need memory to work. But memory is also limited and processes are not always friendly. They can be malicious per design or run amok because of bugs.

So the kernel steps in and defines a mapping of memory and isolates memory per process.

Virtual file system (VFS) A hard drive is just a big heap of blocks, where data can be stored. To access this data, some structure is necessary, also called file system. A lot of file systems have been invented and Linux supports lots of them. To prevent file system specific userland programs, the kernel has an abstract layer that provides the same interface for all file systems: The virtual file system.

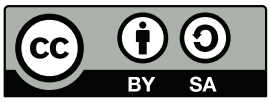
Block I/O (BIO) Hard drives are randomly accessible in theory. But in reality, this comes with certain costs. Most of the time successive blocks are faster accessible than random blocks. So the kernel tries to collect accesses to block devices, cache already accessed data and change its ordering.

Device drivers (DD) A good operating system supports a lot of hardware. Users want to buy several WLAN chips, TV cards, Bluetooth devices, etc. To support this, Linux defines an interface for device drivers that allows to write specific code for specific hardware, but also to have a generic way to work with them.

Network Access to networks is an important part of computers, but does not necessarily belong into the kernel. Nevertheless, it is part of the kernel, because this is the only location where network handling is fast enough. Network handling will not be covered in this book.

1.2 License

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.



2 Process Management

Process management is one of the key purposes of an operating system. Therefore it is crucial for the operating system to use appropriate algorithms and data structures for process management. Especially tasks like process creation, process termination and switching tasks while scheduling multiple processes on the system are substantial. However, the overhead of process management and scheduling needs to be minimized and respective tasks need to run fast. The operating system provides a transparent abstraction layer between hardware and programs, i.e. each process has its own virtual processor and virtual memory. Subsequently, from one process' point of view, it is the only one that owns the CPU; the execution of multiple processes on the system and switching between them happens transparent to the single process. This abstraction layer ensures, that each process has a controlled access to the hardware while it is isolated from other processes.

In the following section, we will look at how the basic concepts of process management are implemented in Linux. The information in this chapter is based on [3] and updated to fit the current Linux kernel 4.13.

2.1 Processes

A process is a program in execution. It does not only consist of the program's code, but of every resource necessary for execution. This includes memory address space, processor state, open files which the process accesses, signals and internal kernel data. Nevertheless, a running program is not limited to a single process, as there can be multiple processes and threads executing the same program.

Process descriptor In Linux, all information about a process is stored in the process descriptor defined as struct `task_struct`[1]. As every information about each process is contained within a corresponding `task_struct`, these are quite big data structures (the definition in *include/linux/sched.h* including comments is about 600 lines long). The kernel manages these structs in a doubly linked list, the `task_list`.

For accessing the process descriptor (`task_struct`) of the currently running process, one can use the `current` macro. In the x86 architecture (and most others), this macro expands to `get_current()` (see *arch/x86/include/asm/current.h*), which returns a pointer to the current `task_struct`. In multicpu or multicore systems, each CPU has its own reference to the `task_struct` of the process currently running on the respective CPU.

The most important fields of `task_struct` are shown in listing (2.1). The first entry holds the task's state, which is, in most cases, one of runnable, unrunnable and stopped. A list of all possible task states can be found in *include/linux/sched.h*. Information about which CPUs the process is allowed to run on and which it is currently assigned to are also stored in the `task_struct` (`cpu`, `cpus_allowed`). Each process descriptor contains information required by the scheduler. What these comprise and how they are used by the scheduler is discussed in the next chapter (chapter 3). For identification, each process has a system-wide unique process id (`pid`). As described later, Linux processes live in a hierarchy of parents and children. Thus, each process knows its parent and also has a list of its children (`real_parent`, `children`). Lastly, there are entries, which contain information about the file system and opened files (`fs`, `files`).

[1] Linux process are called task internally, thus both terms can be used, and the process descriptor is called `task_struct`

```
struct task_struct {
    ...
    // -1 unrunnable, 0 runnable, >0 stopped:
    volatile long               state;

    // Current CPU:
    unsigned int                cpu;
    cpumask_t                   cpus_allowed;

    const struct sched_class    *sched_class;
    struct sched_entity         se;
    int                         prio;

    pid_t                       pid;

    // Real parent process:
    struct task_struct __rcu    *real_parent;
    struct list_head            children;

    // Filesystem information:
    struct fs_struct            *fs;
    // Open file information:
    struct files_struct         *files;
    ...
}
```

Listing 2.1: Most important entries of the `task_struct` structure

2.2 Process creation - Forking a process

In Linux, as in Unix, process creation is done via the `fork()`- or `clone()` system call (or short: syscall). It is used to copy an existing process and then alter it for the purpose of the newly created process. The forked process is called the child of the original process, which is also labeled parent. Consequently, when a Linux system runs and processes are created, a process hierarchy is build. Herein, each process has exactly one parent, except for the init process, which is the very first process created when the Linux kernel boots.

While forking a task, only a minimal set of information in the new `task_struct` is altered, such as the `pid` (process id), the `ppid` (parents `pid`) and other resources, that will not be inherited (mainly statistical information). After `fork()` finishes, `exec()` can be called to load a new executable for the task and to execute it. Otherwise, the forked process executes the same program as its parent. A well known example is the Apache web-server. When a new connection has to be established, it forks itself. Even though the new process also executes code of the web-server, it is a different process responsible for the new connection.

In many cases, it is not useful to copy a whole task when creating it. If the new task executes a different program (by calling `exec()`), it would be a waste to duplicate all resources of the parent task. To avoid this, a technique called copy-on-write is used for creating new tasks. Using this approach, when a task is forked, the resources of the parent task will not be copied instantly. Only when the child task alters (i.e. writes) a part of the resources, a copy of that part is made. Hence, both tasks have separate data. However before that, the resources are shared read-only between the two tasks. So the actual copy process is divided and delayed and can be omitted if it is not necessary. This is done by the use of copy-on-write memory pages.

When a process makes a `clone()` syscall, `_do_fork()` is executed due to system call definitions of clone in `kernel/fork.c`. `_do_fork()` is the main function which actually forks a task. It is also defined in `kernel/fork.c`, together with its relevant sub-functions. The interesting part of copying a task is done in `copy_process()`:

- After mandatory error handling, `dup_task_struct()` is called, which, as its name says, duplicates the parents `task_struct` (see listing 2.2). Except for a few flags, the new `task_struct` is identical to the old one. Thereafter, it is checked whether the user of the parent task has not exceeded the number of processes allowed.

- In the progressing execution of `copy_process()`, the statistical information in the new `task_struct` is reset and various fields are set to initial values. Two noteworthy flags which are reset are the `PF_SUPERPRIV` flag, which indicates whether a task has super user privileges, and the `PF_FORKNOEXEC` flag, which indicates that the process has not called `exec()` (listing 2.3). However, the bulk of `task_struct` remains the same.

- Eventually, scheduler specific setup is done in `sched_fork()` (listing 2.4). In particular, the new task is assigned to a CPU and the task's priority is set to the default value. Also, the task's state is set to `TASK_NEW` so the scheduler will not place it in a runqueue yet.

- After that, dependent on the `clone_flags` passed to `copy_process()`, the parent task's resources will be shared or copied so the child task can use it (listing 2.5). Cloneable resources include the address space of a process, open files and signal handlers. A complete list of all clone flags shows all cloneable resources in *include/uapi/linux/sched.h*.

- With `alloc_pid()`, the new task gets its unique process id.

- At the end, `copy_process()` returns a pointer to the `task_struct` of the newly created task.

After the task is copied, `_do_fork()` wakes up the new task with `wake_up_new_task()`.

```
struct task_struct *tsk;
unsigned long *stack;
tsk = alloc_task_struct_node(node);
...
stack = alloc_thread_stack_node(tsk, node);
...
err = arch_dup_task_struct(tsk, orig);
tsk->stack = stack;
...
return tsk;
```

Listing 2.2: Important parts of `dup_task_struct()`

```
if (atomic_read(&p->real_cred->user->processes) >=
        task_rlimit(p, RLIMIT_NPROC)) {
    if (p->real_cred->user != INIT_USER &&
        !capable(CAP_SYS_RESOURCE) && !capable(CAP_SYS_ADMIN))
        goto bad_fork_free;
}
...
p->flags &= ~(PF_SUPERPRIV | PF_WQ_WORKER | PF_IDLE);
p->flags |= PF_FORKNOEXEC;
```

Listing 2.3: In `copy_process()`: Evaluate user's process limit and (re)set flags

```
/* Perform scheduler related setup. Assign this task to a CPU. */
int sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    unsigned long flags;
    int cpu = get_cpu();

    __sched_fork(clone_flags, p);
    /*
     * We mark the process as NEW here. This guarantees that
     * nobody will actually run it, and a signal or other external
     * event cannot wake it up and insert it on the runqueue either.
     */
    p->state = TASK_NEW;

    /*
     * Make sure we do not leak PI boosting priority to the child.
     */
    p->prio = current->normal_prio;
    ...
}
```

Listing 2.4: Scheduler related setup in `sched_fork()`

```
/* copy all the process information */
shm_init_task(p);
retval = security_task_alloc(p, clone_flags);
if (retval)
    goto bad_fork_cleanup_audit;
retval = copy_semundo(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_security;
retval = copy_files(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_semundo;
retval = copy_fs(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_files;
retval = copy_sighand(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_fs;
retval = copy_signal(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_sighand;
retval = copy_mm(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_signal;
retval = copy_namespaces(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_mm;
retval = copy_io(clone_flags, p);
if (retval)
if (retval)
    goto bad_fork_cleanup_namespaces;
retval = copy_thread_tls(clone_flags, stack_start, stack_size, p, tls);
if (retval)
    goto bad_fork_cleanup_io;
```

Listing 2.5: In `copy_process()`, rescources are cloned, depending on `clone_flags`

2.3 Threads

As described in the last section, process creation in Linux is done via the `clone()` system call. In comparison to processes, threads do not have a clearly distinguished concept in Linux. Other operating systems usually implement threads as lightweight processes. Herein threads use the concept of quickly creatable execution units in contrast to normal "heavy" processes which require more time to create.

This is not the case in Linux. The creation of a thread and a process does not differ substantially. Rather, a thread is a process which shares different resources with another process. Mainly, the address space of the parent task is shared with its threads. Also, task specific file system information, open files and signal handlers are shared additionally.

This is implemented in Linux by calling `clone()` with specific flags, so called `clone_flags`, which determine the resources to be shared. Therefore, to create a thread in Linux the clone syscall is called with the following flags:

```
1 clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

3 Scheduling in Linux

In this chapter the process scheduler of Linux will be described. Scheduling is an important part of an operating system, as it is needed to run multiple processes on a computer. On a running Linux system dozens of processes (programs in execution) exist. Although not all processes are always ready to run, there are at least some processes which are concurrently in a runnable state. On each execution unit (CPU core) however, only one process can run at the same time. The scheduler's task is to manage the running and waiting processes, so that simultaneous execution of processes is possible. Therefore, to provide (pseudo-)multitasking, the scheduler regularly preempts the currently running process and chooses another runnable one to proceed with. The time one process runs, before it is preempted, varies depending on the process's priority and the scheduling strategy. However, it is so short, that to a human computer user the system seems to execute multiple programs simultaneously.

After a brief discussion of scheduling classes and process categories, the problems of timeslice based schedulers are shown. The Completely Fair Scheduler (CFS), the current default scheduler in Linux, aims to solve these problems and is presented in the last section of this chapter.

3.1 Scheduling-Classes

There are different scheduling classes for tasks in the Linux kernel. Each scheduling class has its own scheduler. The three most important scheduling classes are

`SCHED_NORMAL` for normal processes (most of the processes belong to this category)

`SCHED_FIFO` for real-time processes

`SCHED_RR` for real-time processes with timeslices. RR stands for round robin.

Real time tasks, which are categorized as `SCHED_FIFO` or `SCHED_RR`, always have a higher priority than normal tasks. Tasks classified as `SCHED_FIFO` will run until they exit or a task with a higher priority becomes runnable and preempts it. In comparison, tasks of the class `SCHED_RR` are assigned timeslices in order for them to be preempted periodically. These tasks are scheduled round robin.

In the next sections, the focus of discussion will lie on the scheduler for the tasks of class `SCHED_NORMAL` (most of the tasks).

3.2 Process Categories

Generally, one can divide processes into two categories, namely I/O-bound processes and CPU-bound processes.

I/O-bound processes Processes of the first category spend most of their time waiting for data from an external source. This includes almost all devices with an input/output rate significantly slower than the CPU speed, for example the hard disk, network devices or the keyboard. Programs with a graphical user interface (GUI) are mostly I/O-bound since they often have to wait for the user to input data. These processes often do not need much CPU time. However, when the awaited data arrives and the process becomes runnable it is required to respond quickly to the incoming data (especially in a GUI).

CPU-bound processes In contrast to I/O-bound processes, CPU-bound processes do not spend much of their time waiting for input data. Rather, they do comprehensive computations on data, therefore needing considerable CPU time.

However, programs can be both I/O-bound and CPU-bound. There is no hard boundary but a seamless transition between the two categories. Sometimes a process starts I/O-bound, often waiting for user inputs. Then, when the user starts a bigger task, such as a conversion of a big video file, it becomes a CPU-bound process.

Processes of different categories have contrary needs and demands towards the process scheduler. I/O-bound processes are required to respond to incoming data as quickly as possible. Therefore, they need to be scheduled very soon after they become runnable. CPU-bound processes on the other hand do not necessarily need to be scheduled quickly but rather for a longer period of time, as to assure high CPU utilization.

3.3 $\mathcal{O}(1)$-Scheduler

The $\mathcal{O}(1)$-Scheduler was introduced in the Linux kernel 2.5. Its name is derived from its algorithmic behavior. Important scheduling operations, such as enqueuing and dequeuing of tasks in run queues, are done in constant time with respect to the number of tasks in the system. It performs well on server platforms and scales well on systems with many processes that want to run concurrently. However, on highly interactive systems such as desktop environments the old $\mathcal{O}(1)$-Scheduler has problems to schedule different kind of tasks. Especially, if background tasks and processes in a GUI that need low reaction times run together on a system, the concept of the $\mathcal{O}(1)$-Scheduler has its drawbacks.

A closer look is taken at different process categories, and general concepts of process scheduling various difficulties, which need to be addressed, can be found. The Completely Fair Scheduler (CFS), which is discussed afterwards, solves many problems of the $\mathcal{O}(1)$-Scheduler by implementing a conceptionally new scheduling algorithm.

3.4 Priorities and Timeslices

In Linux each process has a priority. Normal processes' priorities are represented by nice values and can be altered by the user. A task's default nice value is zero, but values in the range of -20 to 19 are possible. The higher a task's nice value is, the lower is its priority. Tasks with a high nice value are "nice" to other tasks and will be preempted more often in favor of higher prioritized tasks with a smaller nice value.

Timeslices A CPU can execute only one process at a time. In order to simulate real multitasking on a CPU running many processes concurrently, each of the processes can use the CPU only for a short amount of time before another process in line supersedes it. The amount of time each process receives is called timeslice.

The $\mathcal{O}(1)$-Scheduler was required to assign concrete timeslices to task priorities and to define a default timeslice for tasks. This leads to different kinds of problems. If the default timeslice is too long, the interactiveness of the system is affected. On the other hand, if it is too short, the scheduling overhead becomes too big and consequently the CPU utilization is bad (scheduling takes time, which can not be used to run actual programs). The following examples shows the fact that a fixed timeslice priority assignment can lead to unintended circumstances.

It is assumed, that the scheduler assigns timeslices of 100ms to processes with default priority (nice value 0) and timeslices of 5ms to processes with the least priority (nice value 19). Process priorities between 0 and 19 are also mapped to fixed timeslices.

Example 3.1 *Consider two runnable tasks A and B. The priority of task A is the same as of task B; both processes have a nice value of 19 and get equally small timeslices of 5*ms *runtime.*

Example 3.2 *The first process (A) has the default priority, hence its nice value is 0 and timeslices of 100*ms *are assigned to it. The other process (B) has a nice value of 19 and therefore a lower priority. When both processes become runnable, process A runs 100*ms *and is then preempted in favor of process B, which runs for 5*ms.

Example 3.3 *Consider four tasks A, B, C, D with following priorities:*

- *Task A: nice 0 (timeslices of 100*ms*)*
- *Task B: nice 1 (timeslices of 95*ms*)*
- *Task C: nice 18 (timeslices of 10*ms*)*
- *Task D: nice 19 (timeslices of 5*ms*)*

There are various problems with fixed priority-timeslice assignments. In the first example (3.1), the scheduling overhead is rather high, because of many context switches in short time. The CPU is divided 50/50 between the processes, but with different frequencies of context switches depending on the priority. When there are two runnable processes with the same priority, it is not meaningful to switch between them with high frequency, due to unnecessary scheduling overhead.

The scheduling of the two processes might be not ideal in the second example (3.2) either. Considering A to be a high priority I/O-bound process and B a low priority but CPU intensive background process, the first situation might not fit the needs of the processes. Process A would need to be scheduled quickly after it wakes up, yet needs the processor only for a short amount of time. Conversely, B does not have a need for quick response-time but rather long timeslices. In this case, the allotment of timeslices is the opposite of an ideal one.

The third example (3.3) shows another problem. Tasks A and B are only slightly different, both of priority and assigned timeslices. However, tasks C and D also differ little in priority, but there is a big relative difference of timeslice-size between them. Task C receives twice the processor time as task D, although their priorities are nearly the same. Therefore, the relative difference between assigned timeslices depends on a task's location in the range of possible priorities.

3.5 Completely Fair Scheduler

Linux aims to provide good interactive responsiveness of processes, thus favoring I/O-bound processes over CPU-bound processes. The current process scheduler, the Completely Fair Scheduler (CFS), for the default scheduling class (`SCHED_NORMAL`) uses a creative approach to not neglect CPU-bound processes at the same time. It replaced the $\mathcal{O}(1)$-Scheduler as the default scheduler in the Linux kernel 2.6.23. Its main approach is to provide an approximately fixed fairness among processes and variable timeslices.

A system of perfect multitasking is modeled which aims to give each process the same or fair amount of the processor. For n processes of the same priority each process gets $1/n$th of the processor time. CFS does not assign specific timeslices to priorities (nice values) but rather uses the nice value to weight the proportion of the processor a process should get. To achieve that, CFS keeps track of the processes' runtimes and weights them by their priorities. When CFS preempts a running process it schedules the process which has been run the least so far next.

3.5.1 Implementation details

The Completely Fair Scheduler defines small time windows in the configuration option `sched_latency_ns`. These are then divided into equal timeslices which are assigned to the runnable processes. Whenever the number of runnable processes changes, the length of the timeslices must be adjusted.

One of the main objectives of operating systems is to execute processes. Hence the scheduling, i.e. preempting of processes, time accounting, selection of a process to run and so forth, produces a management overhead. As described before, CFS tries to provide each equally prioritized process with the same amount of processor time. The more processes are runnable, the less CPU time CFS permits each of them in a time window (*scheduling latency*). In consequence, CFS must schedule the processes more often. Therefore, if there are too many runnable processes, this behavior would lead to a crucial performance decrease due to increased scheduling overhead. To avoid a large overhead, making scheduling inefficient, CFS sets a lower bound to the timeslice length it assigns to a process. This lower bound is called *minimum granularity* and is the minimal amount of time a process runs on the CPU before it can be preempted. The default value lies between 1 to 4 ms depending on the Linux distribution. It is one of the few configuration options of CFS and can be adjusted by changing `/proc/sys/kernel/sched_min_granularity_ns`.

CFS's approach leads to a more meaningful allotment of timeslices with respect to the process priorities. Depending on their nice values, each runnable process gets a relative timeslice. For example, consider two processes (A and B) with different nice values. Process A has the default nice value (zero), while B's nice value is 5 (lower priority). In this case, B would receive about 1/3 of the targeted latency and A the rest of the time. Assuming the targeted latency to be 20ms, B would get a 5ms time slot and A 15ms on the CPU. If A and B's nice values change to 10 and 15, the relative priority difference would still be the same. Hence, the allotted timeslices would stay 15ms and 5ms. To conclude, CFS does not care about the absolute priority difference between processes, but takes the relative difference into account. Also the timeslices, CFS assigns, are not fixed or bound to concrete priority values. Instead, before CFS schedules a process, it will calculate a dynamic timeslice for it depending on the number of currently runnable processes. The weighting of processes is related to their priorities and is done by using the concept of virtual runtime. Simply spoken, the runtime of low priority processes will elapse faster and respectively slower for high priority processes. This is implemented by managing the `vruntime` of a process as described in the next paragraph.

vruntime In order to keep track of all processes runtimes, the field `vruntime` is updated by the scheduler. `vruntime` contains the runtime of a process weighted by its priority. It is part of the `sched_entity` struct, which itself is referenced in the `task_struct` of a process.

The value of `vruntime` of the currently running process must be updated regularly. This task is done by the `update_curr` function (listing 3.1), which is invoked periodically by the system timer and on events regarding the processes state, accordingly when the process becomes runnable or blocks. The main purpose of `update_curr` is to measure the runtime of the current process since the last call of `update_curr` and update the weighted `vruntime` according to the priority of the process. This is done by adding the result of `calc_delta_fair` to `curr->vruntime`. In most cases, a process has the default priority (nice value 0). Thus, its physical runtime (the time a process spend on the CPU) equals its virtual `vruntime`. If the process priority is higher, its runtime is weighted less and the amount added to `vruntime` is lower than the actual runtime. Therefore, in fact the process gets more time on the CPU. Respectively, the added time to `vruntime` is greater than the actual, if the priority is lower.

Furthermore, the minimal runtime of all processes in a runqueue (`cfs_rq->min_vruntime`) is updated. The scheduler needs this information to keep the runqueue (sorted by `vruntime`) up-to-date.

Runqueue implemented as red-black tree The runqueue CFS uses is not an actual queue but a red-black-tree. These are binary trees with a self-balancing property, meaning all paths in the tree are nearly of the same length (one path can never be twice as long as any other path). This guarantees that operations on the tree (such as insert, delete or search) can be made in $\mathcal{O}(log(n))$. To manage runnable processes, CFS

