Linux Kernel: monitoring the scheduler by trace_sched* events

Marco Perronet

June 23, 2019

Contents

List of Figures

Abstract

This thesis is about process scheduling and its implementation in the Linux kernel. Moreover, it covers other the kernel architecture and the event tracing, since they are a crucial component needed for kernel debugging. The focus is on illustrating the scheduler implementation in Linux kernel version 4.20.13 (released on 27–02–2019), as well as documenting scheduling-related events. In fact, both the kernel and the event tracing infrastructure are poorly documented. Since the kernel and the tracing support are tightly related to each other, it is challenging to document the events without understanding how the scheduler works. Hence, most of the events are mentioned and explained on the fly, while discussing kernel concepts.

Examples of code are always provided, each piece of theory will have its implementation counterpart shown and explained. Most of the concepts are very easy to visualize once the code is split into its most significant parts and then dissected line by line; however, this is not an internals manual for development. This thesis should be considered a guide. It's meant for people who are curious about the inner workings of the kernel but have never looked too much into it, its goal is to give interesting insights about the architecture of operating systems. The only prerequisite is to have some experience with C and know the very basics of GNU/Linux and operating systems.

All references to the source code are from kernel version 4.20.13, the latest stable version at the time of writing. When discussing architecture dependent code it will be assumed that the architecture is x86. In the code, every comment spanning over multiple rows (`/*...*/`) is written by the kernel developers, my comments will always be inline (`//`).

In the first part I will give a brief overview of the kernel and then explain some basic scheduling concepts.
. . . (will write this at the end) . . .
Here is a useful tool to quickly look up mentioned kernel functions for yourself `elixir.bootlin.com/linux/v4.20.13/source`. Another alternative is to download the whole source code from `kernel.org`, which is needed if you want to compile and load it.

Chapter 1

Basics of Linux kernel

1.1 Operating Systems

The operating system (OS) comprises the software intended to manage the hardware resources and the *application software*, which performs specific, high-level tasks. The application software, which is the larger part of the OS, is made of utility programs and any other software with which the user interacts directly. These programs are not part of the core OS. Rather, they are necessary to do anything useful. The operating system acts as an intermediary between the user and the machine by abstracting away the hardware, which makes interaction easier: this is why almost every computer runs an operating system.

It can be argued that the OS is not strictly necessary because it's possible to execute a program without loading an OS: this is referred as *bare metal programming* which is common in small size embedded systems. Because there is no operating system (which means no file system, memory management or any useful application such as compilers), programs cannot be written on the system itself. Instead, the program is written on another machine with an operating system, then compiled with a cross-compiler, which compiles for a target architecture different from the one it is running on. Finally, the compiled binaries are loaded at boot time on the target embedded system. This is the opposite of what we are used to do on our laptops/desktops: to be able to reprogram the machine as it is running, by writing and compiling our program with *application software* designed to edit text and compile code. Thus, an operating system greatly simplifies interaction with the machine by offering a platform for the user and, at a higher level, by making general-purpose computing possible.

Windows, MacOS, iOS, Android... Most of us are familiar with these operating systems. Besides the platform on which they run, they are all general-purpose and their goal is the same. What really changes among them is the architecture and philosophy in their design. At a macroscopic level they differ in kernel design approach (monolithic kernel vs. hybrid kernel). This is explained later in section

1.3. At a microscopic level, there is literally not much to see because the code of most OSes is closed, so it's impossible to see the implementation differences with Linux. This leads us to one of the peculiarities of Linux: it's completely open source and community developed. Besides ethical matters, which are not discussed here, this means that it's possible to study the code and get a full understanding of operating systems. In fact, before Linux, there was no way to see how operating systems work in practice. The only option was to study them from textbooks in order to implement your own kernel, which is exactly what Linus Torvalds did.

As stated earlier, a key component of an OS is "the software intended to manage the hardware resources": this is what we refer as the *kernel*. Dennis Ritchie, among the inventors of Unix and C, also called it the "Operating system proper"[1], which most likely means "The component that is the actual operating system". On the one hand this definition makes sense, because the low level tasks performed by the kernel are essential (and also because it's the most difficult component to develop). But on the other, without application software the kernel is useless. In such scenario, the kernel is loaded at boot, then it initializes and starts running, and then there is nothing but a black screen because there is no other program to start. It's clear that the kernel is not an operating system by itself, but what Dennis meant is that when we think about the core architecture of an OS, we think about the kernel. An engine is indeed useless without the rest of the car, but does that make the other components as important as the engine, where all the complexity resides? Despite the application software being the largest part of the OS; it is within the kernel that the hardest engineering challenges are found, which makes it the most interesting—and difficult—part to understand and analyze.

1.2 A general overview

The kernel's job is to manage hardware resources, which means handling all interactions with the CPUs, the memory hierarchy and the I/O devices. More specifically, the kernel needs to respond to I/O requests, manage memory allocation and decide how the CPU time is shared among the demanding processes. To achieve this, it has access to all resources in the system, which is needed to make the most out of the hardware. Its performance is what makes the difference between a fast or a slow operating system. This critical role requires a protection mechanism to ensure the stability and the security of the whole system. This is achieved by separating kernel code and user application code. In practice, depending on the configuration settings at compile time, what happens is:

1. The kernel binary image is loaded in RAM in a memory area which can start from a low or high address.
2. A predefined slice of RAM next to that memory area is reserved to the kernel.
3. The remaining part of the memory is accessible to the user.

These two portions of the address space are called kernel space and user space. The former is a reserved area dedicated to critical system tasks and it's protected from user access, the latter is the area where system utilities and user programs run. This memory partitioning makes sure that kernel and user data do not interfere with each other. Also, it is a security measure to prevent that a malfunctioning or malicious user program may affect the entire system.

1.2.1 System calls

By extension of this design, the interaction with the user space is regulated with a privilege system. Each process can run either in user mode or kernel mode. Processes running in user mode can access privileged kernel functionalities through special gates in a predefined and controlled manner. These gates are implemented as functions called *system calls*, which serve as APIs between user and kernel space. When a user process performs a system call

1. it temporarily executes in kernel mode,
2. it performs tasks that require a high privilege, and finally
3. it switches back to low privilege.

This mechanism exploits the availability of hardware functionalities. For example, in the x86 architecture 2 bits in the code selector (`cs`) register indicate the current privilege level (CPL) of the program that is running on the CPU. This value is 0 or 3, respectively, for kernel and user mode and each system call changes this value temporarily.

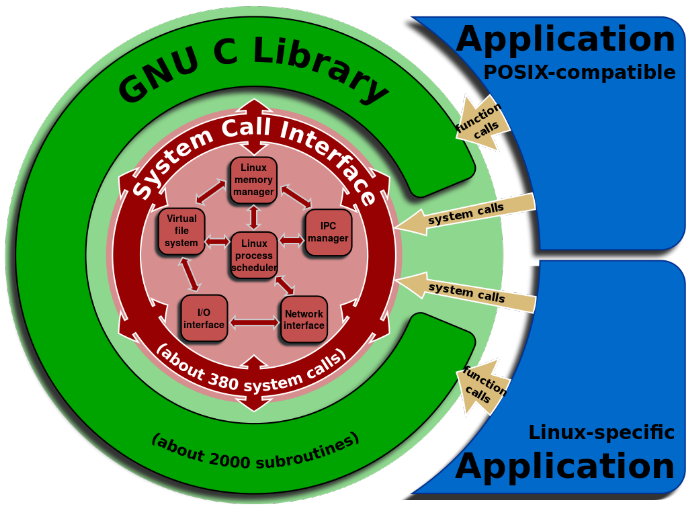


Figure 1.1: Kernel space (in red) and user space (in green and blue)

Very often, useful operations in the system require privileged services provided by the kernel. For example, even an extremely simple shell command such as `echo` performs dozens of system calls, which are reported below as listed by the command `strace -wc echo`

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 29.57    0.000514         514         1           execve
 17.03    0.000296          33         9           mmap
 11.62    0.000202          67         3           brk
  8.52    0.000148          37         4           mprotect
  7.19    0.000125          25         5           close
  6.10    0.000106          35         3           open
  5.93    0.000103          26         4           fstat
  5.58    0.000097          32         3         3 access
  2.82    0.000049          49         1           munmap
  2.24    0.000039          39         1           write
  1.84    0.000032          32         1           read
  1.55    0.000027          27         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.001738                    36         3 total
```

System calls can be called in user space applications directly, through assembly, or indirectly, by calling wrapper functions from the C standard library (`glibc`), as shown in Figure1.1.