```
//Update the current task's runtime statistics.
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;
    ...

    delta_exec = now - curr->exec_start;
    curr->exec_start = now;
    ...
    //statistics
    schedstat_set(curr->statistics.exec_max,
    max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);

    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);
    ...
}
```

Listing 3.1: `update_curr()`. Function to account (virtual) runtimes of processes

stores them in a time-ordered red-black tree. Referenced by the left-most node in the tree is the process with the least `vruntime`. When a running process is preempted and another waiting, runnable process must be selected, CFS just picks the process associated with the left-most tree-node. To provide a faster access to the next process, the left-most node is always cached. The process which was just preempted and is still runnable is reinserted in the red-black tree with its updated `vruntime`. Because it was the last running process, it is inserted into the right side of the tree. An example of such a red-black-tree is presented in figure 3.1.

Balancing of I/O-bound and CPU-bound processes Processes waiting for data are blocked until the requested data is available. During this time, they are not runnable and hence are no longer in the red-black tree. Thus, these processes will not be considered when CFS assigns runtime to the runnable processes. When a waiting process (A) becomes unblocked eventually, i.e. becomes runnable, the runtime assignment must be redone because more processes are runnable now. Since the other runnable processes were able to run while process A was blocked, their `vruntime` values are non-zero. In contrast, the newly unblocked process A has not run since it was blocked, so its `vruntime` is zero. Therefore, its `vruntime` is the smallest and it is inserted into the left-most position of the red-black tree. Consequently, when the scheduler picks a new process it will be chosen next. Hence, the reaction to the now available data is very fast.

3.6 Kernel Preemption

Since Linux version 2.6, the kernel is fully preemptive. Therefore processes in kernel context and kernel threads can be preempted at any time it is safe to do so. To preempt a task in kernel mode, for example when it executes a system call, it must not hold locks. Otherwise it would not be safe. Locks are used to mark regions that must not be preempted. The kernel keeps track of a task's locks with the `preempt_count` variable. Initialized to zero, it is incremented when a lock is taken and decremented when the task releases

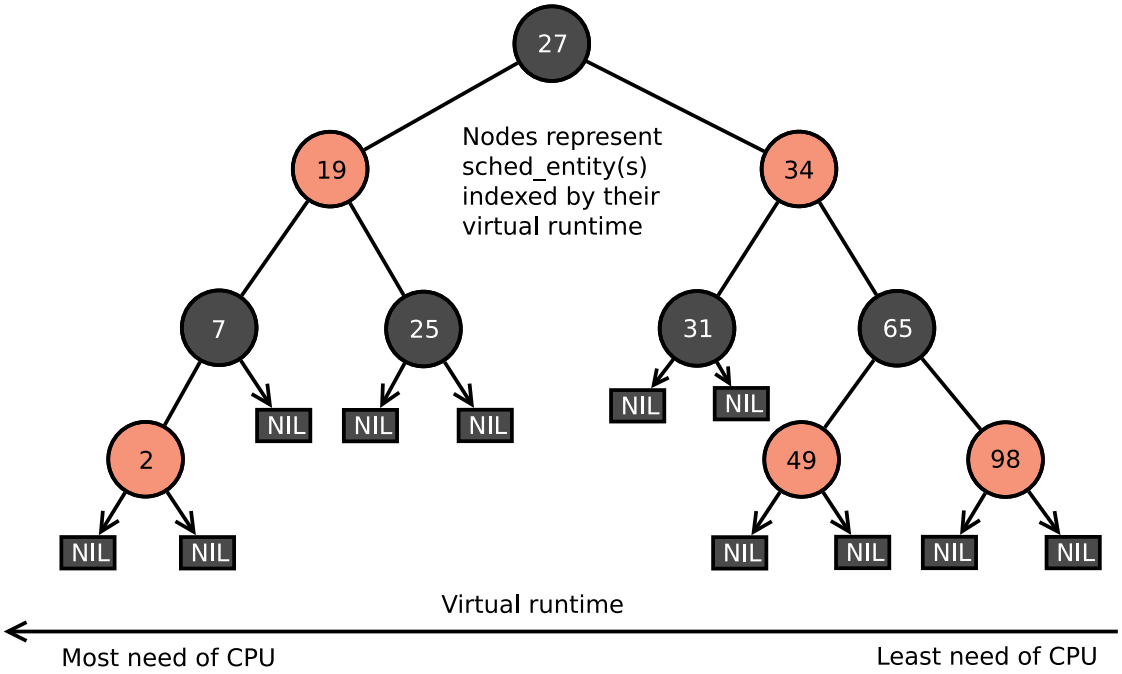


Figure 3.1: Example of runnable processes in a red-black tree of CFS[1]

a lock. Only if a task's `preempt_count` is zero and a higher prioritized task is runnable, the current task can be preempted by it. This could be the case if upon returning from an interrupt to kernel-space, the `need_resched` flag[2] of the current task is set (due to the existence of another higher prioritized process).

[1] Adopted from [2]

[2] The `need_resched` field of the current process is set whenever it has to be preempted, so the scheduler is invoked when the timer interrupt handler terminates (see section 8.3).

4 Interrupts

The typical task of a processor is to process a predefined series of steps (a program). To notify the processor of some event, sometimes an interruption of the task currently being processed is wanted. Interrupts can be triggered by the program or asynchronously from an external source. If an *Interrupt Request, IRQ* occurs, the processor executes a predefined *Interrupt service routine, ISR*.

Software errors, such as division by zero, or explicit interrupt calls (e.g. syscalls) can be handled by the current CPU. Hardware interrupts, such as "data available", are routed by an interrupt controller to a specified processor (IO-APIC on current x86 systems).

Example 4.1 *Every time a button gets pressed on your keyboard, the ISR in the kernel has to decide what to do, probably send an event to userland, where a client can consume it.*

4.1 Interrupt processing

When an interrupt gets triggered, the processor executes the ISR, while normal scheduling is halted. To keep the system as responsive as possible, most of the interrupt handling should be done while scheduling is active. Therefore, the processing of interrupts is usually divided in two halves:

- The *Top Halve* is the immediate code executed after entering the interrupt context through the ISR. While the processor executes this part, only high priority interrupts can preempt the execution. Interrupt priorities and masking are architecture specific and usually get managed by a programmable controller. All processing that does not need to be done immediately gets scheduled as a *Bottom Halve*.
- In the *Bottom Halve* the rest of the interrupt processing takes place. The Linux kernel provides multiple mechanisms: Softirqs, tasklets and Work queues. (See chapter 5.)

4.2 Interrupt entry

Interrupts can be called at nearly every part of execution, which means that the handler cannot use every functionality available. User space and blocking calls, such as sleep are not usable, because the processor cannot switch into the interrupt context.

The *Interrupt Vector Table, IVT* is an array of ISR descriptors, with the IRQ number acting as an offset to map each handler to an interrupt.

Example 4.2 *On x86 systems the IVT is called the* Interrupt Descriptor Table (IDT)*, where an entry is called a* gate*. The CPU has a special register that points to the current IDT. It can be updated by the* "lidt [array]" *instruction. IRQ 1 could be the keyboard interrupt.*

4.3 Event handler

There are only 256 different IRQs so interrupt lines have to be shared between devices. Handlers for these *shared interrupts* have to check if they are responsible or return immediately (see `handle_IRQ_event()` inside `/kernel/irq/handler.c`).

Event handlers get registered by the `request_irq` function. Internally, the handler is packaged inside an action structure and added to a linked list inside an IRQ description structure, which exists for every line.

```
// Interrupt handler prototype       (IRQ, device_data *)
typedef irqreturn_t (*irq_handler_t)(int, void *);
// irqreturn_t == IRQ_HANDLED upon successful execution

// Register handler with the kernel
int request_irq(
    unsigned int irq,
    irq_handler_t handler,
    unsigned long flags, // e.g. shared
    const char *name, // for /proc/interrups
    void *dev)     // unique, usually device data
```

Listing 4.1: `request_irq`

Some subsystems, such as PCI *Message Signaled Interrupts*, differ from these mechanisms so it is recommended to first read the documentation of the subsystem where the interrupt should be triggered.

4.3.1 Example: Generic Top Halve

Almost all requests for data are built asynchronous where the data is requested, then some other code is executed and when the data is available, an interrupt is raised. Let's look at an example interrupt of a "generic data provider device".

1. The interrupt is raised.
2. The processor saves the current state and enters the ISR.
3. The kernel searches all handlers associated with the IRQ. (`do_IRQ()`)
4. Each handler checks if it is responsible. (`handle_IRQ_event()`)
5. The data is copied into a buffer.
6. The device is reset, in order to receive new data.
7. A bottom halve is scheduled to process the data.
8. The process context gets restored and execution continues. (`ret_from_intr()`)

4.3.2 Example: Handler

A concrete example of a handler function is the ACPI screen brightness handler in listing 4.2 (`/drivers/acpi/acpi_video.c`).

```
static void brightness_switch_event(struct acpi_video_device *video_device,
                    u32 event)
{
    /* check responsibility */
    if (!brightness_switch_enabled)
        return;

    /* Save data */
    video_device->switch_brightness_event = event;
    /* schedule bottom halve */
    schedule_delayed_work(&video_device->switch_brightness_work, HZ / 10);
}
```

Listing 4.2: brightness_switch_event

4.4 /proc/interrupts

The interrupts file inside the procfs provides statistics for all current IRQs and CPUs.

The first column shows the interrupt number or the symbol. Most non-numeric interrupts are architecture specific and handled separately. The *CPUX* columns show the occurrence of every interrupt per CPU. The last columns describe the interrupt type, e.g. edge-triggered interrupt over IO-APIC and the name from `request_irq()`.

```
           CPU0       CPU1
  0:         10          0     IO-APIC    2-edge      timer
  8:          0          1     IO-APIC    8-edge      rtc0
  9:     548601      27793     IO-APIC    9-fasteoi   acpi
 29:        738         22     IO-APIC   29-fasteoi   intel_sst_driver
 45:     149948     149956     IO-APIC   45-fasteoi   mmc0
168:          0          1    chv-gpio   53  rt5645
180:        122          8    chv-gpio    9  ACPI:Event
182:      28926        144    chv-gpio   11  i8042
183:     232034       7416    chv-gpio   12  ELAN0000:00
309:          0          0     PCI-MSI 458752-edge      PCIe PME, pciehp
310:          0          0     PCI-MSI 462848-edge      PCIe PME
311:        444         65     PCI-MSI 327680-edge      xhci_hcd
312:     956827      49628     PCI-MSI 32768-edge      i915
NMI:          0          0     Non-maskable interrupts
LOC:    3901481    3583168     Local timer interrupts
SPU:          0          0     Spurious interrupts
```

4.5 Conclusion

For every interrupt event, the kernel preempts the current process and executes the appropriate handler. This handler does the least amount of work possible and schedules the rest for later in the Bottom Halve (see chapter 5).

5 Bottom Halves

Linux has multiple Bottom Halve mechanisms, where work can be scheduled without worrying about the implementation. The structure describing the job has to be created and then passed to the scheduling functions.

The main Bottom Halve mechanisms are *work queue*s, *softirq*s and their simpler interface *tasklet*.

Softirq Softirqs are actions defined at compile time, that get triggered every time the `do_IRQ()` routine gets called. Blocking or sleep are not available because execution can and usually will happen inside interrupt context. With this method multiple instances of the same handler can run simultaneously, which makes locking inside the handler necessary.

Tasklet Tasklets are build on top of softirqs and can be defined at runtime. They are also easier to use because they are serialized with respect to itself which means the handler is not reentrant.

Work Queue Every work queue is a group of kernel threads, one per processor, which executes work tasks. Every functionality available to normal kernel threads, such as blocking and sleep, is also available in functions deferred through the work queue.

5.1 Selecting the correct Bottom Halve mechanism

If you are wondering which of the Bottom Halves you should use, the answer is usually tasklet or `work_queues`, if you need blocking.

The use of softirqs or the definition of special kernel threads is discouraged and should only be considered if it is the only possible way.

The decision diagram shown in Figure 5.1 can help selecting the correct Bottom Halve mechanism.

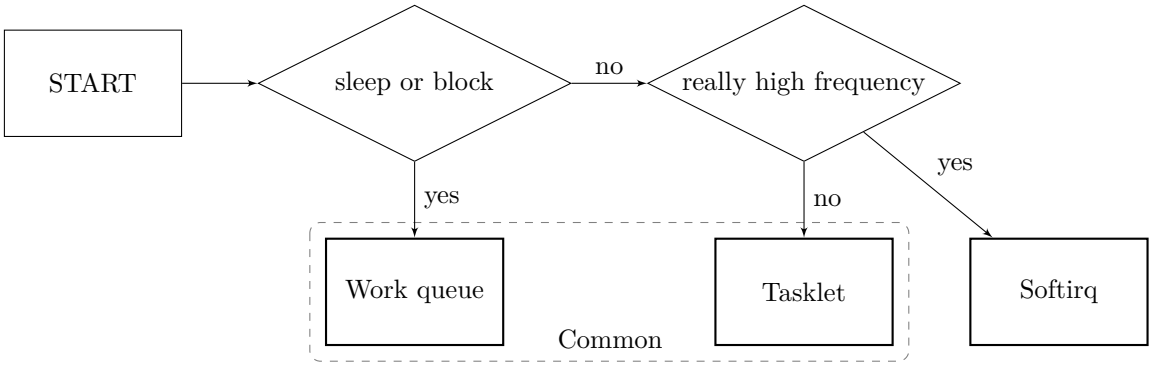


Figure 5.1: Which Bottom Halve should you use?

5.2 Preemption and Priorities

During the development of interrupt handlers and Bottom Halves, it is important to keep in mind what can and will be preempted in order to not lock up the machine. When a data structure is locked in both top and bottom halve and the interrupt handler spins while the bottom halve holds it, it will spin forever.

Figure 5.2 shows the order in which preemption occurs while interrupts are not deactivated.

The highest level is immediately after the interrupt context is invoked, where the handlers are the functions that get executed. Softirqs, such as tasklets, usually run before the kernel leaves the interrupt context and therefore at any point in the execution of the active task. Kernel threads, such as work queues, are scheduled as normal processes.

The *ksoftirqd* kernel threads are used to execute softirq in process context to reduce the time the processor spends in the interrupt context and to make the system more reactive.

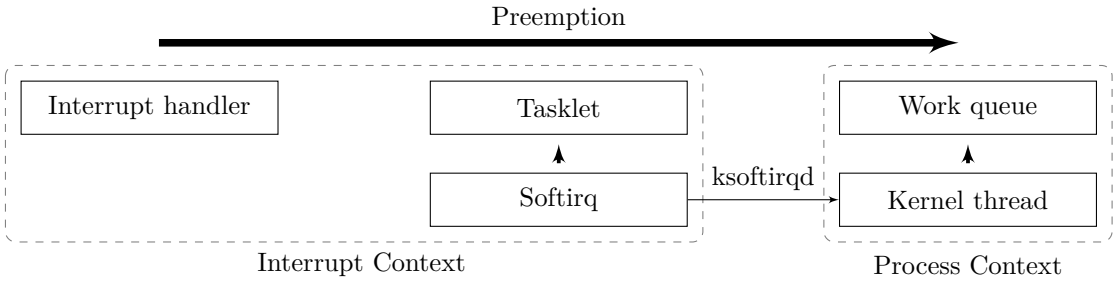


Figure 5.2: Preemption

5.3 Softirq

Each softirq action is executed every time the `do_softirq()` function is called, mainly after the interrupt handlers. If the execution exceeds a defined amount of time, remaining actions are transferred into the *ksoftirqd* thread, where they are scheduled like processes.

A function pointer can be associated with a softirq through `open_softirq()`, see Listing 5.1.

```
// Structure, so that it could be extended in the future
struct softirq_action
{
    void    (*action)(struct softirq_action *);
};

// One action per softirq number
extern void open_softirq(int nr, void (*action)(struct softirq_action *));
```

Listing 5.1: softirq_action

The nine softirq actions include the tasklet entry points HI and TASKLET, the NET networking subsystem, the SCHEDuler and the Read Copy Update system. Similar to the interrupt statistics, there exists a softirq file inside the procfs where every execution of a softirq is counted (See Listing 5.2).

```
$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3
          HI:      126689      98478     104566     101418
       TIMER:     7297990    6580197    6082010    5518867
      NET_TX:         337        273        392        365
      NET_RX:        2715       2817       2947       2675
       BLOCK:          61          0         13         49
    IRQ_POLL:           0          0          0          0
     TASKLET:     1307276     188466     188938     188711
       SCHED:     3507967    2579418    2457851    2006234
     HRTIMER:           0          0          0          0
         RCU:     2614582    2151239    2078531    1918210
```

Listing 5.2: SoftIRQ - procfs

5.4 Tasklet

Tasklets are a simple bottom halve mechanism to defer work in interrupt context. Tasklets and high priority tasklets have linked lists of `tasklet_struct`(Listing 5.3) that get executed through the softirq `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`.

They can be dynamically created through `tasklet_init()` and contain a function, data pointer and serialization fields. One type of tasklet only runs simultaneously on one processor, different tasklets can run concurrently. The `tasklet_schedule()` function schedules a tasklet for execution(lines 10-16 Listing 5.3).

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

// Create struct dynamically
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long),
                  unsigned long data);

// Schedule for execution
void tasklet_schedule(struct tasklet_struct *t);
```

Listing 5.3: tasklet_struct

5.5 Work queue

Work queues are another way to defer action inside the Linux kernel. The usage is similar to tasklets: A work struct gets created and attached to a linked list for scheduling. In contrast to tasklets, the function inside work queues is executed in a normal thread, where you can sleep, block and allocate as much data as necessary.

These threads are visible through the *ps* tool and are named `kworker/%u:%d%s (cpu, id, priority)`. Each work queue has one thread per CPU so that every `work_struct` gets executed on the processor it was scheduled on. Also similar to tasklets, work gets scheduled through a call to `work_schedule()`.

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
...
};

// Create work struct (MACRO)
#define INIT_WORK(_work, _func)                         \

// Schedule for execution
bool schedule_work(struct work_struct *work);
```

Listing 5.4: work_struct

5.6 Conclusion

Bottom Halves are the mechanism to defer actions inside the Linux kernel and used to execute parts of the interrupt processing without interrupts disabled. The mechanisms are softirq, tasklet and work queue. Tasklets are build on top of softirqs and are executed inside process context. Work queues are executed inside kernel threads.

6 Kernel Synchronization Methods

6.1 Why Is Synchronization Needed?

Kernel synchronization methods are very important when it comes to concurrent calculations and therefore are necessary for software projects, including the Linux kernel. Without the use of synchronization, data will become inconsistent and therefore wrong and might lead to instability of the system. To synchronize threads, different locks are used for limiting access to shared data. This will synchronize parallel computations, with the advantage that no calculations will output wrong results, because calculations were done on old, already updated data. This chapter will talk about standard methods, then present some more special methods, and at last discuss interrupts and decision-making. For a better understanding we will start with an example.

Synchronization is not only used in computer programs. In many cases it appears in common situations. Let's take the citizen office as example. When someone visit it to get a new ID card, he doesn't need an appointment. Instead, he pulls a ticket with a number and waits till the number is called out. But how is this comparable with a synchronization method? In most cases there are a few employees covering the daily business. For this example we will stick with three employees. Over time people arrive, the first one takes a number and is picked directly, so he can just walk in. When all three employees are busy, people have to wait. This is similar to a program where a resource is not available. The people are waiting in the lobby for a resource to become available. Programs do the same thing, for example wait for the network card to be idle, so they can send a HTTP request to their favorite website. You can have multiple network cards if you have a lot of request, like a citizens office would hire more employees, if citizens complain about long queues.

6.2 Standard Synchronization Methods

Atomic The atomic data types and operations are a valid method for synchronization. They allow to securely access and change a variable. There are two sets of operations, one for integer and one for bit wise based operations.

The integer operations operate on a special data type, called `atomic_t`(32 Bit) or `atomic64_t`(64 bit). These data types prevent the usage of standard C types with atomic functions and the usage of an atomic type with a standard C function. With this technique, it disables the unsynchronized use for the atomic types. The atomic datatype is a good way to secure an integer because of the mentioned limitations.

The bit wise operations do not have their own data type, so they can be used with every data type. Therefore, the variable can still be used with unsynchronized methods. As the programmer needs to ensure that it is always used in a synchronized way, it is not as safe as the integer counterpart. Still, they are a convenient way to safely access and change a data type, but notice that other developers might use another way to lock it, so document the decision. Implementations for atomic operations are often based on spin locks, but some CPU architectures might have build-in atomic instructions.

This example is from the x86 architecture. It shows the `atomic\_add`, which is written in inline assembler.

```
1 static __always_inline void atomic_add(int i, atomic_t *v)
  {
      asm volatile(LOCK_PREFIX "addl␣%1,%0"
      : "+m" (v->counter)
```

```
5       : "ir" (i));
}
```

Listing 6.1: linux/arch/x86/include/asm/atomic.h

Spin Lock The spin lock is a simple method for synchronization. It provides a structure to lock the access. To use it, an instance of the spin lock is required to be acquired. If the lock is already acquired, the process gets thrown into a spin and spins as long as the lock is not released. When the processor is aquired to spin, it executes no useful operation. If you can acquire the spin lock, you can continue the calculations. As spin locks let threads wait via spinning, it should never be locked for a long time, because this generates a processor load which does not calculate anything meaningful. If you need to lock for a longer time, it is better to use a semaphore.

This type definition is from the spin lock. This definition gives a hint about it's implementation. It uses the existing atomic to control the access to the lock.

```
1 typedef struct qspinlock {
      atomic_t    val;
  } arch_spinlock_t;
```

Listing 6.2: linux/include/asm-generic/qspinlock_types.h

Semaphore A semaphore works with another methods than the spin lock. Firstly, it does not let waiting threads spin. The threads are appended to a waiting queue and then put to sleep. When the lock is released, a thread from the waiting queue gets waked up. Secondly, big difference to the spin lock is that more than one thread can Hold it. You can configure, in the initialization, how many threads can be acquired simultaneously, by that semaphore. It seems like you could always use the semaphore, but you should not.

```
1 int down_trylock(struct semaphore *sem)
2 {
      unsigned long flags;
      int count;
      raw_spin_lock_irqsave(&sem->lock, flags);
      count = sem->count - 1;
7     if (likely(count >= 0))
      sem->count = count;
      raw_spin_unlock_irqrestore(&sem->lock, flags);
      return (count < 0);
  }
```

Listing 6.3: linux/kernel/locking/semaphore.c

If you lock for a short time, the spin lock works better because the overhead from putting the thread to sleep and waking it up again is not existing. If you want to allow just one thread, you should have a look at the mutex.

Mutex The mutex is "similar to a semaphore with a count of one, but it has a simpler interface, more efficient performance, and additional constraints" [3] Page 195. In comparison to a semaphore there are three contraints:

1. allows only one, instead of multiple threads, to acquire the lock
2. the mutex must be released in the same context

3. the process can not be exited while holding a mutex
4. does not allow recursive locks

Besides that, the kernel can check for violations of these constraints and warn about it.

This example shows the unlocking of a mutex.

```
void __sched mutex_unlock(struct mutex *lock)
{
    #ifndef CONFIG_DEBUG_LOCK_ALLOC
    if (__mutex_unlock_fast(lock))
        return;
    #endif
    __mutex_unlock_slowpath(lock, _RET_IP_);
}

static noinline void __sched __mutex_unlock_slowpath(
        struct mutex *lock, unsigned long ip)
{
    struct task_struct *next = NULL;
    DEFINE_WAKE_Q(wake_q);
    unsigned long owner;

    mutex_release(&lock->dep_map, 1, ip);

    /*
    * Release the lock before (potentially) taking the spinlock such that
    * other contenders can get on with things ASAP.
    *
    * Except when HANDOFF, in that case we must not clear the owner field,
    * but instead set it to the top waiter.
    */
    owner = atomic_long_read(&lock->owner);
    for (;;) {
        unsigned long old;

        #ifdef CONFIG_DEBUG_MUTEXES
        DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
        DEBUG_LOCKS_WARN_ON(owner & MUTEX_FLAG_PICKUP);
        #endif

        if (owner & MUTEX_FLAG_HANDOFF)
            break;

        old = atomic_long_cmpxchg_release(&lock->owner, owner,
        __owner_flags(owner));
        if (old == owner) {
            if (owner & MUTEX_FLAG_WAITERS)
                break;

            return;
        }

        owner = old;
    }
```

```
49
        spin_lock(&lock->wait_lock);
        debug_mutex_unlock(lock);
        if (!list_empty(&lock->wait_list)) {
            /* get the first entry from the wait-list: */
54          struct mutex_waiter *waiter =
            list_first_entry(&lock->wait_list,
            struct mutex_waiter, list);

            next = waiter->task;
59
            debug_mutex_wake_waiter(lock, waiter);
            wake_q_add(&wake_q, next);
        }

64      if (owner & MUTEX_FLAG_HANDOFF)
            __mutex_handoff(lock, next);

        spin_unlock(&lock->wait_lock);

69      wake_up_q(&wake_q);
    }
```

Listing 6.4: linux/kernel/locking/mutex.c

Read/Write Locks For better access to variables that are mostly read and written to, there are special locks for better performance. The read/write locks allow programs to read simultaneously with multiple threads and only need to wait if someone wants to write. Implementations as spin lock and semaphore are available. It is to be noticed that this lock favors a read operation, whereas a write operation needs to wait until no thread is locking it anymore. The problem is, anyone can start reading while the read lock is held and therefore they can starve the writer, if the read lock is reacquired before it is released. The example below shows that we acquire the read before we even check if someone is writing. It will be released if the lock is unavailable.

```
1   static inline void queued_read_lock(struct qrwlock *lock)
    {
        u32 cnts;

5       cnts = atomic_add_return_acquire(_QR_BIAS, &lock->cnts);
        if (likely(!(cnts & _QW_WMASK)))
            return;

        /* The slowpath will decrement the reader count, if necessary.*/
10      queued_read_lock_slowpath(lock);
    }
```

Listing 6.5: linux/include/asm-generic/qrwlock.h

```
1   void queued_read_lock_slowpath(struct qrwlock *lock)
    {
        /*
4       * Readers come here when they cannot get the lock without waiting
        */
```

```
    if (unlikely(in_interrupt())) {
        /*
        * Readers in interrupt context will get the lock immediately
        * if the writer is just waiting (not holding the lock yet),
        * so spin with ACQUIRE semantics until the lock is available
        * without waiting in the queue.
        */
        atomic_cond_read_acquire(&lock->cnts, !(VAL & _QW_LOCKED));
        return;
    }
    atomic_sub(_QR_BIAS, &lock->cnts);

    /*
    * Put the reader into the wait queue
    */
    arch_spin_lock(&lock->wait_lock);
    atomic_add(_QR_BIAS, &lock->cnts);

    /*
    * The ACQUIRE semantics of the following spinning code ensure
    * that accesses can't leak upwards out of our subsequent critical
    * section in the case that the lock is currently held for write.
    */
    atomic_cond_read_acquire(&lock->cnts, !(VAL & _QW_LOCKED));

    /*
    * Signal the next one in queue to become queue head
    */
    arch_spin_unlock(&lock->wait_lock);
}
```

Listing 6.6: `linux/kernel/locking/qrwlock.c`

6.3 Special Synchronization Methods

Completion Variable The completion variable is a simpler solution for some problems where you normally would have used a semaphore. The thread can wait for completion and when the variable is ready, the thread gets a signal. This signal wakes the thread up, so it can work with the completed variable.

Big Kernel Lock The Big Kernel Lock is an old, not used, lock method, which allows you a lot of things. First of all you can hold this when you want to sleep. It's a recursive lock. It can be used in an Interrupt context. But in general it works like a spin lock. The big problem is that it is a global lock and therefore not very friendly for concurrent calculations. After all, this lock is not used anymore and replaced by other locks.

Sequential Lock This lock has an additional variable, a sequential counter. The sequential counter is incremented whenever a write process acquires or releases the lock. The counter starts at 0, therefore when it is even, no process is trying to write. A read process needs to read the sequential counter before reading and compares it to itself after reading and its still the same, the value is valid. From this principal it is noticeable that writers are preferred. Below is an example of the usage of the sequential lock. The loop

starts with a read of the sequential counter and then a read of the value. The verification of the counter is in the while statement. As long as the counter is not same, the loop will be repeated.

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xtime_lock,seq));
    return ret;
}
```

Listing 6.7: queued_read_lock

6.4 Interaction with Interrupts, Bottom Halves and Kernel Preemption

Interrupts disabling When working in an interrupt handler routine and you need to use a lock, you can use a spin lock. When you acquire a spin lock, make sure to also disable interrupts or the interrupt appears again, trying to gain the same interrupt again and therefore will never get it and hang. For this purpose special functions are provided.

```
spin_lock_irqsave(&lock, flags);
spin_unlock_irqrestore(&lock, flags);
```

Listing 6.8: spin_lock_irqsave4

These functions disable the interrupts and save the flags which are enabled. With the unlock function, it will restore the old flags and by that enable the interrupts again. You should always use this in an interrupt handler, but be careful with using this in a normal program, it is not meant to be used for that.

Bottom Halves disabling The problem that the interrupts have also appears with the bottom halves, so there are some special functions as well.

```
spin_lock_bh();
spin_unlock_bh();
```

Listing 6.9: spin_lock_bh

Preemption disabling Preemption is an ability of the kernel to prioritize threads when needed. Therefore, the current thread gets rescheduled and the prioritized threads gets executed instead. If you want to prevent this, you can disable it, but this will affect real time constraints and should be used as short as possible.

6.5 Which one to choose

With the diagram 6.1 you can easily choose the lock method that fits your purpose. This only covers the standard methods, but you can acquire everything with them. Some questions might be a bit unclear. With short lock time the amount of instructions were meant, mostly if it is worth to make a thread sleep versus

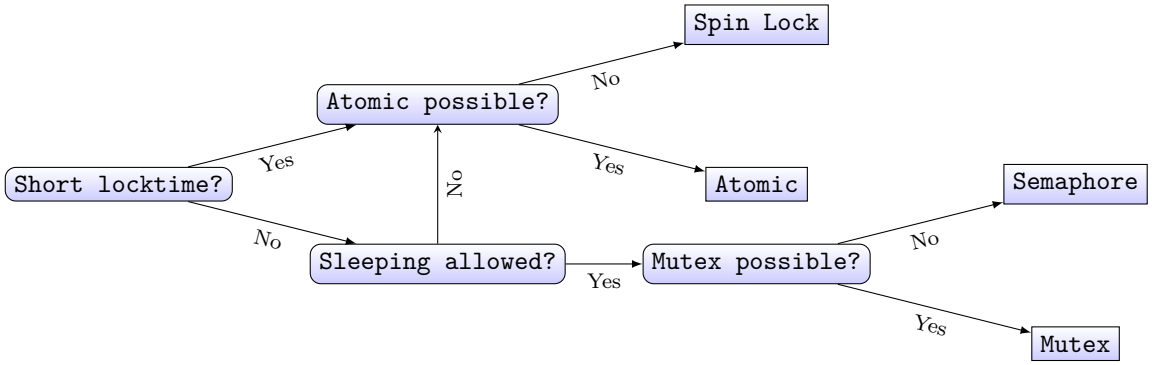


Figure 6.1: Decision diagram to choose a lock

if the time to spin is less. The Sleeping allowed node asks if the thread is already holding a spin lock or if it is in an interrupt context. This are situations where you would need a spin lock. The more special ones are nice if it fits your purpose and mostly can be exchanged for a semaphore or mutex.

6.6 Problems

Locks are used at many places inside of the kernel. Still, they have some disadvantages, which you should look out for. First of all, there is an overhead, this may not be critical for most applications, but for real time application it is hard to predict when you get a lock you are waiting for. Second, there is the problem of a deadlock. If you don't pay attention to the order you acquire the locks, it might end in a deadlock. To oppose this, you should document the order somewhere. It is a curcial to lock data and not code.

6.7 Conclusion

Locks are very useful if you want to work with multiple threads. This can prevent a lot of trouble with sharing data to multiple threads. If used right, it is a really powerful tool, but it can also lead to a deadlock.

7 System calls

Development of operating systems has a lot to do with isolation. An operating system can and should not trust any program that is executed on the processor, except for itself. The system therefore acts as a guard, which on the one hand has full control, but on the other hand limits access to resources and other processes.

To achieve this, the processor has to support hardware-wise several privilege levels. The operating system then typically runs on the highest level and all other code on lower levels.

However, sometimes programs (or more correctly processes) need to access resources that controlled by the operating system. They need to communicate with the kernel. For that, an interface is needed and the concept of *System Calls* (or *Syscall* in short) was introduced. System calls represent a clear, strictly defined way to request kernel functions from within a process context.

This chapter shows an example usage of a system call and explains how to use and implement them in general.

7.1 The `sync` system call

To learn about system calls in general, a specific one is introduced: the `sync` system call.

The `sync` system call asks Linux to write all cached data to the persistent storage. This is especially useful if the specific storage should be removed or changed.

The `sync` call begins in the userland and is then executed in the kernel. After that, execution continues in userland. In figure 7.1 all stages of execution are shown. In the following part, the particular steps will be explained.

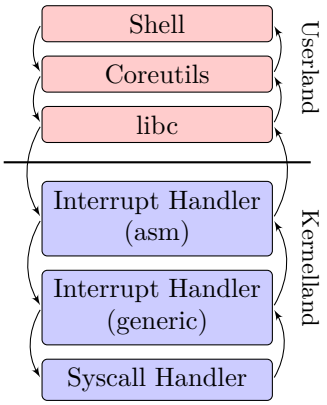


Figure 7.1: Visualisation of all parts that are needed for the execution of a system call.

```
#include "system.h"

int
main (int argc, char **argv)
{
  ...
  enum sync_mode mode;
  ...
  if (mode == MODE_SYNC)
    sync ();
```

Listing 7.1: Code of GNU coreutils sync (`./src/sync.c`)

```
#include <unistd.h>
#include "syscall.h"

void sync(void)
{
    __syscall(SYS_sync);
}
```

Listing 7.2: Implementation of the `sync()` function in musl libc (`./src/unistd/sync.c`)

7.1.1 Userland part

The `sync` call is often triggered by the program `sync`, that is part of GNU coreutils. Within the code of the `sync` program, some argument and error checking is done and subsequently `sync()` is called (see listing 7.1). `sync()` is defined in `stdlib.h`, a part of the libc.

This is the case for most system calls. A developer who wants to use a system call in most cases invokes a specific wrapper function of the libc. For the libc, multiple implementations exists. Here, the musl libc is taken. musl aims to be a tiny, compact libc and is therefore more readable than other implementations.

The implementation of the `sync()` call is shown in listing 7.2 and leads to the `__syscall` function, that is defined architecture specific. The x86 architecture is presented here (because of its popularity). The assembler routine is listed in listing 7.3.

So what does this routine do? First, it pushes to callee saved registers on the stack (line 4-7, they are later popped again). Then it moves the content at address `%esp + 20` to the EAX register (line 8). Because of the previous stack operations, ESP has moved, so `%esp + 20` points to the first argument of the function. In the case of the `sync()` call, this is `SYS_sync`, which is an alias for 36. In general, the first argument of a syscall is its number. A syscall is defined by a number and enable the kernel to decide what system call was invoked from userland. Back in the assembler code now the following arguments of the functions are pushed into the registers EBX, ECX, ... This belongs to the calling convention of system calls and is part of the interface of the Linux kernel. The kernel will later look in exactly these registers for the arguments of the syscall. If more than 6 argument are necessary, the last register stores a pointer to some memory area where the other arguments are stored.

After filling the registers, the `int 0x80` call is invoked (line 15, 0x80 in hexadecimal is 128 in decimal). This is a software interrupt that tells the processor to go into kernel mode and is exactly the point where the actual call happens.

When returning from the interrupt, the callee saved registers are restored (line 16-19) and the function returns the value stored in EAX.

```
.global __syscall
.type __syscall,@function
__syscall:
    pushl %ebx
    pushl %esi
    pushl %edi
    pushl %ebp
    movl 20(%esp),%eax
    movl 24(%esp),%ebx
    movl 28(%esp),%ecx
    movl 32(%esp),%edx
    movl 36(%esp),%esi
    movl 40(%esp),%edi
    movl 44(%esp),%ebp
    int $128
    popl %ebp
    popl %edi
    popl %esi
    popl %ebx
    ret
```

Listing 7.3: Implementation of the `__syscall` function in musl libc (`./src/internal/i386/syscall.s`)

```
static const __initdata struct idt_data def_idts[] = {
    ...
    SYSG(IA32_SYSCALL_VECTOR,   entry_INT80_32),
    ...
};

void __init idt_setup_traps(void) {
    idt_setup_from_table(idt_table, def_idts, ARRAY_SIZE(def_idts));
}
```

Listing 7.4: Setup of the interrupt routines for traps for x86 within the Linux Kernel (`./arch/x86/kernel/idt.c`).