```c
// Two different ways of calling open/close through glibc wrapper functions
// SYS_open and SYS_close correspond o the syscall numbers
int fd = syscall(SYS_open, "example.txt", O_WRONLY);
syscall(SYS_close, fd);
fd = open("example.txt", O_WRONLY);
close(fd);
```

Calling through assembly means filling the right CPU register with the syscall arguments and then using a special assembly instruction. On x86 machines it is required to fill the `EAX` register with the system call number (by a `mov` assembly instruction) and then invoke the interrupt 128 (by the instruction `int 0x80`). Modern processors may use a different one. This is what will happen upon its execution:

1. Interrupt number 128 (=`0x80`) is released. In Linux, it corresponds to a system call interrupt.

2. The process execution is suspended and the control passes to the kernel (kernel mode), which will look up the entry 128 in the *interrupt vector table*. This table simply associates interrupt numbers with their handler: a function that gets executed when the interrupt happens.

3. The corresponding handler is executed: this function copies the syscall number and arguments from the registers onto the kernel stack. It will then look up in the *system call dispatch table* the handler corresponding to the syscall number and call it with the correct arguments like any normal C function, because the arguments are now located on the kernel stack.

4. The system call is finally executed and the return value is stored in a general purpose data register.

Registers are used to pass the parameters because this way it's easier to get them from user to kernel space. It is intuitive that such a procedure to invoke system calls is architecture dependent. For this reason, `glibc` wrappers are always used: they internally execute the assembly code that we just illustrated and do it differently for each architecture. Calling wrappers is also very safe since it avoids to accidentally fill the wrong registers or miss the right number of arguments. It's important to note that the kernel can protect itself against invalid syscall arguments in registers. This is crucial since, as we saw, syscalls are easily called from user space directly by executing the proper assembly instructions.

1.2.2 A different kind of software

The separation between kernel/user space and the fact that we are working at such low level makes the kernel a very peculiar piece of software. One of its properties is that there is no error checking, this is because the kernel trusts itself: all kernel functions are assumed to be error-free, so the kernel does not need to insert any protection against programming errors[2]. Instead, what the kernel does is to use assertions to check hardware and software consistency; if they fail then the system goes into *kernel panic* and halts. Later on in Section 5, it is shown an example of code where the `panic()` routine is called. The choice of checking assertion (and possibly going to kernel panic is something went wrong) is that since the kernel controls the system itself, error recovery and error correction is very hard and would take a huge part of the code. Another way of thinking about it, is that there is no meta-kernel that handles kernel errors. Of course programming or hardware errors can (and will) still occur: when this happens the offending process is killed and a memory dump called "*oops*" is created. A typical example of this is when the kernel dereferences a NULL pointer: in user space this would cause a *segmentation fault*, while in the kernel it will generate an oops or in the worst case go directly into panic. After this kind of event, the kernel can no longer be trusted and the best thing to do would be to reboot, because the kernel is in a semi-usable state and it could potentially corrupt memory. Furthermore, a panic in this state is more likely to happen. Possibly, the user experiencing the kernel panic may also inform the kernel mantainers.

Another peculiarity of the kernel is that it uses its own implementation of the functions in the standard C library. For example `printf()` and `malloc()` are implemented as `printk()` and `kmalloc()`. There are different reasons for this choice, one of those is that the C standard library is too big and inefficient for the kernel. Another reason is that implementing your own functions gives more freedom because they can be customized for their purpose in the kernel. Memory allocation in user or kernel space is very different, so the `kmalloc()` implementation is very specific. For instance, kernel data structures need a contiguous physical memory segment to be allocated, while regular user space allocation doesn't have this restriction. Furthermore, `printk()` writes its output into the kernel log buffer (that you can read by using the `dmesg` command in user space); this is very different from `printf()` that writes on standard output.

1.2.3 User and kernel stacks

As stated earlier, the memory management is different in kernel/user space. The same applies to the execution. Every process in the system has two stacks, located respectively in user and kernel space, and it will use one of the two while executing in the corresponding privilege mode. x86 CPUs automatically switch stack pointers when privilege mode switches occur, which usually happens for syscalls. The user space stack can potentially be very big, with a very high limit (8MB on my machine, but it can be increased), and even though it's initially small it can allocate more physical memory as it needs it. This mechanism is called "on-demand paging", which is discussed in Section **??** together with other topics on virtual memory. The kernel stack, unlike the user stack, cannot expand itself and it has a fixed size of two pages. Since, 32-bit and 64-bit systems have 4KB and 8KB sized pages, then the kernel stack size is of size 8KB or 16KB, respectively. These two pages must be allocated contiguously, which can cause memory fragmentation for long system uptimes as stacks get deallocated. In other words, it becomes increasingly hard to find two physically contiguous unallocated pages as the OS runs for a long time. For this reason, in the past efforts were made to reduce the stack size to one page, which would eliminate fragmentation, but after many stack overflows the standard settled on two pages.

This leads us to an interesting example of the kernel trusting itself: it makes the strong assumption that the stack will never overflow: **no protection against it is in place**. So what happens if it overflows? First, it will corrupt the `thread_info` data structure, which is the first data that the stack encounters along its path (Figure 1.2). This will make the process nonexistent for the kernel and cause a memory leak (we will see why). Next, the stack can overflow outside of the address space and silently corrupt whatever kernel data is stored; the best case scenario here would be a kernel panic to prevent any further memory corruption. Another natural question might be "why are kernel stacks so small?" and the answer is simple: first, to use a small amount of kernel memory, and secondly, because of fragmentation. The bigger is your data structure in contiguous physical memory, the more it is hard to allocate. It is expected that any process stays in kernel mode for a small amount of time, so it should use a very small portion of the stack. A consequence of small stacks is that very few recursive functions are used to avoid long call chains and minimize stack usage; the same is true for big static allocations on the stack.

It's important to note that there are special processes called *kernel threads*

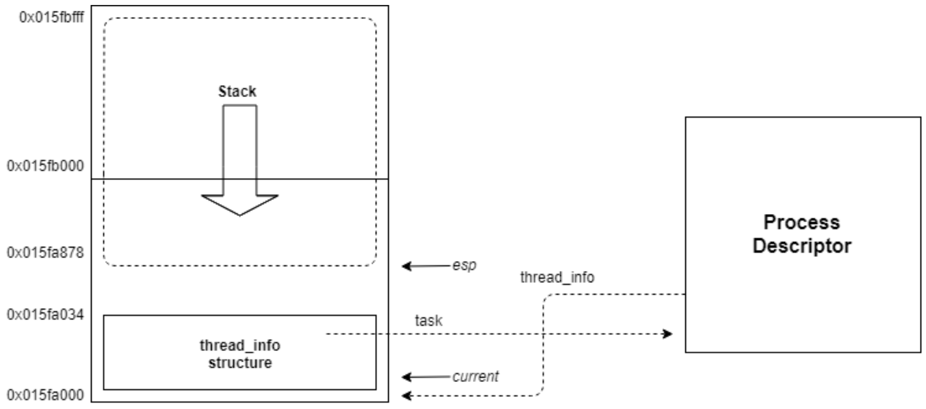


Figure 1.2: The kernel stack inside its small address space of two pages, it grows downward towards low memory.

that do not follow this pattern of kernel/user stack. Kernel threads perform a specific system task, they are created by the kernel and they live exclusively in kernel space, never switching to user mode. Their address space is the whole kernel space and they can use it however they want. Besides this, they are normal and fully schedulable tasks just like the others. An example of a kernel thread is `ksoftirqd`: there is always one for each CPU and their job is to dispatch interrupt requests. As a side note, the name stands for "Kernel Software Interrupt ReQuest Daemon", many kernel threads follow a similar naming convention.

1.2.4 A monolithic design

There are fundamentally different design approaches in kernel development. We can see these as a spectrum, where on one end there is the *monolithic kernel*, and on the other one the *microkernel* (or *μkernel*). The choice depends on how many services are located in kernel space: while in monolithic design every service is in the kernel itself, microkernels strive to reduce as much as possible the code running in kernel space. This is done by moving most services in user space, while keeping only essential primitives in the kernel (Figure 1.3).

These services are implemented as *servers*, and communication between the servers, applications, and the kernel is based on message passing. As in classic client/server approach, applications send requests to the servers, which can in turn request services to the kernel or satisfy the request directly. Because of this design choice, the system relies heavily on *Inter-Process Communication* (IPC), which can be achieved in different ways: in this case, there are actual messages being passed between processes. Even if they are part of the core architecture, the servers are user processes and run in user space just like the other user processes, though they get higher privileges.

By reducing the code running in kernel space, there is less risk for bugs.

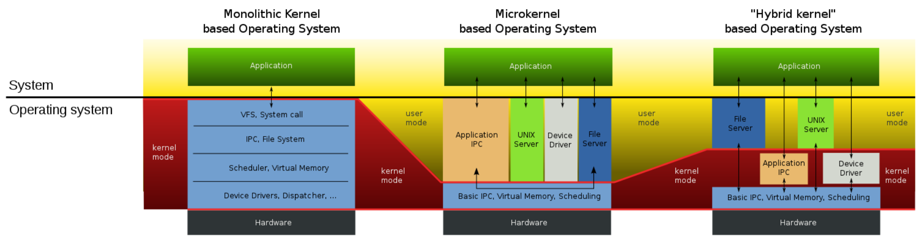


Figure 1.3: The most popular kernel designs and their differences

Because the trusted codebase is very small, there is no need to make big assumptions like in monolithic kernels. As stated before, a bug in the kernel can bring down the entire system, but in microkernels bugs are contained. For example, if the networking service crashes, then we can just restart it since it's just a user process; in a monolithic environnement, this problem would have crashed the entire system: this is one of the biggest flaws of the monolithic design. A small trusted codebase also means more portability because all the achitecture dependent code is concentrated in the small kernel. The actual operating system is built on top of it, so it would be possible to implement it in a more high level language, while only the primitives in the kernel must be ported. Conversely, in a monolithic kernel, many functions must be rewritten for each architecture: in Linux, the folder for architecture dependent code (`arch`) is the second biggest folder and it represents 8% of the code.

Another direct consequence of the shift of the code in user space is that microkernels are more easily maintainable. Development is easier because most of the code runs in user space, so the usual restrictions for kernel code are not present: for example, it would be possible to make use of `glibc`. Furthermore, testing can be done without rebooting the system: just stop the service, recompile the code and then start it again. On a monolithic system, not only it's needed to recompile the whole kernel, we must also reboot in order to load the image again. And if this new image doesn't work, then we must reboot again with the working image. In practice, this is always done in a virtual machine in order to test more efficiently, but it's still a tedious process.

Given all these advantages, why aren't microkernels always used? It's (mostly) because of one deadly flaw: the performance penalty. It's easy to see this if we think that monolithic kernels communicate directly with hardware, while in microkernels most of the operating system does not; essentially, microkernels add an additional layer of abstraction though heavy use of IPC. More precisely, the task of moving in and out of the kernel to move data between the applications and servers creates significant overhead. This process results in two major problems:

- A large number of system calls, caused by services frequently needing to use the primitives.
- Many context switches, because each service must be scheduled as a process. In order to pass a message between two services, a full context switch is

needed to send and receive.

This last problem is not an issue in a monolithic setting, because kernel functions are executed when any currently running process enters kernel space. Of course, calling a plain function is much less costly than doing a system call or context switch. Furthermore, IPC in monolithic kernels is implemented through shared memory, which is more efficient than IPC with message passing. In Linux, because every functionality is in the kernel, it is a single, big program running in his dedicated address space: this means that every subsystem (scheduling, IPC, networking, memory management. . .) shares the same memory. Paradoxically, all the auxiliary code needed for interfacing and communication often makes microkernel-based operating systems larger than monolithic kernels, even though all this code is not in kernel space.

Linux is a monolithic kernel, and because of this design choice, even the device drivers are located in kernel space: in fact, more than 65% of the kernel code is just drivers (in the `driver` directory, the largest folder). This means that while the system is running a huge part of the code is not being used. For this reason, many miniaturized versions of Linux have been distributed: a fully functional—and still monolithic—kernel can fit on a single floppy disk. If we wanted to create just a reduced version, it would not be too hard to remove drivers that are not needed and then recompile the kernel.

A problem of the monolithic design is the natural lack of modularity; microkernels don't have this problem because it's very easy to start/stop drivers running in user space. Monolithic kernels try to achieve the modularity of microkernels by using *kernel modules*: they are simply code that can be inserted/removed from the kernel at runtime. A module can be linked to the running kernel when its functionality is required and unlinked when it is no longer useful: this is quite useful for small embedded systems, to keep running code to a minimum. Modules are often used to add/remove drivers and this approach is much faster than having drivers in user space: since the code runs in kernel space, there is no need to do message passing or communicate with user space at all. It's just like in a microkernel and without performance penalty, but then again, now we are programming in kernel space, which is harder. In the end, it's a choice between ease of development/fault tolerance or performance. Furthermore, modules, unlike the external layers of microkernel operating systems, do not run as a specific process. Instead, they are executed in kernel mode on behalf of the current process, like any other kernel function: this means less switching between processes, so again, better performance. Because of the big flaw of monolithic kernels mentioned earlier, if a driver module doesn't behave correctly, the system can crash upon module insertion. In section 4.4.1 it's shown just how easy it is to crash the system by inserting our own—voluntarily bugged—module. Needless to say, this operation requires root privileges: module insertion essentially means injecting arbitrary code in the kernel.

Modules are powerful, but cannot always accomplish what a microkernel can. As an example: on Linux, it's not possible to replace the scheduler at runtime. In order to do that, it's needed to have the two different schedulers directly in

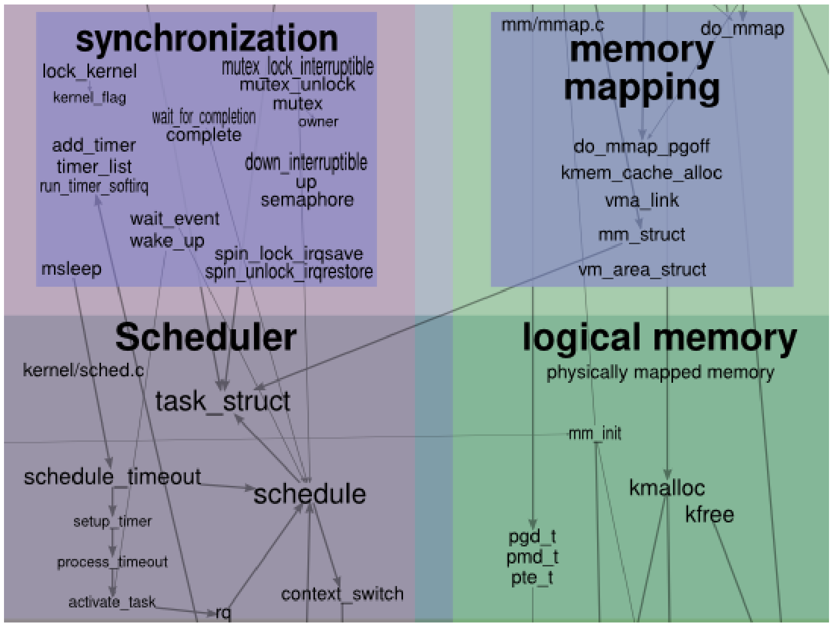


Figure 1.4: A portion of the kernel subsystems map (source: `www.google.com`)

the core code and switch between them at runtime (This is how it's actually done in Linux: there are different schedulers already implemented). Modules usually aren't used to implement core functions, but are rather seen as extensions of the kernel. This means that is very difficult to modify policies decided by the kernel through modules, and users must adapt to these policies or modify the code and recompile the whole kernel. Conversely, in microkernels it's easy to change core implementation, since it runs as a service.

Finally, the hybrid design is halfway between monolithic and microkernel, as it tries to take the best side of both approaches: having good performance, but also, to some extent, flexibility and maintainability. In practice, its philosophy is very similar to a monolithic kernel and hybrid kernels have been dismissed by Linus Torvalds as "just marketing"[3]. Notable OSes that use a hybrid kernel are Windows and MacOS.

1.3 Process management

The kernel is divided into subsystems that interact with each other. Figure 1.4 is a zoom into the kernel mechanisms inside the red part of Figure 1.1. The image represents the part of the kernel that will be covered, for the most part we will swing between the scheduling and memory mapping subsystems. The names in the picture are structs, functions or source code files; we will get familiar with most of these as we go on.

1.3.1 Processes and threads

A process is an instance of a running program. Each process has resources associated with it, such as an address space, open files, global variables and code. Each process must have its own address space that only he can access: when a process tries to access a memory location that does not belong to it, a segmentation fault interrupt is generated. A thread is defined as a single flow of execution, it has associated a stack of execution and the set of CPU registers that it uses, most notably the stack pointer and program counter. Each process can have multiple threads, in which case it's a *multi-threaded process*; threads belonging to a process will share resources between each other. The execution aspect of a process is always represented by threads, which means that a process cannot exist without at least one thread associated.

The kernel does not distinguish between processes and threads, so they are treated as the same entity. Because of this, a problem in terminology arises. Next, it is shown how processes ans threads are distinguished.

Each process has its own PID (Process IDentifier) and groups of processes are identified by the TGID (Thread Group ID). If a process has only a single thread then its PID is equal to its TGID. If a process is multi-threaded then each *thread* has a different PID, but they will all have the same TGID. Furthermore, there will be a thread in this group called *thread group leader* that will have its PID equal to the TGID, so the TGID field in each thread is just the PID of their leader. Just to add some more confusion, when you call `getpid()` you are actually getting the TGID (the group leader PID, identifying the whole process), and when you call `gettid()` you are getting the PID (which identifies a single thread, not a group). Hence, the PID resambles more a thread identifier. This confusing way of using IDs was implemented to comply with POSIX standards, which require that each thread of a multi-threaded process must have the same id: this is why `getpid()` returns the TGID.

The real difference between threads and processes is that threads share the same address space, while processes do not. By saying that some threads are associated to a same process just means that they are sharing an address space. This enables concurrent programming, enables communication among threads via shared memory, and requires then synchronization methods. As shown in Section 1.3.3, using threads in a program instead of spawning new processes results in much better performance, which is why threads are sometimes called *lightweight processes* or LWP.