7.1.2 Kernel part

Now the kernel has the task to execute the sync system call. Therefore, we first have to take a look what routine is called when the `int 0x80` call arrives. This is part of the boot process of the kernel and architecture specific again. The x86 trap is registered within the function `idt_setup_traps()` with the help of a table. The relevant part of the table is shown in listing 7.4. `IA32_SYSCALL_VECTOR` is simply a define for the number 128 (0x80). So it can be seen that the function `entry_INT80_32` is called, that is defined as an assembler routine (see listing 7.5). The assembler routine stores the values of the registers on the stack and calls the `do_int80_syscall_32` function, defined in C again. When returning from the C function. The registers are restored and `iret` is called[1]. This again leaves the kernel mode and the processor continues execution in the libc code. `do_int80_syscall_32` activates the interrupts again (`enter_from_user_mode()` is for tracing), and calls another function, that gets a function from the syscall table and calls it with the syscall arguments, that where stored in the registers (see listing 7.6. In case of `sync`, the table leads to the

```
ENTRY(entry_INT80_32)
    ... /* prepare C stack */
    call    do_int80_syscall_32
    ...
    RESTORE_REGS 4
    INTERRUPT_RETURN
```

Listing 7.5: Shortened implementation of the `entry_INT80_32` routine (`./arch/x86/entry/entry_32.S`).

```
void
do_int80_syscall_32(struct pt_regs *regs)
{
    enter_from_user_mode();
    local_irq_enable();
    do_syscall_32_irqs_on(regs);
}
...

static _always_inline void
do_syscall_32_irqs_on(struct pt_regs *regs) {
    unsigned int nr = regs->orig_ax;
    ...
    if (likely(nr < IA32_NR_syscalls)) {
        regs->ax = ia32_sys_call_table[nr](
            (unsigned int)regs->bx,
            (unsigned int)regs->cx,
            (unsigned int)regs->dx,
            (unsigned int)regs->si,
            (unsigned int)regs->di,
            (unsigned int)regs->bp);
    }
}
```

Listing 7.6: Implementation of the `do_int80_syscall_32` and `do_syscall_32_irqs_on()` functions (`./arch/x86/entry/common.c`).

```
SYSCALL_DEFINE0(sync)
{
    int nowait = 0, wait = 1;

    wakeup_flusher_threads(0, WB_REASON_SYNC);
    iterate_supers(sync_inodes_one_sb, NULL);
    iterate_supers(sync_fs_one_sb, &nowait);
    iterate_supers(sync_fs_one_sb, &wait);
    iterate_bdevs(fdatawrite_one_bdev, NULL);
    iterate_bdevs(fdatawait_one_bdev, NULL);
    if (unlikely(laptop_mode))
        laptop_sync_completion();
    return 0;
}
```

Listing 7.7: Actual implementation of the `sync` system call (`./fs/sync.c`).

```
#define SYSCALL_DEFINE0(sname)      \
    SYSCALL_METADATA(_##sname, 0); \
    asmlinkage long sys_##sname(void)
```

Listing 7.8: Definition of the SYSCALL_DEFINE0 macro (./include/linux/syscalls.h).

```
asmlinkage long sys_read(unsigned int fd, char __user * buf, size_t count)
    __attribute__((alias(__stringify(SyS_read))));

static inline long SYSC_read(unsigned int fd, char __user * buf, size_t count);
asmlinkage long SyS_read(long int fd, long int buf, long int count);

asmlinkage long SyS_read(long int fd, long int buf, long int count)
{
    long ret = SYSC_read((unsigned int) fd, (char __user *) buf, (size_t) count);
    asmlinkage_protect(3, ret, fd, buf, count);
    return ret;
}

static inline long SYSC_read(unsigned int fd, char __user * buf, size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;
    /* ... */
```

Listing 7.9: Expansion of the SYSCALL_DEFINE3 macro (./include/linux/syscalls.h).

sys_sync function.

When searching for the implementation of sys_sync, only the header can be found in ./include/linux/syscalls.h. The implementation is hidden with some preprocessor macros in ./fs/sync.c and shown in listing 7.7. The interesting part is the SYSCALL0 macro, that is defined in ./include/linux/syscalls.h again, see listing 7.8. It can be seen that the expansion leads directly to the definition of the sys_sync function[2].

7.2 Syscalls with arguments

The handling of sync was straightforward and simple. But what about system calls with arguments?

Such system calls can also be defined with a SYSCALLx macro, where x is the number of arguments of the syscall. However, these macros are not that simple. As an example, listing 7.9 shows the expansion of the read system call. It can be seen that the actual implementation is in the function SYSC_read, that is defined static and therefore hidden outside of the module. SYSC_read is called from SyS_read, that takes only long as arguments. However, SyS_read is aliased (a GCC extension) as sys_read with the correct types. This indirection has its origin in an ancient bug and leads to correct extension of the parameters.

7.3 Other input paths

What was not said up to now is that the part of musl shown above is actually the musl implementation of 2012. In 2012 musl was extended to support the sysenter directive. Sysenter/Syscall is a mechanism

[1] Actually INTERRUPT_RETURN is called, which is a macro for iret. This is necessary to enable solutions like Xen to redefine the INTERRUPT_RETURN call.
[2] The SYSCALL_METADATA code is for tracing and in most cases compiled away.

of newer x86 Intel and AMD chips that provides a faster input path into the kernel. An interrupt is extremely expensive, especially when you invoke a system call like `getpid()` that only returns the value of one variable. Therefore, hardware manufactures made other mechanisms available that are more complex to use but magnitudes faster.

In recent years, another solution of the speed problem was invented – the vDSO mechanism. This is a page of kernel memory mapped into the userspace, that act as shared library. The background is that some system calls are considered safe, even if they are in userspace, e.g. the `gettimeofday()` system call. With vDSO, an application can bind against the vDSO library and use the system call routines like regular functions, without ever entering the kernel mode.

7.4 Conclusion

In this chapter, system calls were explained. It was shown how a system call is invoked from userland (in the simplest case). Moreover, the handling of the system call in kernelland both the architecture specific part and the generic part was shown.

8 Timer

The kernel has to deal with time somehow. The most prominent reason for this is the frequent call of the scheduler, but also other subsystems need time. Examples are the network stack and a whole bunch of drivers.

The solution of Linux is to activate and handle a timer interrupt. To generate the interrupt, a piece of hardware has to be capable of ticking in a specific frequency.

On the basis of the interrupt, several other actions take place. In this chapter, the measurement of time in the Linux kernel will be explained and the technique of timers. The last part will explain how code can wait in an efficient way.

8.1 Hz

As explained, Linux works with a timer interrupt. But how often should the interrupt trigger? For this question, no distinct answer exists so Linux let the user choose the frequency.

The chosen frequency is stored in the global preprocessor value `HZ`. Listing 8.1 shows the possible configuration values. It has to be said that newer Linux kernels are capable of deactivating the timer interrupt in certain circumstances. The default setting is to turn the timer interrupt off if the system is on idle. This leads to a significant decrease in power consumption. It is also possible to deactivate the timer interrupt on SMP systems on all cores except one. However, this is mostly only useful for special purposes like HPC or on embedded devices. Listing 8.2 shows the appropriate settings.

8.2 Jiffies

The knowledge of how often the timer interrupt occurs does not help much. Also necessary is some kind of measurement when the interrupt occurs. Therefore, a global variable called `jiffies` exists (see listing 8.3. This variable stores the amount of *ticks* (a single occurrence of the timer interrupt) in the system.

`jiffies` is stored as an unsigned long and has therefore a length of 32 bit on 32 bit systems. With 1000 Hz, this leads to an overflow within 49 days. Because of this, a second global variable of type `u64` exists, which is 64 bits long on all architectures and never overflows[1]. The two variables are then overlined at link time, so they use the same memory (see figure 8.1).

[1] In fact, it overflows but at a frequency of 1000 Hz the overflow happens after 580 million years.

```
Processor type and features  --->
    Timer frequency (1000 Hz) --->
        100 HZ
        200 HZ
        300 HZ
    <X> 1000 HZ
```

Listing 8.1: Configure options for the value of `HZ`.

```
General setup  --->
    Timers subsystem  --->
        Timer tick handling (Idle dynticks system (tickless idle))  --->
            Periodic timer ticks (constant rate, no dynticks)
        <X> Idle dynticks system (tickless idle)
            Full dynticks system (tickless)
```

Listing 8.2: Configure options for the type of kernel ticks.

```
/*
 * The 64-bit value is not atomic - you MUST NOT read it
 * without sampling the sequence number in jiffies_lock.
 * get_jiffies_64() will do this for you as appropriate.
 */
extern u64 __cacheline_aligned_in_smp jiffies_64;
extern unsigned long volatile __cacheline_aligned_in_smp jiffies;
```

Listing 8.3: Global definition of the `jiffies` variable `./include/linux/jiffies.h`.

8.2.1 Calculating with Jiffies

As of the different frequencies of the kernel, the length of a jiffy is not fixed. But is can be calculated using this relation:

$$\texttt{HZ} = \frac{\#\text{ Jiffies}}{s}$$

The amount of jiffies per second is equal to the freqency. Therefore, the length of one Jiffy is $\frac{1}{HZ}$.

Another point of calculating with jiffies is the prevention of the overflow. It is often needed to do calculations like this:

```
unsigned long before = jiffies;

do_action();

if ((jiffies - before) > MAXIMUM_JIFFIES)
    klog("action required too much time.")
```

This code can easily overflow and result in wrong conditions. Therefore, the kernel provides some macros as shown in listing 8.4 that prevent the overflow with some casting tricks.

With that, the initial value of jiffies can now be shown:

```
#define INITIAL_JIFFIES ((unsigned long)(unsigned int) (-300*HZ))
```

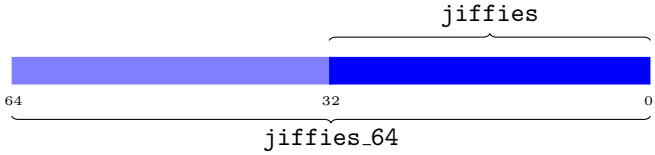


Figure 8.1: Memory location of `jiffies` and `jiffies_64`.

```
/*
 * time_after(a,b) returns true if the time a is after time b.
 */
#define time_after(a,b)    ((long)((b) - (a)) < 0)
#define time_before(a,b)   time_after(b,a)
```

Listing 8.4: Macros for preventing overflows with `jiffies ./include/linux/jiffies.h`.

```
static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {
        ...
        do_timer(1);
        ...
        update_wall_time();
    }

    update_process_times(user_mode(get_irq_regs()));
    ...
}
```

Listing 8.5: Definition of `tick_periodic ./kernel/tick/tick-common.c`.

It is set to 5 minutes before the overflow occurs. This is a trick to trigger kernel bugs related to the jiffies overflow in a realistic time.

8.3 The timer interrupt handling

The `tick_periodic` function (listing 8.5) is the entry point of the timer interrupt. `tick_periodic` does several things:

1. Update the jiffies variable. The function definition for `do_timer` is shown in listing 8.6. The update is saved because the timer lock is taken before.
2. Recalculate the load (also in `do_timer`). This is a global measure for the system load.
3. Update the process wall time, so the runtime of processes can be measured.
4. Call `update_process_timers` (listing 8.7) that
   a) triggers the execution of timers (see next section).
   b) triggers the scheduler.

```
void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks;
    calc_global_load(ticks);
}
```

Listing 8.6: Definition of `do_timer ./kernel/time/timekeeping.c`.

```
void update_process_times(int user_tick)
{
    ...
    run_local_timers();
    ...
    scheduler_tick();
}
```

Listing 8.7: Definition of `update_process_timess ./kernel/time/timer.c`.

```
struct timer_list {
    struct hlist_node   entry;
    unsigned long       expires;
    void            (*function)(struct timer_list *);
    u32         flags;
};
```

Listing 8.8: Definition of the timer data structure `./include/linux/timer.h`.

8.4 Timer

Sometimes code has to execute some actions in the future. An example is some network code that specifies a timeout and has to react somehow if the remote side does not answer.

Therefore, the concept of *timers* exists. Timers are defined with the data structure listed in 8.8. The two imported values are `expires`, the moment where the timer is executed, and `function`, the function that is executed when the moment arrives.

A timer can be set up with the `setup_timer` function. After that, it can be added to the timer wheel (the infrastructure that take care of executing timers), the expire time can be modified or the timer can be deleted. The interfaces are shown in listing 8.9. Denote the `del_timer_sync` function: On SMP systems timers can be deleted on one core, while executing on another. `del_timer_sync` takes care that the timer is not executed while the deletion takes place.

Level	Offset	Granularity		Range		
0	0	1 ms		0 ms -	63 ms	
1	64	8 ms		64 ms -	511 ms	
2	128	64 ms		512 ms -	4095 ms	(512ms – ∼4s)
3	192	512 ms		4096 ms -	32767 ms	(∼4s – ∼32s)
4	256	4096 ms	(∼4s)	32768 ms -	262143 ms	(∼32s – ∼4m)
5	320	32768 ms	(∼32s)	262144 ms -	2097151 ms	(∼4m – ∼34m)
6	384	262144 ms	(∼4m)	2097152 ms -	16777215 ms	(∼34m – ∼4h)
7	448	2097152 ms	(∼34m)	16777216 ms -	134217727 ms	(∼4h – ∼1d)
8	512	16777216 ms	(∼4h)	134217728 ms -	1073741822 ms	(∼1d – ∼12d)

Table 8.1: Levels of the Linux timer wheel.

```
/**
 * timer_setup - prepare a timer for first use
 * @timer: the timer in question
 * @callback: the function to call when timer expires
 * @flags: any TIMER_* flags
 *
 * Regular timer initialization should use either DEFINE_TIMER() above,
 * or timer_setup(). For timers on the stack, timer_setup_on_stack() must
 * be used and must be balanced with a call to destroy_timer_on_stack().
 */
#define timer_setup(timer, callback, flags) ...
#define timer_setup_on_stack(timer, callback, flags) ...
...
extern void add_timer_on(struct timer_list *timer, int cpu);
extern int del_timer(struct timer_list * timer);
extern int del_timer_sync(struct timer_list *timer);
extern int mod_timer(struct timer_list *timer, unsigned long expires);
```

Listing 8.9: Usage functions of timers `./include/linux/timer.h`.

8.4.1 The timer wheel

The three actions of timers have to be done as soon as possible. Especially the check if timer has to be expired should work within an appropriate speed. This resulted in different implementations of the timer subsystem. The current one is a rework of the timer wheel of 2015.

The timer wheel consists of a number of buckets, where timers could be placed that are executed at specific times. The access to the bucket is an O(1) operation. However, because of memory limitations, the resolution of the buckets is different, so they are stored in different levels. For a 1000 Hz system all levels are shown in table 8.1.

So, if the timer should expire within in the next 64 ms, it is put into the appropriate bucket with 1 ms resolution. But if it should expire within the next 453 ms, it is put into a bucket with 8 ms resolution.

This results in fast adding, expiration and removing times, but leads to imprecise moments of expiration. However, research of use cases of timers in the kernel has shown that most timers are used for timeouts that are deleted before execution. Even if they are executed, it is not relevant if the execution time is precise.

8.5 Waiting

With that, the `timeout_schedule` function could be explained. It is sometimes wanted to wait for a certain time. This can be achieved with the `timeout_schedule` function.

The function simply defines a timer, sets the appropriate expire time and schedules the current thread away (see listing 8.10). Of course, this only works with waiting times longer than one jiffy.

For waiting time shorter than one jiffy, the kernel provides three methods, that are based on busy looping (see listing 8.11). A loop is executed as long as the subpart of the jiffy has elapsed. Therefore, the kernel

```
signed long __sched schedule_timeout(signed long timeout)
{
    ...
    expire = timeout + jiffies;

    timer.task = current;
    timer_setup_on_stack(&timer.timer, process_timeout, 0);
    __mod_timer(&timer.timer, expire, 0);
    schedule();
    ...
}
```

Listing 8.10: Implementation of `schedule_timeout` `./kernel/time/timer.c`.

```
/*
 * Delay routines, using a pre-computed "loops_per_jiffy" value.
 */
#define udelay(n) ...
#define ndelay(n) ...
#define mdelay(n) ...
```

Listing 8.11: Definitions for sub jiffy delays `./include/asm-generic/delay.h`, `./include/linux/delay.h`.

has to know how often the loop can be executed within one jiffy. This is measured at boot time and results in the *BogoMips* value, that is given for example in `/proc/cpuinfo`.

8.6 Conclusion

In this chapter, the two values HZ and jiffies were introduced. It was shown that Linux can tick with different frequencies and that the tick can be turned off under certain circumstances. The overflow of the jiffies variable was shown and how it could be handled.

After that, the concept of timers was introduced, followed by a short explanation how they are implemented and how they can be used to wait.

9 Memory Management

Just like user processes, the kernel's processes need memory to operate. Yet within the kernel, there are no predefined functions that can be called to allocate and free memory. Thus, the kernel needs to define its own memory management functionality.

To do so, the concepts of nodes, zones and pages are introduced. They intend to describe memory in a way that is as architecture independent as possible to support multiple architectures. The memory management functionality is built upon these concepts.

This chapter first introduces nodes, zones, pages and how they are related to each other. Following this, various approaches to manage memory are presented, including low-level mechanisms, the buddy system and the slab allocator.

9.1 Description of Physical Memory

First of all, it is explained how physical memory is described by the Linux kernel.

9.1.1 UMA and NUMA Architectures

There are *Uniform Memory Access* (UMA) and *Non-Uniform Memory Access* (NUMA) architectures.

In a UMA architecture, each CPU is connected to the same memory. The time needed to access the global memory is equal for each CPU. This architecture is also referred to as *symmetric multiprocessing* (SMP).

In contrast, in the NUMA architecture each CPU has its own memory, which is connected to the other CPUs through a bus, as opposed to having one global memory. A CPU can access its own memory as well as the local memory of another CPU, since they are connected and share the same address space. The time needed to access the memory of another CPU, however, is greater than the time needed to access the own local memory.

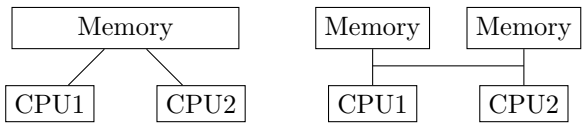


Figure 9.1: UMA (left) and NUMA (right) [4]

9.1.2 Nodes

Each individual RAM used on NUMA (and UMA) machines is called *node*. On a UMA machine, there is only one such node, whereas on NUMA machines there can be multiple ones. Nodes are described by the `struct pglist_data`, referenced by `pg_data_t` and kept on a NULL terminated list.

These nodes are further divided into zones, which represent ranges within memory.

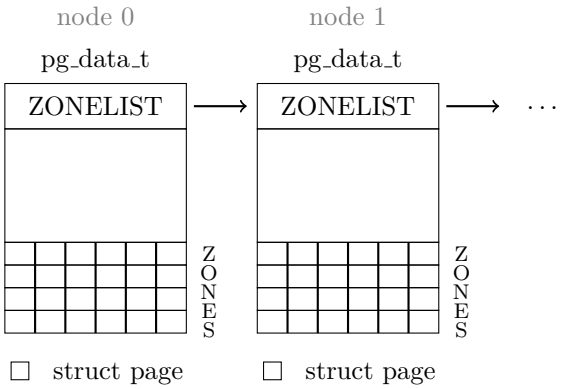


Figure 9.2: Nodes in a NUMA system [4]

9.1.3 Zones

The division of memory into zones is due to limitations of physical memory.

In some systems, there is only a small fraction of memory that is able to perform *Direct Memory Access* (DMA). In the x86 architecture, this is the memory ranging from 0 MiB to 16 MiB. DMA is a hardware mechanism that allows peripheral devices to read and write data directly from and to memory independently of the CPU. It is used for devices such as sound cards and video cards that need to process much data. The devices account for the reason as to why only a part of physical memory can be used for DMA. For instance, ISA devices are limited to 24-bit addresses, because the bus offers no more bandwidth. In most modern systems, however, there no longer is such a problem.

Another limitation concerns the so-called high memory. High memory is memory that can not be addressed directly. For example, 32-bit systems can only address up to 4 GiB of memory. Thus, logical addresses do not exist for this memory, and the kernel can not access it directly without first setting up a special mapping. In 64-bit systems, there usually is no high memory since memory can be addressed with 64-bit addresses, which suffice in most cases.