```c
//stack size for cloned child
const int STACK_SIZE = 65536;
//start and end of stack buffer area
char *stack, *stackTop;
// ... define child startup function "do_something" ...
stack = malloc(STACK_SIZE);
stackTop = stack + STACK_SIZE; //stack grows downward

//spawns a new thread
clone(&do_something, stack + STACK_SIZE, CLONE_VM | CLONE_FS | CLONE_FILES |
 ↪  CLONE_SIGHAND, 0);
```

```
11  //spawns a new process, this is the same as using fork()
12  clone(&do_something, stack + STACK_SIZE, SIGCHLD, 0);
```

The system call `clone()` spawns a new child process. It's very similar to `fork()` but it's more versatile because flags can be used to decide how many resources are shared with the new process. `CLONE_VM` (where vm stands for virtual memory) makes the child process run in the same address space as the father, while the other flags clone filesystem information (such as working directory), open files and signal handlers. The flag `SIGCHLD` at line 4 requires that the parent process receives a `SIGCHLD` signal upon the termination of the created child process. Ultimately, the reason why threads and processes are treated as the same entity in Linux, is that processes are just threads that share nothing. In fact, the word *task* is always used inside the kernel instead of process/thread and we will do the same, especially when discussing implementation.

Each task is represented in the kernel with the struct `task_struct`, this is a fairly big structure that can be almost 2KB in size, depending on configuration at compile time. `task_struct` is what is often referred as the *process descriptor* or PCB (*process control block*): every information about a task is stored in here.

```
struct task_struct {
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;
        void                           *stack;
        /* Current CPU: */
        unsigned int                   cpu;
        // A boolean, "on_runqueue"
        int                            on_rq;
        int                            prio;
        int                            static_prio;
        int                            normal_prio;
        int                            exit_state;
        int                            exit_code;
        int                            exit_signal;
        /* The signal sent when the parent dies: */
        int                            pdeath_signal;
        pid_t                            pid;
        pid_t                            tgid;
        /* Real parent process: */
        // The original parent that forked this task
        struct task_struct __rcu        *real_parent;
        /* Recipient of SIGCHLD, wait4() reports: */
        // The current parent, maybe the original one exited
        struct task_struct __rcu        *parent;
        // Executable name, usually the command that spawned this task
        char                            comm[TASK_COMM_LEN];
        /* Filesystem information: */
        struct fs_struct               *fs;
        /* Open file information: */
        struct files_struct            *files;
        /*
         * Children/sibling form the list of natural children:
         */
        struct list_head              children;
        struct list_head                sibling;
        struct task_struct           *group_leader;
        /* PID/PID hash table linkage. */
        struct pid                       *thread_pid;
        struct hlist_node          pid_links[PIDTYPE_MAX];
        struct list_head                thread_group;
        struct list_head                thread_node;
};
```

Code from `include/linux/sched.h`
These are some of the most basic fields the struct, most of them are self explanatory.

The `volatile` keyword asks the compiler not to optimize by caching the storage of this variable. This indicates that the value may change even if the variable does not appear to have been modified. Hence, every time a `volatole` variable is accessed, it needs to be read from the main memory. The opposite of `volatile` is the compiler hint `register`. The fact that the task state is volatile makes sense because it could be unpredictably modified by interrupts: it could be possible that an old value of the variable is read from the cache instead of the actual value.

Let us now focus on the `pid` fields to show how Linux uses pids to find any information and resources of a task. Given a pid, searching linearly through the pids to find the task we are looking for would be very inefficient. Instead, a hash table known as *pid hash table* is used for this purpose. The identifiers in this table are simply the result of hashing a given pid, you can see in figure 1.5 that conflicting entries are simply stored in a list associated with the same id. Because it's a hash table, the kernel can quickly look up the pid and find in $O(1)$ time the corresponding process descriptor. This procedure is, for example, applied when the command `kill [PID]` is launched.