The zones resulting from above mentioned physical limitations are *ZONE_DMA* and *ZONE_HIGHMEM*, the rest of the memory being in *ZONE_NORMAL*.

In 64-bit systems, a zone *ZONE_DMA32* is established for DMA-able memory that can only be addressed with 32 bits. The kernel also implements the pseudo-zones *ZONE_MOVABLE* and *ZONE_DEVICE*, for memory the kernel can move to a different location, and memory that is associated with a hardware device and does not belong to main memory.

The occurring zones are:

Table 9.1: Zones in the Linux Kernel

Zone	Description
ZONE_DMA	DMA-able memory
ZONE_DMA32	DMA-able memory, addressable with 32 bits
ZONE_NORMAL	Normal, regularly mapped memory
ZONE_HIGHMEM	High memory, not permanently mapped
ZONE_MOVABLE	Memory the kernel can move
ZONE_DEVICE	Device memory

The zones are declared in `linux/mmzone.h`:

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
    ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32,
#endif
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE,
#endif
    __MAX_NR_ZONES
};
```

`MAX_NR_ZONES` is an end marker used when iterating over the zones, since not all zones have to be used. In `mmzone.h`, a **`struct zone`** holds information like the size of the zone, its name, the node it is on, and much more.

To keep track of where the zones begin, a `zone_mem_map` points to the first page that a zone refers to.

9.1.4 Pages

The term *page frame* refers to a physical page, whereas a *page* concerns pages in virtual address space.

In most 32-bit architectures, a page is of size 4 KiB, and in case of 64-bit architectures, it is usually of size 8 KiB.

Each page frame is represented by a **`struct page`** in `linux/mm_types.h`:

```
struct page {
    ...
    unsigned long flags;            /* status of the page */
    atomic_t _count;                /* number of references to this page */
    struct address_space *mapping; /* address space in which the page
                                                    frame is located */
    void *virtual;                  /* virtual address, if mapped */
    ...
};
```

This is only a fraction of the fields described in the **`struct page`**. The kernel makes an effort to keep this structure small (since every physical page frame has one and therefore takes up memory), which is why many unions are used.

All these structures are kept in a global array called `mem_map`.

9.2 Managing Memory

Since allocating and freeing memory is a frequently occurring task, it needs to be done quickly. Moreover, in order to allocate memory, the kernel has to know which pages are allocated and which are free. It also aims to reduce fragmentation of memory, which poses a threat to performance because pages often need to be physically contiguous.

9.2.1 Allocating and Freeing Pages

To allocate 2^{order} contiguous pages, the following functions can be used:

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

unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)

unsigned long get_zeroed_page(unsigned int gfp_mask)
```

The first function, `alloc_pages`, returns the first `struct page` of the allocated memory. `__get_free_pages` does not return a `struct page`, but an address to the page. If a page filled with zeroes is needed, `get_zeroed_page` can be called.

To later free the pages, `free_pages` can be called:

```
void free_pages(struct page *page, unsigned int order)

void free_pages(unsigned long addr, unsigned int order)
```

This frees 2^{order} pages, starting at either the given `struct page` or address.

The `gfp_mask` (gfp standing for *get free pages*) flags that are used for allocations define the behavior of the allocator. These flags are divided into three categories: action modifiers, zone modifiers and type flags. Action modifiers specify how the memory should be allocated, zone modifiers state the zone it should be allocated from, and the type flags are a combination of both, now commonly used and meant to simplify this concept.

For example, the flag `GFP_ATOMIC` is used for allocations with high priority that are not allowed to sleep or block, such as in an interrupt context, and `GFP_KERNEL` for regular allocations within the kernel that may sleep. The entirety of flags can be found in `linux/gfp.h`.

9.2.2 High Memory Mappings

High memory is mapped via page tables. They can be either permanently mapped using `kmap`, or temporarily mapped with `kmap_atomic`.

```
void *kmap(struct page *page)

void *kmap_atomic(struct page *page, enum km_type_type)
```

`kmap` returns a virtual address for any page in the system. If the pages are located in the normal zone, it simply returns their logical addresses, and if the pages reside in high memory, it maps them into the virtual address space. The number of permanent mappings is limited. The mappings can be removed using the function `kunmap`. `kmap_atomic` can be useful in an interrupt context, since the temporary mapping does not require the process to sleep to map the page. It also performs better than `kmap`, because it merely uses a dedicated slot where it temporarily maps the page into.

9.2.3 Per-CPU Allocations

At times, CPUs require data that only belongs to them. This data is usually kept inside an array. It is not protected with some sort of lock, thus while writing to this array, a process can be rescheduled on another CPU, having lost its data. It can also be overwritten by a different process.

Per-CPU data can be allocated using the `alloc_percpu` function:

```
1 void *alloc_percpu(type)
```

9.2.4 Allocation Interfaces

There are higher-level interfaces for memory allocations that work more efficiently and prevent fragmentation of memory. The figure represents the hierarchy of the methods that are introduced in the following part of this chapter, building upon the page frames that were discussed above.

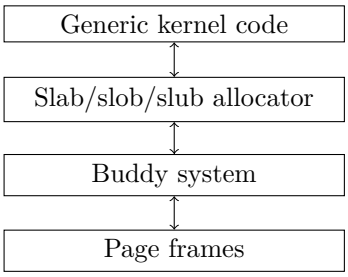


Figure 9.3: Hierarchy of allocation methods

9.2.5 The Buddy System

The buddy system means to provide a quick method to find an area within memory that has the required amount of free, physically contiguous pages. This algorithm furthermore is supposed to reduce *external fragmentation*, which occurs when free pages are scattered within memory, thus not being able to serve a request for contiguous pages, even if the sum of free pages suffices.

The concept of the buddy system lies in so-called *free lists*. Memory is divided in blocks of pages, each block being a power of two number of pages. A free list for each power of two up to a specified `MAX_ORDER` keeps track of these blocks. Figure 9.4 offers a visualization for better understanding of these free lists.

When a certain amount of memory is requested, yet a block of this size not available, a large block is split up in half, and the two originating blocks become *buddies*. This is done until halving a block would no longer suffice the required amount of memory. When a block is freed later on, and its buddy is not allocated, the two buddies are reunited.

Despite its efforts to reduce external fragmentation, internal fragmentation can still occur. This can be observed in the following example.

Example 9.1 *A process wants to allocate 3 KiB of memory. There is no free block of 4 KiB, yet a block of 8 KiB is available. So, this 8 KiB block is divided into two 4 KiB buddies.*

As seen in figure 9.5, a 4 KiB block of memory is allocated, yet only 3 KiB are used. The remaining memory is allocated, but it remains unused. This is called internal fragmentation.

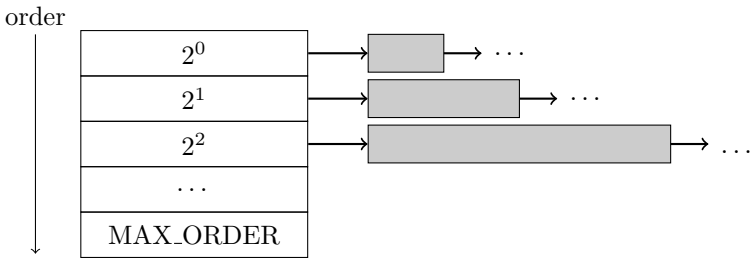


Figure 9.4: Free lists of the buddy system [1]

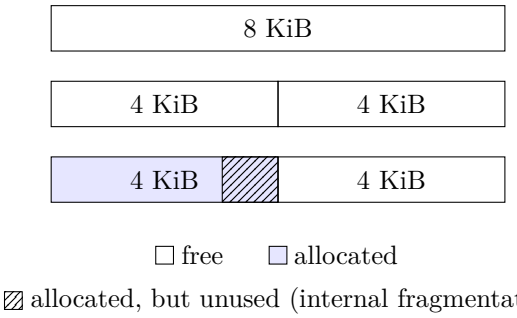


Figure 9.5: Allocating memory using the buddy system

9.3 The Slab Allocator

While the buddy system proved to be an effective method to conquer external fragmentation, the slab allocator means to reduce internal fragmentation. It does so by dividing the pages into smaller chunks of memory, called *slab caches*. These slab caches are created for both directly allocating objects frequently used by the kernel, and for the general allocation of memory blocks smaller than a page.

9.3.1 Slab Caches

A slab cache is created for each type of object. Also, there exist slab caches in different sizes for later use in `kmalloc`.

A slab cache contains several *slabs*, which are structures holding *objects* of the type in the current slab cache.

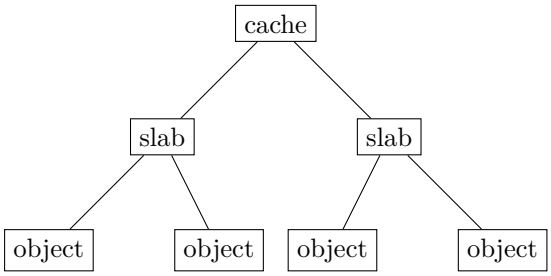


Figure 9.6: Relationship between caches, slabs, and objects [3]

The caches are defined in `mm/slab.h`:

```
struct kmem_cache {
    unsigned int object_size; /* Original size of object */
    unsigned int size;        /* Aligned/padded/added on size  */
    unsigned int align;       /* Alignment as calculated */
    slab_flags_t flags;       /* Active flags on the slab */
    const char *name;         /* Slab name for sysfs */
    int refcount;             /* Use counter */
    void (*ctor)(void *);     /* Called on object slot creation */
    struct list_head list;    /* List of all slab caches on the system */
};
```

Each cache has a name, a size, several flags to describe its slabs and a list to those slabs it contains.

A slab consists of one or more physically contiguous pages. It can have one of three states: full, partial, or empty. If it is full, all objects contained in the slab are allocated. If it is free, none are allocated, and if it is partial, there are some objects that are still free, but not all of them.

9.3.2 Slob and Slub Allocator

There are two different kinds of allocators associated with the slab allocator. These are the *slob* and the *slub* allocator.

The slob allocator is implemented using simple linked lists of blocks. To allocate memory, a simple first-fit-algorithm is used, stepping through the blocks of memory and allocating a free block as soon as one whose size suffices is found. It is a simple, yet slow algorithm used in small-scale systems that need to be very compact.

The slub allocator is built upon the slab allocator. Its goal is to reduce overhead by grouping page frames and overloading unused fields in `struct page`. It has proved to be more efficient than the slab allocator, and therefore is used by default in the Linux kernel. Thus, the term *slab allocator* does not imply usage of the *slab* allocator per se, but rather refers to the concept as introduced above.

9.3.3 Allocating a Slab

When an object is required, it can be allocated from the cache, from which it is then removed until the object is freed again. The slab cache automatically handles the interaction with the buddy system and requests new pages if needed.

The function `kmem_cache_alloc` is used to allocate an object from the cache.

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
{
    void *ret = slab_alloc(cachep, flags, _RET_IP_);

    kasan_slab_alloc(cachep, ret, flags);
    trace_kmem_cache_alloc(_RET_IP_, ret, cachep->object_size,
                                          cachep->size, flags);

    return ret;
}
```

This function gets a pointer `cachep` to a cache that was previously created. `slab_alloc` tries to get an object from this cache. Yet if the allocation fails, first the local node, and in case it fails again, other nodes, if available, are examined in order to find a free object. If no free object could be found, new pages are allocated using the given `gfp_t` flags. An existing cache then *grows*. The pointer `ret` then points to the allocated object, or `NULL` in case any errors occured. The rest of this function is merely tracing and error detecting (KASAN standing for Kernel Address Sanitizer, a fast memory error detector), which will also appear in the functions discussed next.

Its counterpart is the function `kmem_cache_free`, used to free a cache.

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp) {
    unsigned long flags;
    cachep = cache_from_obj(cachep, objp);
    if (!cachep)
        return;

    local_irq_save(flags);
    debug_check_no_locks_freed(objp, cachep->object_size);
    if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
        debug_check_no_obj_freed(objp, cachep->object_size);
    __cache_free(cachep, objp, _RET_IP_);
    local_irq_restore(flags);

    trace_kmem_cache_free(_RET_IP_, objp);
}
```

The function gets a pointer to the object to be freed, and the cache containing it. It checks whether the object actually belongs to the given cache, whether it is no longer used and whether there are no locks held within the memory to be freed. If so, `__cache_free` releases the object back to its cache. This is done while disabling interrupts with `local_irq_save` to ensure proper deallocation of the object.

9.3.4 kmalloc, kfree

The `kmalloc` and `kfree` functions are the kernel's equivalent to `malloc` and `free` in the userspace. They are intended to allocate and free memory in byte-sized granularity, as opposed to objects, and built upon the slab allocator.

To allocate memory, `kmalloc` is defined as follows:

```
static __always_inline void *__do_kmalloc
          (size_t size, gfp_t flags, unsigned long caller) {
    struct kmem_cache *cachep;
    void *ret;

    cachep = kmalloc_slab(size, flags);
    if (unlikely(ZERO_OR_NULL_PTR(cachep)))
        return cachep;
    ret = slab_alloc(cachep, flags, caller);

    kasan_kmalloc(cachep, ret, size, flags);
    trace_kmalloc(caller, ret, size, cachep->size, flags);

    return ret;
}
```

At first, `kmalloc_slab` finds a `kmem_cache` structure that serves the given *size* of allocation. If such a cache was found, just as `kmem_cache_alloc` does, `slab_alloc` is called to allocate a slab within this cache, or, as mentioned above, find or grow a different cache if there is no object of requested size. The pointer to the slab, or a `NULL` pointer in case of failure, is returned.

To free this slab, `kfree` is used:

```
1
void kfree(const void *objp) {
    struct kmem_cache *c;
    unsigned long flags;
5
    trace_kfree(_RET_IP_, objp);

    if (unlikely(ZERO_OR_NULL_PTR(objp)))
        return;
10  local_irq_save(flags);
    kfree_debugcheck(objp);
    c = virt_to_cache(objp);
    debug_check_no_locks_freed(objp, c->object_size);

15  debug_check_no_obj_freed(objp, c->object_size);
    __cache_free(c, (void *)objp, _RET_IP_);
    local_irq_restore(flags);
}
```

`kfree` is called with a pointer to the object to be freed. Again, interrupts are temporarily diasbled. Using the pointer to the object, the cache it should be returned to is found. After ensuring the object is neither locked nor already freed, `__cache_free` returns the object to the cache it belongs to.

9.3.5 Allocating Non-Contiguous Memory

Sometimes, the kernel does not need to allocate contiguous pages. That is why the functions `vmalloc` and `vfree` were implemented. Similarly to what `malloc` and `free` in userspace do, `vmalloc` looks for free pages that suffice the requested amount of memory and if they are not consecutive, it adjusts the page table entries in a way that these pages now are *virtually contiguous*.

The preferred method of allocating a chunk of memory is `kmalloc`, which is more efficient, since `vmalloc` produces a lot of overhead by having to adjust the page table entries each time. Yet for larger areas of memory, the required amount of physically contiguous pages may not be available. Hence, `vmalloc` is a appropriate option if a large amount of memory is used.

For allocations where `kmalloc` is the preferred option for memory allocation, yet might fail to allocate memory because the requested block is too large, a function called `kvmalloc` can be used. This function first tries to allocate memory with `kmalloc`, but if this allocation fails, `vmalloc` is called.

9.3.6 Choosing an Allocation Method

The question which method to choose for memory allocation depends on the kind of allocation.

Most of the time, `kmalloc` will be the best option, even if the pages need not be physically contiguous. If, however, a large amount of memory is requested, `vmalloc` or `kvmalloc` may be more appropriate. Should the desired memory be allocated from the high memory, it can be done so by using `alloc_pages`. In case of often used data structures, it can be beneficial to create a slab cache. Frequent allocation and freeing of

these data structures can be done more efficiently this way, since there is no need to look for a free chunk of memory and initialize the structure before each allocation.

9.4 Conclusion

In this chapter, the description and management of physical memory was introduced.

There exist two memory architectures, one being the Uniform Memory Access and the other being the Non-Uniform Memory Access architecture. Memory is divided into nodes, one for each RAM on the system. These nodes contain different zones resulting from physical limitations. Inside these zones, the page frames are located, the smallest unit used for memory management. Page frames are represented respectively by a `struct page`.

There are several low-level methods to allocate and free these pages. Because of physical limitations, not all memory is treated equally. For instance, high memory needs to be mapped in 32-bit systems in order to be used.

Several interfaces are used to allocate memory in an easier and more efficient way. These include the buddy system, which works with free lists and splits bigger blocks of memory in half, and the slab allocator, creating caches for frequently used data structures and general purpose caches for the allocation of smaller chunks of memory, the latter being used by `kmalloc` and `kfree`.

10 VFS

The virtual file system is an abstraction layer. User-space applications do not have to bother whether they are trying to access a file on an ext4 or btrfs partition. To achieve that, the kernel provides an abstraction on top of the specific file systems: the VFS. This layer defines an interface and standard implementation of functions like read or write, and the different file system drivers have to implement these functions. The structure of the VFS is similar to Unix file systems. Other file systems have to do more processing to adjust to the interface. That is the cost of such an abstraction.

10.1 Filesystem

The following, simplified structs contain how the kernel represents file systems, superblocks and mount points. These parts are necessary to bind a file system in the Linux file system tree.

10.1.1 file_system_type

The `file_system_type` corresponds to `ext4` for example. It describes the capabilities of one file system and its struct includes:

- `name`: The name of the file system (e.g. `ext4`).
- `fs_flags`: Different flags (e.g. whether it is virtual file system).
- `mount`: A function which is called to mount a partition with that file sytsem.
- `kill_sb`: A function that is called when such a partition is unmounted.
- `next`: A list pointer for a list of all supported file systems.
- `fs_supers`: A list pointer for a list of superblocks of that file system.

This struct is defined in `include/linux/fs.h`.

10.1.2 superblock

Superblocks contain information regarding a partition. They sometimes even correspond to a part of the file system which is the case for `ext4`. The struct itself is quite large with over 50 members and is also defined in `include/linux/fs.h`. It contains information such as:

- `s_list`: A list pointer for the list of all superblocks.
- `s_blocksize`: The block size.
- `s_maxbytes`: The maximum file size.
- `s_type`: The type of the file system.
- `s_flags`: Flags such as `SF_NO_TRUNCATE` (file names must not be shortened) or `SF_IMMUTABLE` (file system cannot be changed).
- `s_root`: The root directory (which is `NULL` for pseudo file system).
- `s_bdev`: The underlying block device (e.g. `/dev/sda1`).
- `s_id[32]`: A user defined id (known as partition name).
- `s_uuid`: A number to uniquely identify the partition.

- `s_max_links`: The maximum number of links to an inode (65000 for `ext4`).
- `s_inodes`: A list of all inodes in that partition.

A part of this information is shown when working with tools like `fdisk`, for example the block size, the block device, the id and the uuid.

10.1.3 vfsmount

Once a partition is supposed to be mounted, this is described using the `vfsmount` struct. This struct contains only three members:

- `mnt_root`: The root of the partition
- `mnt_sb`: The superblock
- `mnt_flags`: Mount flags such as `NOEXEC`

More interesting than the struct itself is the fact that in Kernel 2.6.34 the struct still had more than 20 members. Although the virtual file systems already exists for a long time in the kernel, there are still drastic changes to it.

10.2 Resolving a file path

In this section, the concepts that are needed to resolve a file path like `/home/user/file.txt` will be explained.
The validity of the path is checked with the help of dentries. The inode contains meta information about the file like the permissions. Finally, the file struct is used to actually read the contents from the hard drive.

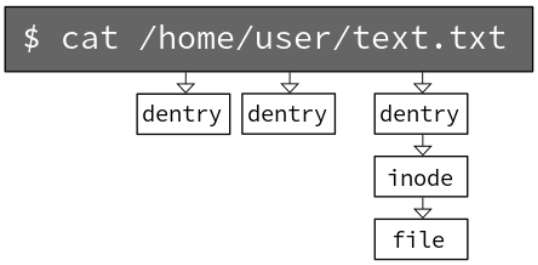


Figure 10.1: The concept of path resolving in Linux.