```
enum pid_type {
        PIDTYPE_PID,  //process PID
        PIDTYPE_TGID, //thread group leader PID
        PIDTYPE_PGID, //process group leader PID
        PIDTYPE_SID,  //session leader process PID
        PIDTYPE_MAX
};
```

Code from `include/linux/pid.h`
There are actually four tables, one for each PID type. Each of these tables is an array of `hlist_head`, the head of the chain list, which points to a list of `hlist_node` (see Figure 1.6). These structures are used for non-circular lists. These lists are populated by `struct pid`, and a pointer to this struct is stored inside each process descriptor in the `thread_pid` field. Figure 1.6 shows an example for the TGID class that we discussed earlier. PIDs in the chain list are colliding and are different processes, PIDs in the `pid_list` are threads in the same group, where the leftmost thread in the image is the group leader. Despite the name "`list_head`" inside the pid structure, such a field points to a circular

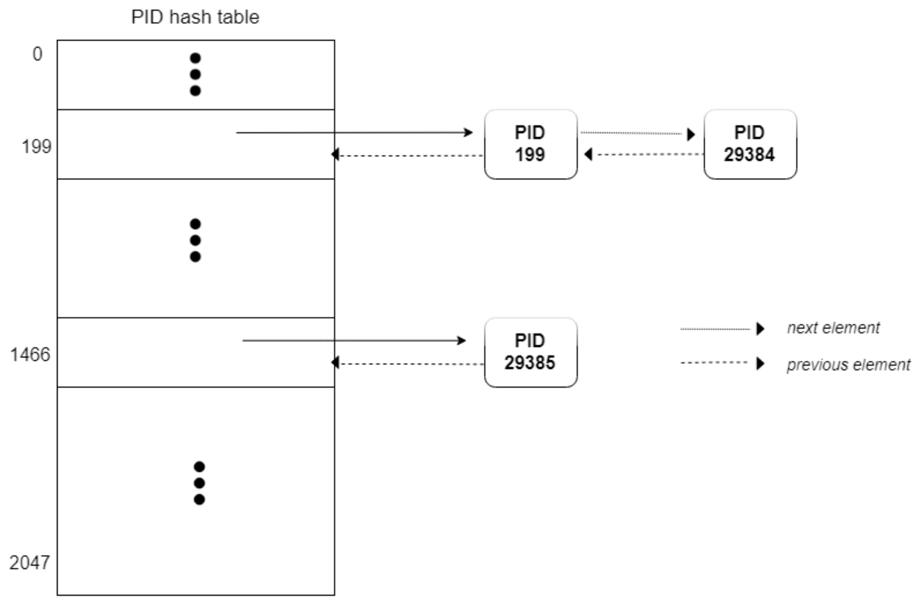


Figure 1.5: Pid hash table, pids 199 and 29384 are both hashed to 199

list, and since it's circular there's really no head structure that points to the first element.

```
struct pid {
        atomic_t count; // number of references to this PID
        int nr; // PID number
        struct hlist_node pid_chain; // Link to next and previous conflicting
         ↪   entries
        struct list_head pid_list;  // per-PID list
};
```

Code from `include/linux/pid.h`

The implementation of `struct pid` is slightly different from what was presented, with other nested structures and a different linkage to the hash table. In this section, only a small portion of the process descriptor is described. Other fields are illustrated in Section 3.

1.3.2 List implementation

In a classic circular, the `struct` of the node contains the data and pointers to the next and previous nodes. This implementation is naive and would lead to have a different structure for each data type, or using a void pointer to your data for no reason. Let's see how lists are used in the kernel.

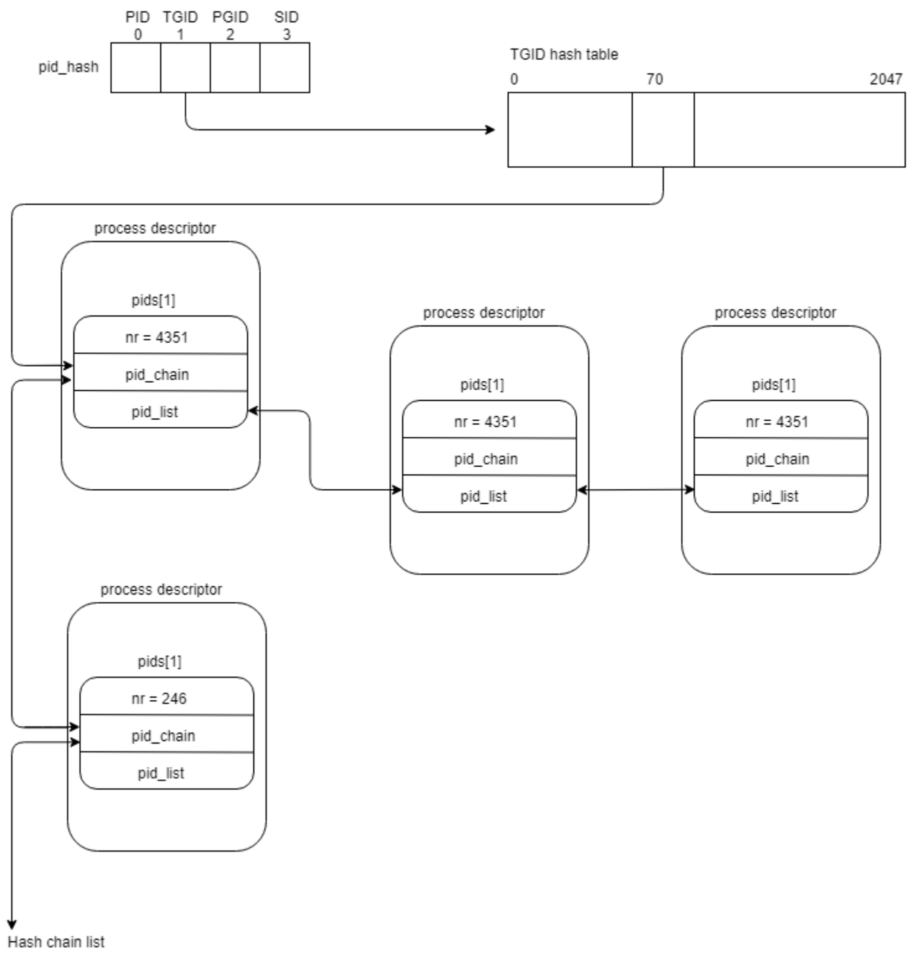


Figure 1.6: Hash table for the TGID pid type

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

The data is not contained in the list itself, but in another structure that contains the list node (Figure 1.7). For example, Linux keeps a big circular list of every `task_struct` in the system: this is done by embedding "`struct list_head tasks;`" into `task_struct`. Notice how this is not a pointer to a node: the node is embedded directly into the structure. So how can we get the data we want in the structure without a pointer in the node? The answer is the `container_of()` macro. This macro works with anything, but let's assume that we have a list

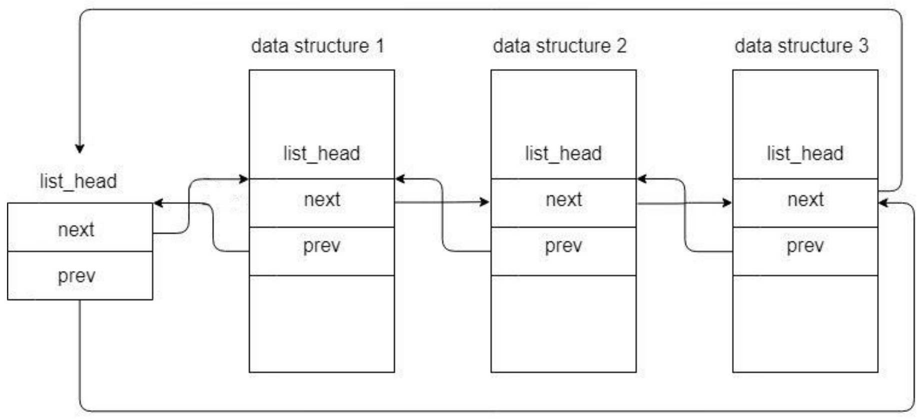


Figure 1.7: A generic doubly linked circular list

embedded in the container structure.

```
#define container_of(ptr, type, member) ({ \
        void *__mptr = (void *)(ptr); \
        ((type *)(__mptr - offsetof(type, member))); })
// An alias that's used everywhere
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

Code from `include/linux/kernel.h`
ptr is the pointer to the list node, type is the container struct, member is the field name of the list node in the container struct. We first cast ptr to a void pointer, then we subtract the offset from the beginning of the container struct of the field we want to get. When we allocate a struct, its field are allocated contiguously in virtual memory in the order that we declared them: this means that by moving the pointer backwards from a field by the right amount, we can end up at the beginning of the container structure. This is how we can get the offset of the specified field in any struct:

```
#define offsetof(TYPE, MEMBER) ((size_t)&((TYPE *)0)->MEMBER)
```

Code from `include/linux/stddef.h`
TYPE is the struct we are considering, MEMBER is the name of the field, what it does is:

1. Take the address 0, the first in the address space of the process
2. Cast it to a TYPE pointer
3. Dereference the pointer and take the MEMBER field

4. Take the address of the field and cast it to a size, now it's no longer an address

Essentially, we are pretending that there is the container structure allocated just at the beginning of the address space. This is arguably a bit of a hack, but it's perfectly safe since we are just playing with pointers and never touching actual memory. Indeed, it would be very dangerous to dereference and modify data from a random pointer in memory. This approach has many advantages, such as being able to have multiple lists associated with the same data. `task_struct`, for instance, contains also the `children` and `sibling` lists among many others. This implementation is also very easy to use and it's oblivious about types.

1.3.3 Scheduling

A system with a single CPU can execute only one process at a time. For this reason, a scheduler for processes is needed. Process scheduling consists in choosing which processes should run in what order, essentially deciding how CPU time is shared among processes. To achieve this, there are many scheduling algorithms such as FCFS (*first come first served*), RR (*round robin*), EDF(*earliest deadline first*) and SJF (*shortest job first*). Most of the scheduling policies are *preemptive*, which means that at any time the scheduler can arbitrarily decide to interrupt the currently running task and assign the CPU to another process. The use of preemption implies that processes have assigned *timeslices*: they are periods of time in which the process is allowed to run and after which it will be preempted.

FCFS, which is the most basic scheduling algorithm, doesn't have preemption nor timeslices: every process runs as much as it wants before voluntarily giving up the CPU to the next task in the queue. Round robin is similar to FCFS because it has a FIFO runqueue; the difference is that it uses a constant timeslice, called *quantum*, assigned to each process: when the quantum expires the process gets preempted and the next task is scheduled.

In a UP (*uniprocessor*) system it is not possible to achieve true parallelism among processes. The only way to do it is to have multiple processors that share a common bus and the central memory: this is known as SMP (*symmetric multiprocessing*). A single processor can also have multiple cores, but each one is treated as a separate processor, so the SMP architecture applies to cores as well. Even on SMP systems, which represent most systems today, there often are more processes than cores. Hence, scheduling is necessary for each processor/core. There are also new problems that arise in SMP, such as *load balancing*: the problem of balancing processes between CPUs so that no CPU goes idle or has an unfair amount of workload. This kind of related problems must also be taken into account by the scheduler.

Every job carried out by the scheduler will eventually lead to a process switch on a given CPU. The kernel has a mechanism to suspend the execution of a process, save its status, and resume another process. This procedure is called *context switch*. Each process has an *execution context*, which includes everything needed for a task to execute, from its stack to the code. While every process can have its own process descriptor, the registers on the CPU must be shared between every process in the system. Every value in any register that a process is using is a subset of the execution context and it's called the *hardware context*. At every context switch the hardware context must be saved and restored, respectively, for the old and the new process. The content of the registers are saved in part in the process descriptor of the preempted process, and in part on its kernel stack.

The routine that performs a context switch is called — not surprisingly — `context_switch()`, and it is called only in one well-defined point in the kernel: inside the `schedule()` routine, which triggers the scheduler and chooses the next task to schedule. `context_switch()` (as we will see the code later) basically switches the address spaces of the two processes and then calls `__switch_to()`. This last function operates on registers and kernel stacks, so it's one of the most architecture dependent in the whole kernel. This is why, like many other similar routines, there is one version for each architecture supported by Linux in the `arch` folder. Next, the x86 version of the context switch is described.

There are 6 *segmentation registers* that hold *segment selector*, basically the starting address of memory segments in the process address space.

- `cs` *Code segment*, this points to the segment containing instructions of the loaded program, also known as the `.text` section. We mentioned in section 1.2 that this register also holds 2 bits that describe the current privilege level of the CPU.
- `ss` *Stack segment*, points to the segment containing the stack of execution.
- `ds` *Data segment*, points to the segment containing global variables and constants, also known as the `.data` section.

The other 3, `es`, `fs` and `gs` are general purpose are don't hold a specific address. There are also general purpose data registers that hold data used in operations (`ax`, `bx`, `cx`, `dx`) and pointer registers, that hold offsets:

- `ip` *Instruction pointer*, offset to the next instruction. If added to `cs` will be the address of the next instruction to fetch (`cs:ip`).
- `sp` *Stack pointer*, offset to the top of stack. If added to `ss` will be the address of the top of stack (`ss:sp`).
- `bp` *Base pointer*, offset to subroutine parameters on the stack (`ss:bp`).

Let's now see which part of the process descriptor is involved in context switching.

```
struct task_struct {
    // ...
    /* CPU-specific state of this task: */
    struct thread_struct    thread;
};
```

```c
struct thread_struct {
#ifdef CONFIG_X86_32
    unsigned long        sp0;
#endif
    unsigned long        sp;
#ifdef CONFIG_X86_32
    unsigned long        sysenter_cs;
#else
    unsigned short        es;
    unsigned short        ds;
    unsigned short        fsindex;
    unsigned short        gsindex;
#endif
    // ...
    /* Floating point and extended processor state */
    struct fpu      fpu;
};
```

This struct is obviously very architecture dependent, its purpose is to save the hardware context before the context switch. You can see that even if it's specific to x86 it can still change depending on 32 or 64-bitness. You can also notice that only a small part of the hardware context gets saved in the process descriptor: the kernel stack pointer, general purpose segmentation registers, data segment and the floating point registers. In older versions of the kernel most of the registers were stored here. Let's see in detail what happens when the kernel switches from process A to process B. There are actually two different mechanisms in this procedure: the entry/exit mechanism (user/kernel stack switch) and the context switch.

1. Process A enters kernel mode, so it will switch from its user stack to its kernel stack, in other words: it saves its **user** hardware context in the kernel stack. It does so by pushing its **user mode** stack (`ss:sp`), instruction pointer (`cs:ip`) and data registers onto the kernel stack, then all CPU registers are switched to use the kernel stack.

2. When in kernel context, process A invokes `schedule()` which will eventually do `context_switch()`.

3. Process A saves its hardware context:

   (a) It pushes most of its register values onto the kernel stack by a series of `mov` assembly operations.

   (b) It saves the value of the stack pointer (which is pointing to the **kernel** stack) into its `task_struct->thread.sp`.

   (c) Other registers such as the floating point registers are saved in the `thread` field of `task_struct`.

4. Process A loads a previously saved stack pointer from process B's `task_struct->thread.sp`, also loads the other saved registers

5. Address spaces are switched.

6. Using the loaded stack pointer, process B moves its previously saved registers from its kernel stack into the registers. This is done by a series of `pop [register]` assembly operations. Process B's state is now completely restored.

7. process B exits kernel mode and restores its **user** context. This is accomplished by loading previously saved registers from the kernel stack: its **user mode** stack (`ss:sp`), instruction pointer (`cs:ip`) and data registers. Process B is now in user context.

To understand scheduling mechanisms in the next sections it's important to highlight something in step 2, when the scheduler gets called by a process running in kernel mode. It may be intuitive to think of the scheduler as some kernel thread that is permanently running in kernel mode, but that is not the case. **The scheduler does not run as a separate thread, it always runs in the context of the current thread.** This means that any process in the system that goes from/to kernel mode can potentially execute the scheduler himself, using its own kernel stack. What exactly can trigger the scheduler is something that we'll see in detail in section 3. The simplest case is when a process voluntarily gives up the CPU by going into a sleep state, in which case it subsequently executes `schedule()` in kernel mode (it would have switched already to kernel mode when sleeping). Another thing to highlight is how the user hardware context has nothing to do with context switch, this is because it always gets saved/restored on the kernel stack when entering/leaving the kernel. An implication of this fact is that context switches always happens in kernel mode, which is expected since it's a core system task.

It's important to understand that a context switch generates significant overhead and, in fact, most of the scheduling overhead comes from context switching. It is caused by the need to switch address spaces and by the fact that context switching is not cache friendly. This is the reason why a context switch between threads (LWP) is almost inexpensive compared to context switching different processes: step 5 in the procedure is skipped because threads share an address space, so there is no need to switch it (again, this is why they are **lightweight** processes).