10.2.1 dentry

Dentries represent parts of a file path and check whether it actually exists. Parsing a file path is an expensive operation and dentries are supposed to make this operation as efficient as possible. The dentry struct contains among other things the following members:

- `d_hash`: A pointer for the linked list in the hash table bucket.
- `d_parent`: The parent dentry.
- `d_name`: The name of the file or directory.
- `d_inode`: The corresponding inode. If the inode is `NULL`, the dentry is invalid.
- `d_iname`: An array for direct storage of small names.j
- `d_sb`: The superblock.
- `d_time`: A timestamp used for revalidation.
- `d_lru`: A least-recently-used linked list that acts as a cache.
- `d_child`: A list of children of the dentry.

- `d_subdirs`: Pointer to the neighboring dentry.

The operations that are defined for inodes are among others [1]:

- `revalidate`: Check whether a dentry in the cache is still valid.
- `hash`: Insert dentry into the hash table.
- `compare`: Compares two filenames. This gets useful for case-insensitive filesystems
- `delete`: Decides whether the dentry should be cached when the last reference is removed.
- `iput`: Called when a dentry loses its inode.

It is important to note that the word *dentry* is misleading: Both files *and* directories are representet by dentries.

To resolve the file path, a lookup function, which checks whether a directory contains a file or directory with the given name, is used. The advantage of the dentry concept is that once these dentries are created, they are cached extensively. There is a dcache which contains both a LRU list and a hash table whereby the mapping to an inode can take place without going through all the previous directories. Even invalid dentries (without an inode) are kept around because it can speed up the lookup of invalid paths. So once the final dentry has been found, the next step is to access its inode.

10.2.2 inode

The metadata of files is represented by inodes. These are typically information the command `ls -l` will show and a couple of internal data. It includes:

- `i_mode`: The access rights
- `i_uid`: The id of the user who owns the file or directory
- `i_gid`: The id of the owning group
- `i_flags`: Flags such as `NOATIME` (inode has no access time) or `COMPR` (inode can be modified with ioctl)
- `i_sb`: The superblock that contains the inode
- `i_nlink`: The number of hard links
- `i_size`: The file size
- `i_atime`: The access time (change time and modification time were omitted in this listing)
- `i_state`: Status flags like `I_DIRTY_SYNC`
- `i_hash`: List Pointer for a hash table entry (which is used for caching)
- `i_lru`: List Pointer for a least-recently-used cache
- `i_sb_list`: List Pointer for the list of all inodes of the superblock
- `i_count`: The number of processes that are accessing the inode

Again, the inode is used for both files *and* directories. Special files like pipes or devices also use inodes and have special fields in the inode structure.

Most of the functions defined in `inode_operations` are obvious because there are GNU userland programs for it as well. These include operations like:

- `link`: Creates a hard link.
- `mkdir`: Creates a directory.
- `rename`: Renames a file or directory.
- `lookup`: Searches the directory for the given name.
- `mknod`: Creates a special file like devices or pipes.
- `setattr`: Sets the attributes like `chmod`.

[1] There are more functions that are for very specific usecases such as autofs or network file systems.

- `fiemap`: An example for one of the very specific commands. It delivers information about sparse files.

10.2.3 file

The `file` struct finally represents the actual opened file that is read by a process. Therefore, it is newly created on every invocation of `open` in contrast to both inodes and dentries. Again, some of the imporant members are: the path (which includes the dentry), a counter of its users, flags (like append), the mode (whether it is opened for reading, writing, etc.), the current offset, a pointer to the address space.

- `f_path`: The path that contains the dentry
- `f_count`: A counter how many processes are accessing this file
- `f_flags`: Flags that are specified when opening the file
- `f_mode`: The access mode such as read-only
- `f_pos`: The current offset within the file
- `f_ra`: The read-ahead state which is used for reading the next parts of the file if it is read sequentially

Some of the operations on files:

- `llseek`: Sets the current position in the file to a new value.
- `read`: Reads a file at a certain position.
- `write`: Writes into a file.
- `mmap`: Maps a file to the address space.
- `fsync`: Writes cached data to the hard drive.
- `poll`: Check whether a device is readable or writable.

To remember these, one can simply think of the C functions. Functions like `seek`, `read` or `mmap` are all available to the users as system calls.

10.3 Process-specific information

There is a number of general file system information that are specific for each process. Two of those shall be presented here.

The `mnt_namespace`s are used to give each process its individual view of the file system and can be used for sandboxing purposes like *docker*, *bubblewrap* or *snaps* which recently gained some traction. It includes information about the root mount point, a list of mountpoints and a reference to the user namespace which can be used for unpriviliged mounts.

A similar struct is the `fs_struct` which includes the umask, the current working directory and the root directory and is used for traditional `chroot`.

10.4 Summary

The virtual file system is one of the core innovations of the Unix system and is also part of Linux. The abstractions make it easy for programmers to create their programs without having to worry about the underlying file systems details.

The core concepts are the dentries which are used to validate a file path, inodes which describe the metadata and the file struct itself which corresponds to a file opened by a process. There are, of course, a number of others structs like superblock and vfsmount to manage partitions, and process specific information to manage namespaces, for example.

11 Virtual Adress Space

Each process has different data it wants to store in memory.

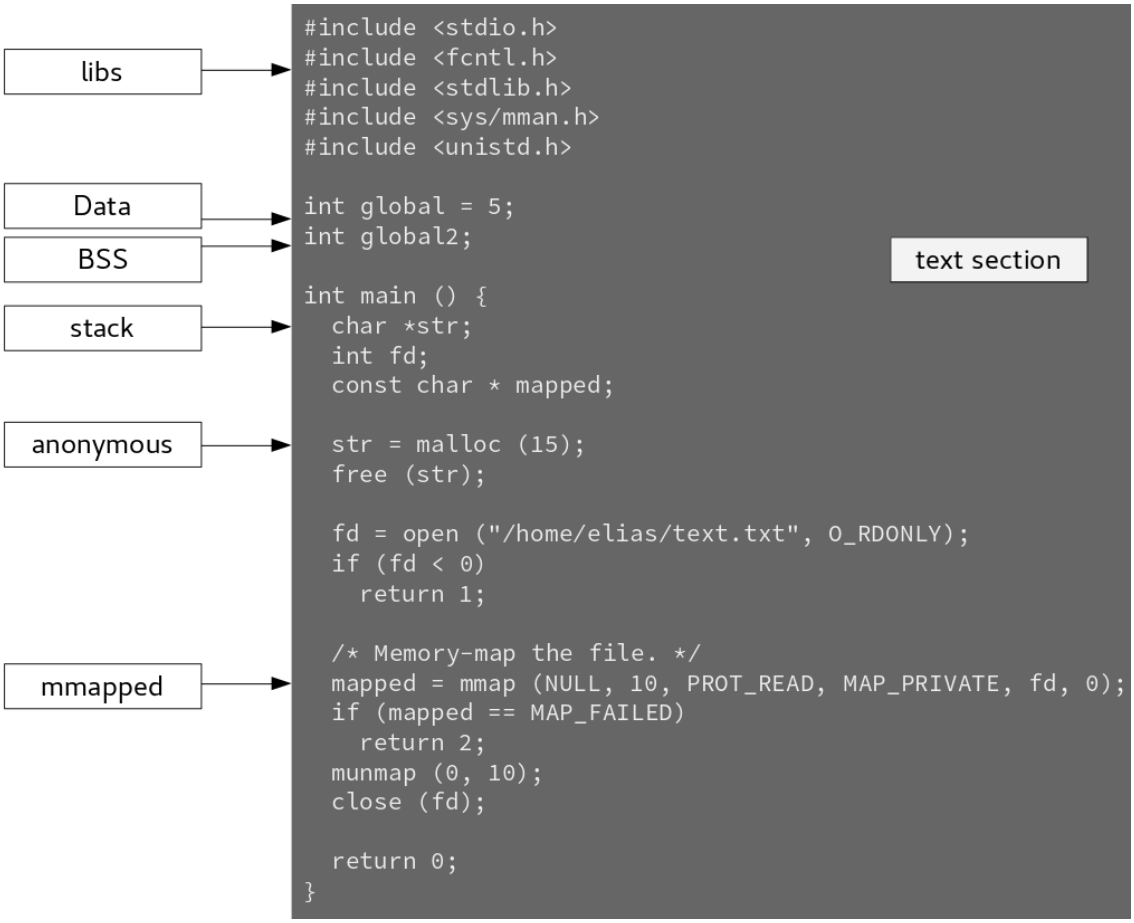


Figure 11.1: A small program with different memory sections marked.

These are shared libraries, global (un)initialized variables, the stack, the heap, the code and memory mapped files as visible in figure 11.1. Storing these areas and accessing data in it using addresses is the job of the virtual address space.

11.1 mm_struct

The `mm_struct` stores general information about the virtual memory of a process. Each process has one of those structs associated with it, except for the kernel processes. This information is used to resolve virtual addresses and check permissions. Processes that share their memory have the same `mm_struct`.

It contains:

- `mmap`: A list of virtual memory areas
- `mm_rb`: A red-black tree of virtual memory areas
- `vmacache_seqnum`: The number of the per thread cache
- `get_unmapped_area`: A function to get an free area to generate a VMA
- `mmap_base`: The start address in the virtual address space
- `pgd`: The page global directory (page tables are explained in section 11.3)
- `map_count`: The number of VMA is in this virtual address space
- `mmlist`: A list of swapped VMAs
- `start_code, end_code`: The start and end address of the code
- `start_data, end_data`: The start and end address of the initialized variables
- `start_brk, end_brk`: The start and end address of the heap
- `start_stack`: The start address of the stack
- `arg_start, arg_end`: The start and end address of the process arguments
- `core_state`: Descriptor of core dump support
- `exe_file`: The file that is executed

Most importantly, the kernel can search the red-black tree for a memory area (in logarithmic time) and it can start to resolve a virtual address with the reference to the page global directory (PGD).

11.2 Virtual Memory Areas

Virtual memory areas (VMA) describe an area in the virtual memory space. The struct is defined in `include/linux/mm.h` and contains:

- `vm_start`: Start address of the area
- `vm_end`: End address of the area
- `vm_next, vm_prev`: Pointer for the double-linked list of VMAs
- `vm_rb`: Node for the red-black tree
- `vm_mm`: The memory descriptor
- `vm_flags`: Flags whether this area is readable/writable/executable
- `anon_vma_chain`: Pointer for the list of VMAs without a file mapping
- `vm_pgoff`: Offset in the file (in case the VMA is file backed)
- `vm_file`: Pointer to file (in case the VMA is file backed)

Some of its operations include:

- `open`: Add a new area to the address space.
- `close`: Remove area.
- `mremap`: Resize an area.
- `fault`: A non-existing page was being accessed.
- `name`: Get the name of an area (e.g. for `/proc/<pid>/maps`).
- `page_mkwrite`: Read-only page becomes writable.

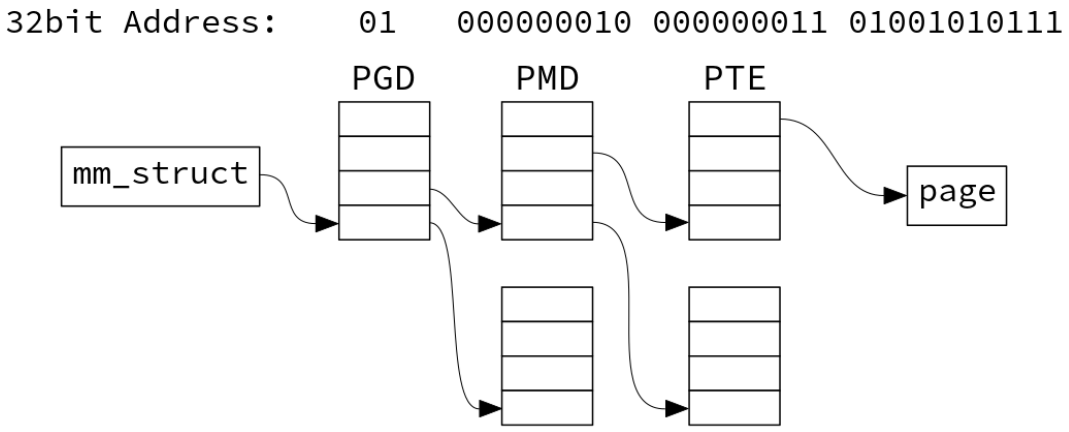


Figure 11.2: Visualization of the different levels of a page table to look up a 32bit address.

11.3 Page tables

Page tables resolve the virtual addresses that a process is using to look up the actual address that contains the value.

The `mm_struct` references the page global directory (PGD) which is the first level of the page table tree. The first part of the address then describes the offset within the PGD and this step is then repeated. The entry in the PGD references a page middle directory PMD, the address describes the offset therein, this entry references a page table entry PTE and yet again the address is used to get the offset. Finally, the actual page in the physical memory is reached, which contains the desired address (if the permissions in the PTE allow the access). This tree has the advantage that it consumes way less memory than a straigt-forward implementation like an array that maps *every* virtual address to a physical address because most processes only use a fraction of the overall memory.

As the described address resolution has to happen very often, there is a translation lookaside buffer (TLB) which acts as a cache for PTEs.

11.3.1 Recent developments

In the midst of 2017, five-level page tables were introduced and while the concept remains the same, it means that 128 PiB of virtual memory can now be addressed. This is necessary because developers start to mmap increasingly large file into memory until they bump into the kernel limits.

Another recent development is the mitigation of the so-called *Meltdown* vulnerability. To prevent access to kernel memory using a processor bug regarding speculative execution, there has been ongoing work since December 2017 for kernel page table isolation (KPTI). Kernel processes do not have a `mm_struct` and use the page tables of the userspace process that ran before to avoid additional overhead (like flushing the TLB). In recent efforts, this is a subject to change. After the KAISER patch series, there is a page table for the kernel which contains the mapping of the whole address space. When a user space process get activated, the page tables are changed and now only consist of a *shadow* page table which does not contain information about kernel memory any more (except for those addresses needed for system calls).

11.4 Summary

The two most important aspects of the virtual address space are virtual memory areas and page tables. The virtual memory areas are used to describe an area in the address space and specify its access rights. The page tables are used to resolve an address to its actual address on the hardware. Both information are accessed very frequently so they are heavily cached.

12 Block I/O (BIO)

The use of the Block I/O Layer is often compared with the basic functionality of an elevator. If an elevator wants to drive to multiple floors, it is convenient to choose a reasonable order of floors to drive to, with the objective to minimize its movement. Otherwise the elevator would drive unnecessary ways and it would take much longer. This example is an analogy for the Block I/O Layer and the management of their block devices. In this case, the elevator is the disk head. Today's storage devices might not have a head anymore. Thus this analogy is meant mainly for devices like a HDD. The goal of the Block I/O Layer is to manage block devices and the requests to them. I/O-Schedulers try to handle the requests and put them in a reasonable processing order to minimize the disk head movement.

12.1 Devices and Units

It is necessary to define different terms when talking about block I/O. First of all, we have to distinguish between a *block device* and a *character device*. A block device is a hardware device which manages its data in fixed-size chunks of data, which are also called *blocks*. Data is then accessed with (not necessarily) sequential access. The order of the blocks is not important, which in turn requires more expensive management of a block device because it must be able to navigate from one location to another on the disk. A common example for a block device is a hard disk drive. A character device on the contrary accesses its data within a sequential data stream, which it has to process in order. As a result, no expensive management is needed because there is only one position on the media which is important the current one. A keyboard is an example for a character device. Two other terms to distinguish are *blocks* and *sectors*. A sector is the smallest addressable unit of a block device. It is the fundamental unit of all block devices because it is a physical property. The size of a sector comes in various powers of two, but the most common size is 512 bytes. A block on the other hand is the smallest *logically* addressable unit of a block device. It is an abstraction of the filesystem because the filesystem can be accessed only in multiples of a block. The size of a block also comes in various powers of two, but has to be a multiple of the sector size. In theory, addressing sector-wise is possible. However in practice, the kernel does all disk I/O in blocks.

12.2 Data structures

We now take a look at the Linux kernel to see how blocks are stored and accessed in the Block I/O Layer. Once a block is stored in physical memory, it is saved in a *buffer*. Consequently, one buffer is associated with one block. In earlier kernel versions a buffer was stored in a separate *buffer head*, which was a descriptor for this buffer. It contained more information, like control information to find the exact position of the buffer on the block device. However, using the buffer head as the primary I/O unit resulted in a number of problems. First of all, the buffer head was too big and too clunky to just represent an I/O operation. Secondly, the kernel prefers to work in pages (more simple and better performance). The biggest problem is the fact that a buffer head only stored a single buffer. If a big I/O operation operated on multiple buffers, it had to be broken down to multiple buffer heads, which, again, contained information which were not relevant for the current I/O operation. The *bio* structure provides a better container for I/O operations. Today, it is the fundamental I/O unit. Every block I/O request is represented in this bio structure as a list of segments. A segment is a chunk of a buffer that is contiguous in physical memory. Hence it is possible for

```
struct bio {
    unsigned short  bi_vcnt;        /* number of bio_vecs */
    unsigned short  bi_max_vecs;    /* maximum bio_vecs possible */
    atomic_t        bi_cnt;         /* usage counter */

    struct bio_vec  *bi_io_vec;     /* bio_vec list */
}
```

Listing 12.1: Code of bio structure (excerpt) (`<linux/blk_types.h>`)

```
struct bio_vec {
    struct page     *bv_page;
    unsigned int    bv_len;
    unsigned int    bv_offset;
}
```

Listing 12.2: Code of `bio_vec` structure (excerpt) (`<linux/bvec.h>`)

the kernel to perform block I/O operations of a single buffer from multiple locations in memory. In listing 12.1 an excerpt of the bio structure from the kernel can be found.

While keeping in mind that one bio structure represents one I/O operation and it allows to operate on multiple buffers, a pointer to an array of bio vectors is used (`*bi_io_vec`). Additional parameters store the number of bio vectors associated with the current I/O operation (`bi_vcnt`) and the maximum number of bio vectors possible (`bi_max_vecs`). The parameter `bi_cnt` represents the usage counter. As soon as it hits 0, the bio struct is destroyed and the memory released.

To represent one page, the structure `bio_vec` is used. It can be found in listing 12.2. It consists of a pointer to the physical page where the buffer is located (`*bv_page`), the length of the buffer in bytes (`bv_len`) and the offset in the page where the buffer is located (`bv_offset`).

A graphical summary of the bio structure and its relation to the bio vectors can be found in Figure 12.1. It can be seen that the bio structure can represent I/O operations that consist of one or more pages in the memory. Therefore it is possible to address contiguous blocks in memory. In this case a bio vectors represents one page.

12.3 I/O Scheduler

I/O Schedulers are managing incoming requests from the Block Layer. The goal is to find an optimal order of requests, that fits best to the physical function of the drive. After that they are given to the device driver. Pending requests are stored in the *request queue*. Every element in this list is a single request. Those requests are sorted and merged by the I/O Scheduler and put into the *dispatch queue*. The dispatch queue managed by the device driver, which further processes the requests.