1.3.4 Tasks lifetime

Tasks have a life cycle: a new child process task is created every time a task uses fork-like system calls. As shown in Section 1.3.1, once a process is created some resources are inherited from the father, depending on the `clone()` flags, while `fork()` will duplicate the calling process. There are some resources that will always be inherited and there is no reason to duplicate, such as the executable object code (the `.text` memory segment in Linux). The new process will be in the runnable state and ready to be scheduled. When the process needs to wait for a particular resource, it goes into a sleep state; it will then become runnable again when the resource is available, or after a predefined time when the syscall `sleep()` is used. A process can also go from running to runnable: this happens if the process is preempted or if it gives up the CPU voluntarily. This last case happens, for instance, if the process needs to do I/O operations for which it doesn't need the CPU. This way no processor time is wasted and another task is scheduled.

A process terminates by executing `exit()` or when it receives a signal (including `SIGHUP`, `SIGINT`, `SIGKILL`, `SIGTERM`, and others) from other processes which have the privileges to do so. Upon exit, its *exit state* will initially be set to the *zombie* state. A zombie process is a process that terminated, but its process descriptor and entry in the pid hash table are still present in memory and accessible (for example, by `ps -aux`). Tasks' resources are not deallocated immediately because the parent process may want to access some of this information, most likely the *exit status*, or may want to synchronize with the child process termination via `wait()` ot `waitpid()`. This is actually a relevant resource leak because `task_struct` is almost 2KB in size. Hence, if there are many zombie preocesses then a big portion of memory is simply wasted until the parent process executes a `wait()`/`waitpid()`. More in details, a `task_struct` plus its kernel stack consumes around 10KB of low kernel memory, that is `THREAD_SIZE + sizeof(struct task_struct)`, assuming that kernel stacks are 8KB in size (`thread_info` and pidhash entry are too small to be relevant).

After terminating and sending a signal to the parent, a task will remain zombie until its parent performs a `wait()`, upon which the parent gets information about the terminated child. Subsequently, `release_task()` is executed and the last data structures from the descriptor get detached. `detach_pid()` is called twice to clear the entry in both the `PID` and `TGID` hash tables, then `task_struct` is finally deallocated. Zombie processes are impossible to kill externally: they can't receive signals as they no longer exists, so a wait by the parent is the only way to clean the memory occupied by the zombie data structure. Suppose that the parent of a zombie process exits without waiting: the child will be an orphan process so it will be become a child of `init`. Luckily, the ancestor process (`init`) has a routine that waits periodically to reap possible zombie processes; so the child process will simply be waited by `init` and get cleared. This mechanism ensures that memory won't be cluttered by zombies and leaves the pid table in a consistent state.

States and exit states of a process are defined in `include/linux/sched.h` as following.

```
/* Used in tsk->state: */
#define TASK_RUNNING    0x0000
#define TASK_INTERRUPTIBLE  0x0001
#define TASK_UNINTERRUPTIBLE    0x0002
#define __TASK_STOPPED                          0x0004
#define __TASK_TRACED                          0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD   0x0010
```

```
#define EXIT_ZOMBIE 0x0020
#define EXIT_TRACE  (EXIT_ZOMBIE | EXIT_DEAD)
```

- `TASK_RUNNING` is either a process that is ready to be run (in which case it's more like "runnable") or that is actually running.
- `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE` are both states in which a task is sleeping, waiting for some condition to be true. The former allows a process to be woken up by signals, the latter does not: an uninterruptible task will ignore any signal and will only wake up on his condition. This distinction is the reason why, as we will see later, the routine that wakes up tasks is called `try_to_wake_up()`.
- `__TASK_TRACED` means that another process is tracing this one, usually a debugger such as `ftrace` (as explained in Section 4).
- A task in `__TASK_STOPPED` is not running and cannot be scheduled: this happens upon stop signals or any signal from a tracing process.

The values associated to these states are defined like this so that they can be used for bitmasks, which is the standard way to handle flags. Each flag is a power of 2 (in hexadecimal) so flags can be combined with bitwise operator `|` or be tested with `&`. For example, checking if a task is sleeping can be easily done like this:

```
if(tsk->state & (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE))
    printk("task %d is waiting for something", tsk->pid);
```

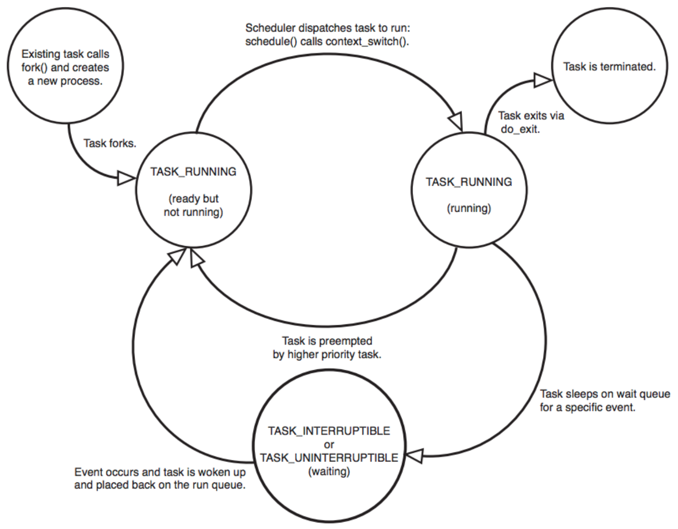

Existing task calls fork() and creates a new process.
Task forks.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

Figure 1.8: State machine of task states

Chapter 2

The Linux scheduler

As illustrated earlier, the CPU can only execute one task at a time and, even on multiprocessors systems, the number of tasks will be larger than the number of cores. For this reason, the scheduler is in charge of alternating the execution of tasks. In order to decide which task to run and for how long, the scheduler has to consider many factors, such as the importance of the task and its type.

2.1 Introduction

2.1.1 Objectives of the scheduler

The scheduler cannot simply choose the tasks in any order: there are some tasks that are highly time sensitive and need to be executed as fast as possible, and others that can wait longer without any consequence. This *interactiveness* criteria is one of the most important aspects that the scheduler has to take into account.

Response time and throughput When interacting with the system, it's expected that it will react almost immediately. Clearly, the user doesn't want to wait a second (or more) between the pressure of a button and the response from the system. Hence, the scheduler aims at detecting if the process interacts with the user and tries to minimize the response time of such processes.

There are different types of tasks: a text editor, for example, will spend most of its time waiting for user input, and when the input arrives, the task should respond as fast as possible. We call this kind of task *interactive*. Other types of tasks, on the other hand, could utilize the CPU all the time without ever sleeping. This means that the scheduler, not only needs to minimize the response time, but it should also give precedence to tasks that are highly interactive—especially on desktop systems.

Ideally, the scheduler should do as few process switches as possible. Preempting a task, choosing another one and then context switching is a long operation.

Every time the scheduler is performing a context switch, the CPU is not being used by any task. This is why, on the long run, frequent context switches will have an impact on the performance of the system. More specifically, they will decrease the *throughput*, which we define as the total amount of work completed per unit of time. It's important to note that we are talking about work done for the user: switching processes doesn't count as work since it's essentially overhead. Increasing the frequency of context switches would increase the responsiveness of the system, but it would also reduce the performance. The scheduler strives to find a balance between responsiveness and performance.

Fairness and Starvation Another important property that needs to be achieved is *fairness*. What it means is that any two tasks with the same priority should run for roughly the same amount of time. Indeed, a task with higher priority should have the precedence over a task with a lower one, but at the same time, every task should get at least some CPU time. A process that doesn't get any CPU time is said to *starve*. In practice, this usually happens when a process with high priority monopolizes a CPU, starving all the other processes waiting for their turn.

2.1.2 Different workloads

Linux is used in all sorts of machines, from desktop computers to high-end servers to mobile devices. For this reason, the workload can vary a lot. Ideally, the scheduler should have good results in every scenario with every different workload, but in practice, this is really hard to implement. Even a small tweak to the scheduler could advantage desktop users over server users or viceversa: in other words, it's difficult to make every user happy.[7] Nonetheless, Linux tries to achieve flexibility by implementing scheduling classes. Ingo Molnar, the creator of the modern Linux scheduler, writes in the patch notes "[...Scheduling classes are] an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming about them too much."[5]. A task from a scheduling class can be chosen to run only if there are no runnable tasks in classes that are higher in the hierarchy. The scheduling classes are organized as shown in table 2.1, from high to low priority.