12.3.1 Linus Elevator

The two fundamental tasks of an I/O Scheduler are *merging* and *sorting* the requests. Here, they are explained by watching the Linus Elevator, which was the standard I/O scheduler in older kernel versions. Most of the schedulers used nowadays are build on it. The Linus Elevator performs four operations while adding a new request:

1. Merging requests that address adjacent on-disk sectors into a single request.

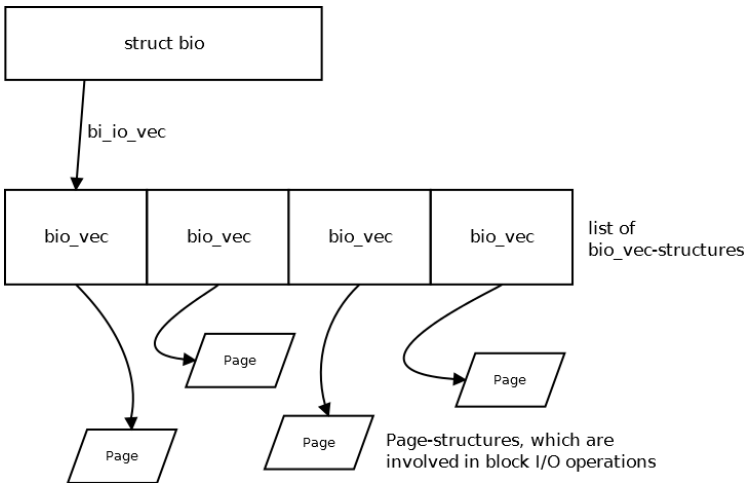


Figure 12.1: The bio structure [3]

2. Sort requests by physical location.

3. Prefer older requests in the queue to prevent starvation.

4. Otherwise the request is inserted at the tail of the queue.

The problem with the Linus Elevator is the starvation of requests in the queue. In the case of always inserting requests at a position that keeps the queue sorted by physical location, it might happen that older requests never get served. The following I/O Schedulers try to solve that problem and are used in the present kernel.

12.3.2 Deadline Scheduler

The Deadline Scheduler adds a deadline to each request. Each request is put into two queues: a queue which is sorted in terms of physical location on the disk and queue which works with the FIFO principle (in fact it is sorted by the arrival time of the requests). The FIFO queue is splitted in two queues, one for read and one for write requests. Requests from the sorted queue are added to the dispatch queue so long as a deadline of one the requests in the FIFO queues expires. If that happens, the affected requests are preferred and added to the dispatch queue. This prevents the starvation of requests effectively. Read requests have shorter deadlines than write requests. This originates in the fact that read requests occur synchronously with respect to the submitting application. The application has to wait for the read request to finish (in contrary to a write request). This results in high read latencies. Taking this into account, the Deadline I/O Scheduler is suitable for read-sensitive applications. A graphical representation of this scheduler can be found in Figure 12.2.

12.3.3 Complete Fair Queuing Scheduler

The Complete Fair Queuing Scheduler (also often referred as CFQ) is the default I/O scheduler in Linux. It creates a separate queue for each process transmitting a request. In each queue the two basic operations (sort and merge) are performed. The requests are then added to the dispatch queue in a round robin fashion: a limited number of requests per queue are added. This is completely fair for all processes as the Complete Fair Queuing Scheduler does not prefer read or write requests.

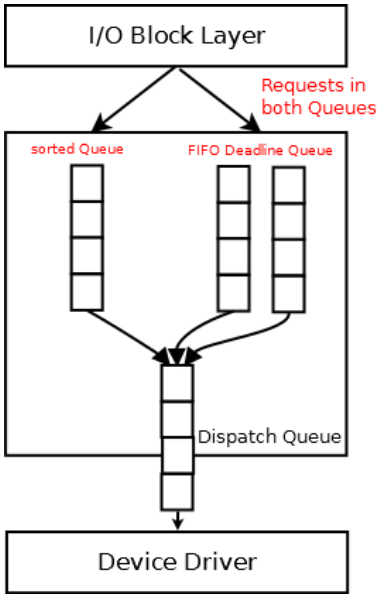


Figure 12.2: The Deadline I/O Scheduler

12.3.4 Noop Scheduler

This scheduler is limiting its operations intentionally. The only operation performed is the merging of requests that address adjacent on-disk sectors into a single request. Requests are then added in a FIFO fashion to the dispatch queue. It is suitable for devices without additional cost for head disk movement, for example random access devices like Flash Memory Cards or SSDs.

12.3.5 Budget Fair Queuing Scheduler

Since Linux 4.12, the Budget Fair Queuing Scheduler (also often referred as BFQ) is part of the kernel. It is based on the CFQ scheduler. Instead of a fixed time unit (like in CFQ), a budget is assigned to each process. The budget consists of an amount of sectors. Once a process consumed its budget, its access is deprived.

12.3.6 Kyber Scheduler

Linux 4.12 also comes with the Kyber Scheduler as the second new scheduler. The Kyber algorithm keeps the dispatch queues short. This means that the number of read/write operations send to the dispatch queue is limited. Consequentially the latency for each request is relatively small. In fact, requests with high priority are ensured to be completed quick.

13 Page Cache and Page Writeback

13.1 Basic concepts and terms

The *page cache* stores additional data in unused areas of physical memory, which is faster than the memory on hard disks. This happens everytime a read or a write operation to data media is performed for the first time. The goal of using the page cache is to minimize I/O operations on the hard disk. If the data is read again later, the cache can be used to quickly read in memory. This results in a tradeoff between using the fast physical memory as often as possible and maintain reasonable cache coherence in case of a system failure. *Page writeback* describes the process of writing back changes from the cache to the hard disk. In terms of a read system call we are talking about a *cache hit* when the requested data is in the page cache. In this case no hard disk access is necessary. If the requested data is not in the page cache, it is called a *cache miss*. That causes a hard disk I/O operation to read the requested data from it. In terms of a write system call the hard disk memory needs to be updated at some point to guarantee cache coherence. The strategy *write-through* always updates the hard disk memory as soon as data in the cache changes. This provides good cache coherence but is not using the benefit of the physical memory. *Write-Back* is the strategy used in Linux to update the hard disk memory. Written pages in the page cache are marked *dirty*. Effectively it means that these pages are not synchronized with the hard disk memory. A Writeback synchronizes the cache with the hard disk memory and makes the pages *clean* again.

13.2 The Linux Page Cache

Lets take a look at the Linux page cache implement in the Linux kernel. In the Linux page cache, every object is cached that is based on a page. The page cache is managed in the structure `address_space` (as seen in Listing 13.1). The `address_space` is associated with a kernel object, usually an inode (`*host`). It also provides a binary radix tree to perform faster search for a wanted page (`page_tree`). The total number of pages in the cache is stored in the parameter `nrpages`. `*a_ops` is a pointer to the operations table, which can be seen in Listing 13.2, where among other methods the two basic functionalities `*writepage` and `*readpage` exist. If a read request is send to the page cache, it is searched for the page in the cache with the method `find_get_page(mapping, index)`. If the page is not found, a new page is allocated in memory and added to the cache. Finally, this page is returned to the user. If a write request is send to the page cache, the written page is set to a dirty state with the method `setpagedirty(page)`. The page cache is then searched for the written page. If it is not found, an entry is allocated in memory and added to the page cache. Finally the data is written back to the hard disk memory.

```
1 struct address space {
      struct inode             *host;      /* owning inode */
      struct radix_tree_root   page_tree;  /* radix tree of all pages */
      unsigned long            nrpages;    /* total number of pages */
5
      const struct address_space_operations *a_ops    /* operations table */
  }
```

Listing 13.1: Code of address_space (excerpt) (`<linux/fs.h>`)

```
struct address_space_operations {
    int (*writepage) (struct page *page, struct writeback_control *wbc);
    int (*readpage) (struct file *, struct page *)
}
```

Listing 13.2: Code of address_space operations (excerpt) (<linux/fs.h>)

13.3 Dirty Page Writeback

When a page is written in the page cache, clean pages are overwritten. Once not enough clean pages are available, dirty pages need to be synchronized with the hard disk memory in order to be clean again. There are different strategies to decide which dirty pages are the first ones to be written back. One strategy selects the dirty pages with the oldest timestamp. It is most improbable that these will be used in the near future. Another strategy is called the *two list strategy* and puts pages in either an active list or an inactive list. Only pages in the inactive list are available for a writeback. To get into the active list a page must be read or written at least twice. Furthermore, the number of pages in the active and inactive list is balanced. In the Linux kernel, the *flusher threads* perform dirty page writeback. There are three situations where the flusher threads are activated to perform a dirty page writeback.

The first situation is about having too little free space in the physical memory. That happens if too many pages in the page cache are dirty. The limit for free space is stored in `dirty_background_ratio`. Once the limit is reached, the flusher threads are activated with the method `wakeup_flusher_threads()` to perform dirty page writeback. The dirty page writeback itself is performed within the method `bdi_writeback_all()`.

The second situation is about dirty pages in the cache that are getting too old. In case of a system crash those pages are not synchronized with the hard disk memory and are in fact lost. To prevent this, the flusher threads wake up periodically to perform a writeback. The configuration on how often the flusher threads wake up is set in `dirty_writeback_interval`. The limit how old data must be to be written out the next time a flusher thread wakes to perform periodic writeback is set in `dirty_expire_interval`

The flusher threads can also be manually activated to perform a writeback. This can be done by the system calls `sync()` or `fsync()`.

14 sysfs

Introduced with kernel version 2.6, the unified device model was an attempt to improve the up-to-then impractical device modeling in the kernel. While subsystems had their own ways of modeling their relevant devices, there was no global, kernel-wide framework.

This problem became prominent especially with the rise of mobile computers and the users' desire to be able to put those in power saving modes. In order to perform this task correctly, a hierarchical topology of the systems' devices was needed for computing the correct order of shutting down devices. For example, an USB mouse should intend to power down before its corresponding controller.

Starting from this rather practical motivation, the device model delivered a lot more features, improving the kernel organisation. A concept with a similar mission creep were `Kobjects`. Initially meant to provide reference counting for kernel structures they are now the essence of the device model and the sysfs filesystem.

Sysfs is kind of a byproduct of the new device model and allows the inspection and modification of kernel object properties from userspace. Initially implemented to facilitate debugging of the new model, it proved to be useful and therefore was kept as a means of communication between kernel and userland. It is to note that in contrast to `procfs` it is not intended to be human processable but rather facilitates machine parsing.

This chapter aims to provide a grasp of kernel data structures, the device model, sysfs and the interrelations between those concepts. For that, Kobjects and related structures will be presented at the beginning. Afterwards, this understanding will be used to comprehend the setup of the sysfs filesystem.

14.1 Data structures

To understand the sysfs filesystem and the device model, one has to know first how objects and especially devices are organised in the kernel.

14.1.1 Kobjects

Kobjects reside at the heart of this the new device model implementation. They were first introduced during the development of kernel version 2.5 to unify the handling of C structures inside the kernel. While different subsystems had to deal with different objects and representing internal structures, they shared common operations like reference counting and naming. This was the cause of the effort to develop an overarching, kernel wide object data structure. Those were called Kobjects. To sum up, they were the proposed solution to the following issues:

- Code duplication
- Managing and maintaining lists of objects
- Locking of sets
- Providing an interface to userspace.

How those goals were achieved will be examined in the next sections.

The following listing shows the resulting `kobject` structure:

```
struct kobject {
    const char      *name;
    struct list_head    entry;
    struct kobject  *parent;
    struct kset     *kset;
    struct kobj_type    *ktype;
    struct kernfs_node  *sd; /* sysfs directory entry */
    struct kref     kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

Listing 14.1: Kobject structure, ¡linux/kobject.h¿

Going through this structure top-down its most important entries are explained below.

`name` serves the simple purpose of naming the object.

`list_entry` is used when the Kobject is part of a list.

`parent` is a pointer to another Kobject. This allows building a hierarchy.

`kset` defines the `kset` this Kobject is part of.

`ktype` specifies the `kobj_type` of this Kobject.

`sd` represents a directory entry in `sysfs`. This will be important later in the chapter.

`kref` implements reference counting for this object.

Usage The abstraction with Kobjects allows for any embedding structure to have the properties of Kobjects. One can think of them as a C equivalent of base classes in object-oriented languages. In this notion, they are the most abstract class from which other classes are derived, thus lending them its properties. Since the C language has no built-in concept of derivation, Kobjects are embedded. For a real world usage example, the following listing shows the `cdev` structure representing a character device.

```
struct cdev {
      struct kobject kobj;
      struct module *owner;
      const struct file_operations *ops;
      struct list_head list;
      dev_t dev;
      unsigned int count;
}
```

Listing 14.2: character device structure, ¡linux/cdev.h¿

This shows, how a Kobject is embedded into a more specific structure which then uses its properties. So, finding the Kobject of a given structure is as easy as accessing the `kobj` field. But often, it is the other way around. Code that works on the system wide Kobject hierarchy wants to get from the Kobject to the embedding structure. An often used workaround for this problem is assuming the `kobj` field is placed at

the beginning of the structure and trying to access it this way. The cleaner method though is obtaining the structure using the `container_of` macro. It is rare that a standalone Kobject is created in the kernel.

Table 14.1: Kobject operations

Method	Function
kobject_get, kobject_put	Incrementing and decrementing the reference counter.
kobject_register, kobject_unregister	Add the object into a hierarchy. It gets added to the set of the provided parent element if existing. It also gets exported to sysfs.
kobject_init	Creates a new Kobject and set the reference counter to zero.
kobject_add	Initializes a new Kobject and adds it to sysfs.
kobject_cleanup	Releases the allocated resources.

Table 14.1 shows the most common operations while interacting with Kobjects. It is to note that a sole call to `kobject_init` is not sufficient for initializing a Kobject. Additionally the name (and with that the name of the sysfs representation) has to be set with `kobject_set_name`. More complex methods like `kobject_add` and `kobject_register` already take care of that step.

Since Kobjects are mostly embedded in higher layer structures (e.g a character block device structure) not even driver writers have to interact with them directly most of the time. They are rather administered by the corresponding driver subsystem. Also, no structure should ever embed more than one Kobject. This would contradict the philosophy behind them.

sectionKtypes Another important point is what happens when the reference counter of a Kobject reaches zero, hence the Kobject and its embedding structure are no longer used. This is important because through the sysfs export, the code creating a Kobject has no control over how long it is used since any userspace program can still hold a reference to it. If the object is not used anymore, the creating code has to be notified that the Kobjects resources can be freed. This notification is sent through the `release` method of a kobject.

It is critical that a Kobject and its embedding structure persist as long as it is used hence as long as the reference counter is greater than zero. The `release` method however is not part of the Kobject itself but inside a structure called `kobj_type`, often abbreviated as Ktype.

```
1 struct kobj_type {
  void (*release)(struct kobject *kobj);
  const struct sysfs_ops *sysfs_ops;
  struct attribute **default_attrs;
5 const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
  const void *(*namespace)(struct kobject *kobj);
};
```

Listing 14.3: Ktype structure, ¡linux/kobject.h¿

Going through this structure top-down, the most important entries of this structure will be explained.

`release` gets called when the reference counter gets to zero.

`sysfs_ops` are function pointers to the functions that get called when an exported attribute gets read from or written to.

`default_attrs` is a set of standard attributes Kobject of this Ktype exports to sysfs.

Usage Every Kobject is of a specific Ktype while Kobjects with the same Ktype share a set of operations and properties. The Ktype a Kobject belongs to can be found at two different locations. If the Kobject is part of a `Kset`, the Ktype is determined by that structure. If this is not the case the Ktype can be found in the Kobjects `ktype` field. The macro

```
1 struct kobj_type *get_ktype(struct kobject *kobj);
```

returns the pointer to a Kobjects Ktype structure.

The usage of Ktypes is mainly geared towards the sysfs representation but Ktypes also include the `release` function, which is called internally when the Kobject is ready to be destroyed.

14.1.2 Ksets

In contrast to Ktypes, which define common types of Kobjects, the main function of Ksets is aggregation of Kobjects. With their help, Kobjects can be grouped into sets of the same Ktype. Ksets are the second method to provide structure and hierarchy to kernel data structures besides Kobjects.

For example, one could group all devices which are classified as PCI device into one Kset and hence have them grouped in one sysfs directory. While grouping Kobjects, Ksets embed a Kobject themselves, and are, Unlike Kobjects, Ksets always appear in the sysfs directory structure.

```
1 struct kset {
      struct list_head list;
      spinlock_t list_lock;
      struct kobject kobj;
5     const struct kset_uevent_ops *uevent_ops;
  }
```

Listing 14.4: Kset structure, ¡linux/kobject.h¿

`list_head` is used when the Kobject is part of a list.

`parent` is a pointer to another Kobject. This allows building a hierarchy.

`list_lock` provides a spinlock for the set.

`kobj` is the Kobject the Kset embeds since Ksets themselves are Kobjects too.

`uevent_ops` defines which events are sent to userspace. This is useful for example for hotplugging events.

Usage The children of a Kset are kept in a linked list. Those Kobjects refer back to the containing Kset via their `kset` field. The initialization and setup routines are similar to those of Kobjects:

```
1 void kset_init(struct kset *kset);
  int kset_add(struct kset *kset);
  int kset_register(struct kset *kset);
  void kset_unregister(struct kset *kset);
```

Internally those call the corresponding routines of the embedded Kobjects.

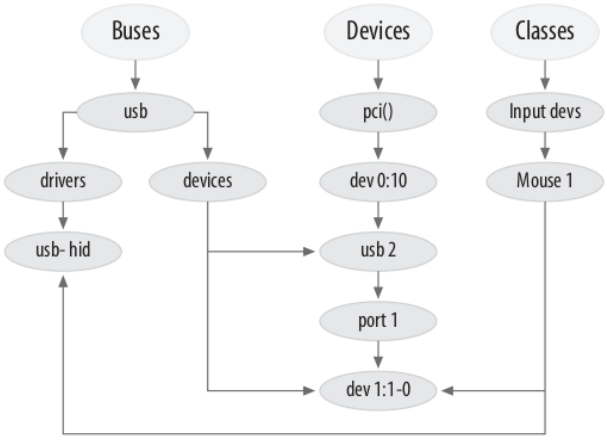


Figure 14.1: Device model visualization, [4]

14.1.3 Subsystems

`subsystem` represents a high-level part of the kernel as a whole. In essence a subsystem is just a Kset with an additional semaphore:

```
1 struct subsystem {
      struct kset kset;
      struct rw_semaphore rwsem;
  }
```

Subsystems usually appear at the top of the sysfs directory. Every Kset belongs to a subsystem, thus subsystems provide hierarchy placement for Ksets.

14.2 Checkpoint

Data structures This chapter's beginning discussed the motivation to unify the base of different kernel data structures and to represent the systems devices in a kernel-wide hierarchy. By the use of `Kobjects`, `Ksets`, `Ktypes` and `subsystems` both of this goals can be achieved. Kernel objects can be organised in hierarchies and groups through this structures. As an addition, the objects can be grouped by the means of shared operations through `Ktypes`. This can be used by the device related subsystems in the kernel to generate a kernel wide device model, hence a global relationship structure between its objects.

Figure 14.1 on page 79 shows an excerpt of a typical device model structure. The central device of this excerpt is an USB mouse which is connected to a specific bus, belongs into the class "Input devs" and is managed by a specific USB controller.

Bridge to sysfs While inspecting the Kobject data structure and its relatives, there already were a lot of connections to sysfs visible. As stated, sysfs maps the kernel-internal Kobject structure on a filesystem. Therefore, it provides a user space view of the interrelations between kernel objects.

On a modern linux system, it thereby has a least 10 top level directories which map the kernel subsystems. Buses, drivers, devices and classes are the main kernel components using Kobjects and therefore account for most of the sysfs content.

The next sections will illustrate how sysfs is built from the kernel internal Kobject hierarchies, though the complexity of the topic and the number of kernel components involved cause it to remain an overview to give a grasp of the mechanisms at work.

14.3 Mapping Kobjects

This section describes the mapping of `Kobjects`, `Ktypes` and `Ksets` onto the components of the sysfs filesystem, namely `directories` and `files`.

14.3.1 Directories

Since every Kobject has a corresponding `sysfs_dirent` structure embedded, mapping the Kobject hierarchy to a filesystem becomes a trivial task.

The following listing shows an example sysfs toplevel. The `device` directory shows the kernel wide device topology, hence the physical connection hierarchy of connected devices, buses and controllers. The other top level directories are representations of the different kernel `subsystems`.

```
dr-xr-xr-x  13 root root    0 20. Mär 08:27 ./
drwxr-xr-x  23 root root 4096 15. Mär 19:34 ../
drwxr-xr-x   2 root root    0 20. Mär 08:27 block/
drwxr-xr-x  33 root root    0 20. Mär 08:27 bus/
drwxr-xr-x  57 root root    0 20. Mär 08:27 class/
drwxr-xr-x   4 root root    0 20. Mär 08:27 dev/
drwxr-xr-x  15 root root    0 20. Mär 08:27 devices/
drwxr-xr-x   6 root root    0 20. Mär 08:27 firmware/
drwxr-xr-x   7 root root    0 20. Mär 08:27 fs/
drwxr-xr-x   2 root root    0 20. Mär 08:27 hypervisor/
drwxr-xr-x  12 root root    0 20. Mär 08:27 kernel/
drwxr-xr-x 129 root root    0 20. Mär 08:27 module/
drwxr-xr-x   2 root root    0 20. Mär 12:42 power/
```

Kobjects are not automatically added to sysfs at creation time though. For this purpose, either `kobject_add` has to be called by the code handling the embedding structure, e.g. the device driver, or the object has to be made part of a Kset of a subsystem, which by default calls the relevant functions. It is to note that most structures that embed a Kobject are part of a hierarchy in their environment and as such part of Ksets or subsystems, so that the latter is much more common.

A Kobject added that is sysfs has three possible places where it may end up in:

- The directory corresponding to the Kset it is part of
- The directory of its parent
- the sysfs root directory

These locations are tried in the listed order such that only a Kobject which is not part of a Kset and has no parent Kobject gets placed in the root directory.

14.3.2 Files

The last section showed how the directory structure comes about. Since the stated goals of sysfs were debugging in the beginning and userspace communication later, a mapping of the kernel structure hierarchy is useful but not sufficient. Also, one of the main features advertised for sysfs was the possibility to inspect and change kernel object parameters. So needs to be a data flow between kernel objects and userspace. This data flow is provided by files and attributes respectively. In addition to a their position in the hierarchies, Kobjects export `attributes`. Those enable the export of relevant information and also the import of data from userspace. As shown in previous sections, an initial set of standard attributes is defined by each Kobjects corresponding Ktype. On top of that Kobjects can define custom attributes.

An attribute has the following format:

```
1 struct attribute {
      const char  *name;
      umode_t mode;
  }
```

Listing 14.5: attribute structure, ¡linux/sysfs.h¿

`name` represents the attribute name and the corresponding file name in the sysfs representation.

`mode` represents the access flags for the corresponding file in the sysfs representation.

Attributes should be ASCII text files and hence be readable and writable with a standard text editor. If possible, one file should represent only one attribute. Since this very inefficient sometimes, arrays of values from the same type can be grouped into one attribute/file.

Custom attributes Most of the times the default attributes provided by the Ktype are sufficient. Subsystems that aim to expose attributes in sysfs should introduce their own attribute structures on top of this basic one and provide methods for reading and writing. If one wants to add custom attributes, thus custom files in the sysfs directory, it is as easy as calling

```
1 int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
```

This method takes the containing Kobject and an attribute structure and creates the corresponding sysfs representation. Removing attributes from sysfs is done in the following way:

```
1 int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
```

It is possible though, that a program from userspace still has an open file descriptor to the file so that it can still interact with the attribute.

An attribute does not contain any methods for reading and writing activity on their surrogate files.

14.3.3 Interaction

The last section showed how Kobjects, Ksets and subsystems are mapped onto the filesystem and how Kobjects can export attributes which can be read from and written to. There are attributes provided by the Ktype and custom ones which can be added to any Kobject. For both of those cases, there have to be ways to handle reading and writing activities on those files. This file interaction methods are the way of communication between the kernel objects and the user space entity which accesses the sysfs entries. As a consequence any subsystem of the kernel that exports attributes and wants to react to interaction with them in a meaningful way has to provide callback functions so that read and write accesses can be forwarded to them.

Since Ktypes define the default attributes of Kobjects and the behaviour when their attributes are being written to or read from, those functions can be found there, namely in the `sysfs_ops` member. `sysfs_ops` is of type `kernfs_ops` in which the `show` and `store` methods are defined.

```
1 ct sysfs_ops {
ssize_t (*show)(struct kobject *, struct attribute *, char *);
ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
```

These two functions get called when the attribute is read or written to. When an attribute is read from user space, the `show` method is called and has to fill the buffer with the value in question. Likewise, `store` processes the value written to the file (stored in the buffer). Thereby the same `show` and `store` methods are used for all attributes associated with a given Kobject, custom and default ones.

This is the mechanims which the kernel data structures can use to export data to and import data from userspace through sysfs.

It's important to note that `store` provides a method for transmitting arbitrary data from user space to kernel space and therefore has to be treated with caution. The subsystem defining the Ktype of the Kobject in question implements them.

14.4 Summary

`Kobjects` are at the core of the device model. They provide hierarchy, namesetting and reference counting themselves and grouping through the Kset structure. They were first thought of as a simple reference counting mechanism but now carry more responsibilities. Among other uses, they are the essential element of the kernel internal device model and the sysfs filesystem. As a side effect, their introduction reduced the amount of duplicated code in the kernel.

On this foundation the `device model` is able to fulfill its initial proposal: Contributing a complete topological overview of the systems devices for the means of power management.

`Sysfs` was initally built on top of that as an offshoot, in its first intention as a debugging tool for the device model development. It maps the kernel-internal object hierarchy as a filesystem into userland. Replacing its initial intention it now serves the purpose of enabling communication between kernel objects and the userspace through attributes which are exported as files and can be read from and written to. The directory structure in sysfs is determined by subsystems, Ksets and Kobjects and their hierarchy and relations whereas the `device` subdirectory represents the physical device topology, which was the main goal of the new device model in kernel version 2.6.

15 Modules

The Linux kernel is monolithic. This means that the whole operating system functionality resides in the kernel, which has performance advantages at the expense of modularity and kernel image slimness.

To lessen those disadvantages, the concept of modules was introduced. It enables dynamic loading and unloading of functionality into the kernel at runtime. This mechanism facilitates especially the loading of device drivers when supported devices are plugged in. So linux distributions are able to precompile a lot of drivers without bloating the kernel binary size and those hotplugged devices' drivers can then be loaded without rebooting the system.

Modules invalidate many of the arguments regarding the advantages of microkernels compared to monolithic kernels. The kernel also allows the use of binary only modules for which the manufacturer does not provide a source code for economic reasons. Their use would not be possible otherwise since the Linux kernel is available under the GNU General Public License and hence its code is always open. Modules pave the way for using those binary drivers without harming the license. Though this is subject to restrictions (see paragraphs `kernel_tainting` and and `kernel_exported_interfaces`). Figure 15.1 on page 84 shows a high level view on the role of modules.

To illustrate all aspects of modules, this chapter will go through a modules life cycle from coding it to loading it into the kernel. For that, it will begin with the format of modules on the source code layer. Afterwards, it will be shown how to compile a module using the Kbuild system and at last mechanisms and tools for loading modules will be discussed.

15.1 Source code

In this section, the coding step of modules will be further examined. On the source code layer, modules are a special kind of C program that can consist of one or more source files. Since there is no libc in kernel space, modules can only use kernel-internal functions. Not all kernel functions are visible for modules though.

15.1.1 Kernel Exported Interfaces

The *Kernel Exported Interfaces* are the collection of methods visible for modules linked to the kernel. Provider of such methods can be any source code in the kernel or other modules.

To make a functionality known to other kernel code and modules, the `EXPORT()` macro has to be called on it. There is also the possibility to restrict access to the exported functions to modules which are published under the GPL license. If this is the plan, `EXPORT_GPL()` has to be used for exporting.

15.1.2 Format

So, how do these source files look like? Unlike a normal C application, modules do not have a main method. Instead, they provide entry and exit methods which are called when the module is loaded or unloaded. Table 15.1 shows macros that have to be implemented for the module to compile whereas the last two are optional.

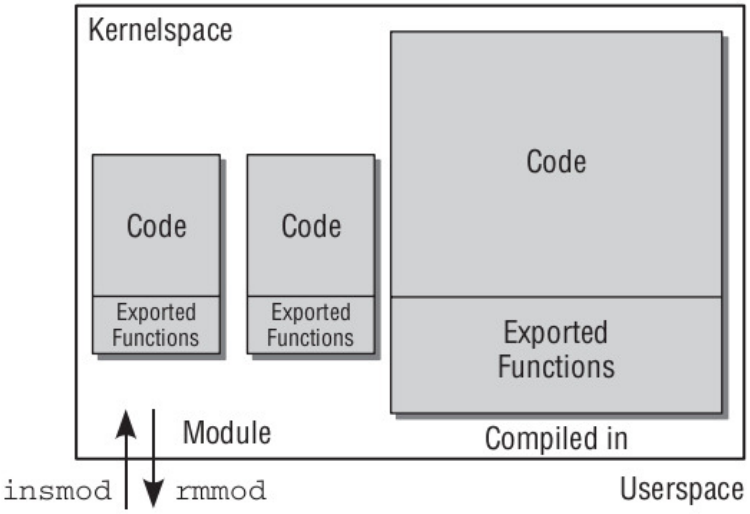


Figure 15.1: Modules high-level view, [3]

Table 15.1: Module macros

Functionality	Macro to implement
Module entry method	module_init
Module exit method	module_exit
Module license	MODULE_LICENSE
Module author	MODULE_AUTHOR
Module version	MODULE_VERSION

While a normal application is built for a specific use case, which it fulfills and then usually exits, modules follow a different approach. Most of the times they offer a specific functionality which then can be used by other code. This design difference leads to them not having a typical main method but only entry and exit methods in addition to the functionality they provide. It is possible to pass parameters to modules at loading time which can be used in its methods.

15.1.3 Providing parameters

The existence of possible parameters has to be announced in the source through a macro. There are some variations which are shown in the following list.

- `module_param(name, type, perm)`
- `module_param_named(name, type, perm)`
- `module_param_array(name, type, perm)`
- `module_param_array_named(name, type, perm)`
- `module_param_string(name, type, perm)`

`module_param` is the most basic one. Its parameters specify the name of the module parameters, its data type and the file permissions if its exported through sysfs.

It is to note that in this variant the name of the parameter as it is presented to the user is the same as the variable name which has to be used in the code. The variable has to be declared in the code before the macro

is called. The variants with the "_named" suffix aim at distinguishing these two names through providing seperate arguments to the macro. The following snippet declares and registers a module parameter of type boolean called `print_greeting_message`.

```
bool print_greeting_message = false;
module_param(print_greeting_message, bool, 0644);
```

It will be also be used in the following example module.

15.1.4 Example

The following listing contains a complete example module source code which respects all mentioned rules and constraints and is ready to be compiled. The module will output a greeting message when it is loaded into the kernel and a farewell message once it's unloaded. Additionally it uses the `current` struct from `linux/sched.h` to determine the name and process id of the loading process.

The main components of the code will be explained below the listing.

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Moritz");
MODULE_DESCRIPTION("Presentation␣example␣module");

bool print_greeting_message = false;
module_param(print_greeting_message, bool, 0644);

static int hello_init(void)
{
    if (print_greeting_message)
    {
        printk(KERN_ALERT "Hello␣AKSI,␣this␣is␣module.\n");
        printk(KERN_INFO "The␣process␣is␣\"%s\"␣(pid␣%i)\n",
                         current->comm, current->pid);
    }
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Bye,␣it␣was␣fun!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

`Lines 1-4` include kernel header files.

In `lines 5-8` the module description macros are used.

`Line 9 and 10` introduce the module parameter `print_greeting_message`. It first gets declared and afterwards registered with the `module_param` macro. m Since `lines 31 and 32` register the entry and exit points `hello_init` and `hello_exit` through the `module_init` and `module_exit` macros, hence the `hello_init` method is called when the module is loaded. m Its execution starts in `line 15` where the method checks whether the `print_greeting_message` parameter is set. If this is the case, it outputs a greeting message and additionally prints the name and PID of the current process (which because of the design will always be the loading program). m On unloading the module prints a farewell message (`lines 24-27`).

For communicating the `printk` method is used which is part of the `Kernel Exported Interfaces`. The first parameter determines the severity of the message.

tionCompilation In this step there is working module source code that is compile-ready and satisfies all demands. In the following sections the mechanisms for compiling kernel modules will be presented. Thereby, one has to rely on `Kbuild`, the kernel build system.

Kbuild The `Kbuild` system is shipped with the kernel source and consists of four main components:

- `Configuration symbols` are variables that can be evaluated inside the kernel code and inside of makefiles.
- `Kconfig files` are used to define those configuration symbols. They reside at every kernel source hierarchy level and always source their successors, so that there is one resulting top level Kconfig file containing all defined configuration symbols.
- The `.config file` resides at the top level of the source and file aggregates all configuration symbols' settings for a compilation run.
- GNU make

Kconfig format The following listing shows an example Kconfig file for creating a config option called `CONFIG_my_printer`. It should be placed in the Kconfig file of the directory where the module was placed.

```
1 config my_printer
      tristate "Support␣for␣my_printer␣model!"
      default n
      help
5         If Y, this printer driver is compiled into the kernel.
          If you say N, the driver is not compiled.
          If you say M, the driver is compiled as a module named my_printer.ko
```

There are some keywords here that require explaining. The first line just names the configuration symbol. In the second line, there is the keyword `tristate` and a short description of the models purpose. `Tristate` signals that the module can either be

- compiled directly into the kernel,
- not be included at all or
- be included as a module.

The `default` variable just sets the default value. With `help` one sets the description which is shown in various graphical kernel configuration tools.

15.1.5 In-tree compilation

This refers to the compilation of the module as part of the kernel. Hereby, the Kbuild systems gets used to its full extent.

The module source has to be placed inside the kernel source tree and made known through a makefile entry. Device drivers for example are placed into the `/drivers` directory. If the module includes more than a couple source files, it is recommended to place them in a separate directory. Assuming the files get put directly in the fitting directory without creating a subdirectory, the following line needs to be added to the directory's makefile.

```
1 obj-m += my_printer.o
```

If the module consists of multiple source files, the following syntax is applied. The `my_printer` module consists of the files `my_printer_main.c` and `my_printer_helper.c`

```
1 obj-m := my_printer.o
my_printer-objs := my_printer_main.c my_printer_helper.o
```

In the case that an own module directory is created, the following line has to be added to the already-existing higher-level source directory. Assuming the new directory is called `my_printer_directory`.

```
1 obj-m += my_printer_directory/
```

This leads to Kbuild descending into the new directory.

Conditional compiling If one wants a module to compile depending on a configuration symbol, the following syntax can be used:

```
1 obj-$(MY_COMPILING_CONDITION) += my_printer_directory/
```

15.1.6 External compilation

When compiling a module externally, its source is stored at an arbitrary place in the filesystem. Regardless, the Kbuild build system has to be invoked in order to produce a valid module file. Hence, syntax similar to the in-tree-compilation will be used while pointing GNU make to the kernel makefiles.

```
1 obj-m := my_printer.o
```

The syntax for multiple source files is the same as the in-tree variant. Now `make` has to know where to find the build rules. So the proper way to call it is

```
1 make -C /kernel/source/location/ M=\$PWD modules
```

This tells `make` where to look for Kbuild and which build rules do apply. The `M` parameter tells it to look for the target module.

Kernel tainting Compiling externally makes the module an out-of-tree module. If such a module is loaded, the kernel becomes `tainted`. This means that it is in a state not supported by the community, independently of how it got tainted. Hence, most kernel developer will ignore bug reports where a tainted kernel is in play. This feature was implemented to identify situations where the ability to troubleshoot kernel problems is reduced.

15.2 Loading modules

After the compilation step the binary modules can be loaded. The interface between userspace and the kernel with respect to loading modules consists of the following two syscalls:

- `init_module` - This inserts a module into the kernel. The only thing that has to be provided is the modules binary data.
- `delete_module` - This unloads a module from the kernel which is only possible if the module is no longer in use.

Two programs implementing this functionality will be shown in the following paragraphs. Dissecting how these tools perform this task in detail lies beyond the scope of this section.

insmod `insmod` is a basic tool which only handles this one task: Loading modules into the kernel. Thereby, it does not check if all module dependencies are loaded too and does not provide error checking routines. It gets called in the following way.

```
1 insmod my_printer.ko [module parameters]
```

And for unloading it:

```
1 rmmod my_printer.ko
```

Both commands have to be used as root.

modprobe `modprobe` is the more sophisticated module loading tool. It is able to resolve dependencies while loading by additionally loading modules the target module depends on. This will be covered in the following section. The syntax for loading a module with modprobe is:

```
1 modprobe my_printer.ko [module parameters]
```

To unload a module use:

```
1 modprobe -r my_printer.ko
```

In contrast to `rmmod`, `modprobe` also removes modules on which the module depends if they are unused. It is to note that for actually loading and unloading modules, `modprobe` uses `insmod` and `rmmod` internally.

15.2.1 Dependencies

It may be the case that a module requires functionality from another module which is a situation the `modprobe` tool is able to resolve. It does so by relying on the `depmod` utility. This tool offers two mechanisms.

On the one hand, it creates a list of all kernel modules known to the system and their used functions.

On the other hand, it maintains a list of all functions provided by known modules.

It then matches those two and creates a dependency list which is usually stored in `/lib/kernel_version/modules.dep`. In most Linux distributions depmod gets invoked automatically at boot or after loading a couple of modules. As a consequence the `modprobe` tool can use an up-to-date dependency list.

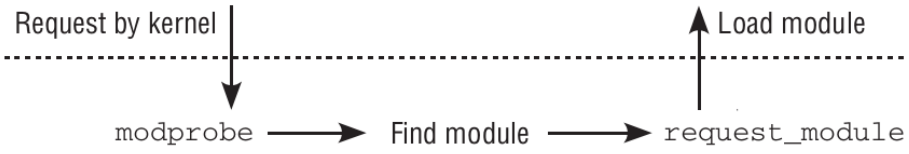


Figure 15.2: Kernel module request, [4]

15.2.2 Loading from inside the kernel

Sometimes the situation occurs where the kernel wants to load a module itself. For example, a block device with a special filesystem shall be mounted. For this task it needs help from userspace since dependency resolution and locating of the modules binary file is easier there. The kernel then uses the function `request_module` as depicted in figure 15.2.

But how does the kernel know which module to load for which device? The answer is a database which is attached to any module and depicts, which devices the module supports.

The identification of devices considers interface types, manufacturer IDs and device names. This information is provided by `module aliases` in the module source code. The following example shows the aliases of the module responsible for RAID support.

```
1 MODULE_ALIAS("md-personality-4"); /* RAID5 */
  MODULE_ALIAS("md-raid5");
  MODULE_ALIAS("md-raid4");
  MODULE_ALIAS("md-level-5");
5 MODULE_ALIAS("md-level-4");
  MODULE_ALIAS("md-personality-8"); /* RAID6 */
  MODULE_ALIAS("md-raid6");
  MODULE_ALIAS("md-level-6");
```

Listing 15.1: Module aliases, `<linux/drivers/md/raid5.c>`

15.3 Summary

This chapter illustrated the way of a kernel module from source code to loading it into the kernel. At the beginning the concept of modules as a way of dynamically loading of functionality in the context of a monolithic kernel was introduced.

In the following sections the different facets of developing a kernel module were shown. An example module was presented and relevant macros and constraints for the creation of modules were highlighted. Afterwards the module compilation and loading steps were treated. In this process, it was distinguished between external and in-tree compilation and afterwards, two different ways for loading and unloading modules were demonstrated. One, `modprobe`, more sophisticated than the others, `insmod` and `rmmod`.

In a last step, a short overview on module dependency handling was given.

Bibliography

[1] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. 2007.

[2] M. Tim Jones. Inside the linux 2.6 completely fair scheduler, December 2009. Accessed: 2018-03-13.

[3] Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005.

[4] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, 2008.

List of Figures