Classes and policies Each class has its code, implementing its own algorithm. Within each class, policies represent special scheduling decisions, and each process has a policy associated with it. This means that at specific points of the code the behavior may change depending on the process' policy. This may seem confusing, but consider `SCHED_RR` and `SCHED_FIFO`: they use the same priority scale, so processes of both policies share the same run queues. Furthermore, FIFO and round-robin are very similar algorithms, differing just because of round-robin having time quantums. From this perspective, it makes sense that tasks of both policies are scheduled using the same code. Scheduling classes are mapped to their code as follows:

Scheduling classes	**Scheduling policies**
`stop_sched_class`	
`dl_sched_class`	`SCHED_DEADLINE`
`rt_sched_class`	`SCHED_FIFO` `SCHED_RR`

`fair_sched_class`	`SCHED_NORMAL`
	`SCHED_BATCH`
	`SCHED_IDLE`

`idle_sched_class`	

Table 2.1: Scheduling classes and policies in Linux

- `dl_sched_class` – `kernel/sched/deadline.c`
- `rt_sched_class` – `kernel/sched/rt.c`
- `fair_sched_class` – `kernel/sched/fair.c`

`SCHED_DEADLINE` is the highest priority policy. A task with this policy defines a deadline, before which the task has to finish its execution. The scheduler guarantees that the deadline is respected. To do that, when the policy or the attributes are changed, the scheduler checks the feasibility of the change, and if the CPU is too busy then it returns an error. `SCHED_FIFO` is a cooperative policy, first come first served: the tasks are executed in the order of arrival, and there is no notion of timeslice. Even if FIFO is cooperative in principle, `SCHED_FIFO` allows preemption. In fact, all scheduling on Linux is peemptive. A running task with this policy can be preempted if another task with higher priority becomes runnable. `SCHED_RR` is a round-robin policy. It's similar to `SCHED_FIFO` but it uses round-robin to cycle trough all the tasks with the same priority. Each task can run only for a maximum fixed timeslice (quantum of time) then it's preempted and put at the end of list of its priority. As for `SCHED_FIFO`, if a higher priority task becomes runnable, the current task is preempted. `SCHED_NORMAL` is the default policy that is used for regular tasks and uses CFS (the Completely Fair Scheduler, implemented in `fair.c`), this chapter later describes how it works. Most tasks run with this priority on most systems. `SCHED_BATCH` is similar to `SCHED_NORMAL` but it will preempt less frequently, so every process will run longer. For this reason, it is more suited for non-interactive workloads, typically on servers. `SCHED_IDLE` is for tasks with very low priority, almost any task can preempt tasks with this policy.

Priorities Each task has a priority associated with it, the first priority levels corresponds to real-time tasks and are scheduled with the `SCHED_FIFO` or `SCHED_RR` policy. Normal tasks are scheduled with the `SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE`, these have a priority (called nice) that ranges from -20 to 19, -20 being the highest priority and $+19$ the lowest. The behaviour of different nice values depends on the version of the scheduler.

2.2 Prior to CFS

The *Completely Fair Scheduler* (CFS) is the default scheduler of Linux since the 2.6.23 release, as a replacement for the $O(1)$ scheduler. To understand the advantages of CFS, it is important to understand how the previous scheduler works.

2.2.1 The $O(1)$ Scheduler

In the $O(1)$ scheduler, runnable tasks are stored in two priority queues (*run queues*): an active and an expired queue. Initially, all the tasks are inside the active queue. Then, the tasks run on the CPU, in order of priority, for the assigned timeslice: when it expires, the task is moved into the expired queue. Once all the tasks have finished their timeslice, the two queues swap roles and the process starts again.

Prior to the introduction of the O(1) scheduler, there was another version of the scheduler that was much simpler. It worked well with a small number of tasks, but there were performance issues when working with many tasks. When the $O(1)$ scheduler was introduced its goal was to keep all the positive aspects of the previous scheduler, like good interactive performance and fairness, but improving the scalability. That is, improving the performance when dealing with a large number of tasks. This was achieved using only algorithms with constant complexity $O(1)$. Meaning that the $O(1)$ scheduler does not have to traverse all the list of runnable tasks when deciding which one to run. Instead, this decision always takes a constant amount of time, regardless of the number of tasks.

Priority Each user task has a priority which consists of a static and a bonus priority. The static priority, which corresponds to the *nice* value, can have a value from -20 to 19, -20 being the highest priority and 19 the lowest. The bonus ranges from -5 to $+5$ and it is determined by the interactiveness of the task: if a task spends a large amount of time sleeping, the scheduler will boost its priority, on the other hand, if a task doesn't sleep much, it will get penalized. The static priority and the bonus are then added to determine the dynamic priority, which is the priority looked by the scheduler when choosing the next process to run.

Timeslice The $O(1)$ scheduler is based on the concept of timeslice, this is the amount of time that a task runs on the CPU during a cycle. A new cycle begins when every task has been moved to the expired queue. Each priority level is mapped to a different timeslice: higher levels get mapped to more time while lower levels get mapped to less time. The timeslice can be expressed as a time quantity, but it's more usual to represent it as a percentage of the total time of a cycle.

Interactive tasks and heuristics A process that is marked as interactive will be reinserted in the active queue after it has expired its current timeslice.

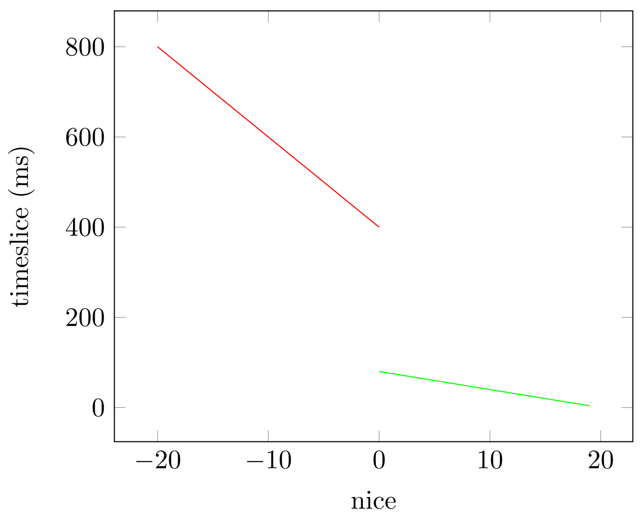


Figure 2.1: Nice to timeslice conversion

This is done to improve the response time of the system. A task is marked as interactive depending on its dynamic priority and its nice value. It is easier for a task with an higher priority to become interactive compared to task with a lower one. A task with a nice value of −20 it's marked as interactive even if it has a dynamic priority of +3, while a task with a nice value of 0 has to have a dynamic priority of −2. With a nice level of +19 it is impossible for a task to be marked as interactive, independently from the dynamic priority.

This system is the biggest weakness of the O(1) scheduler: it generated unpredictable behavior and could cause some tasks to be marked as interactive even when they were not. Furthermore, the different kinds of complex heuristics made the code bigger and harder to understand and maintain.

Nice levels Prior to the introduction of the $O(1)$ scheduler, tasks with a nice of 19 were using too much CPU and users were complaining.[7] Because of this, $O(1)$ was specifically designed to have the minimum timeslice set to one *jiffy*, which is the minimum measurable amount of time. The amount of time corresponding to one jiffy depends on the tick rate. In the kernel, the tick rate is the constant `HZ`, which is defined as the number of ticks per second. Despite the name, this value is not tied to the processor's frequency. With the advancements of computer hardware, this value increased, and today it's usually 1000 on desktop computers. With such a value, a jiffy is equivalent to 1 msec, meaning that a task with nice +19 would get a timeslice of only 1 msec (only 0.1% CPU time) and this would cause too frequent rescheduling, causing problems like cache trashing. The value of the minimum timeslice can be adjusted, but it is still `HZ` dependent. More details on this topic can be found in Section 3.2.

Another problem, that can be noticed in the graph of Figure 2.1, is that the behavior of the nice level depends on its absolute value. But the nice API only allows the nice to be changed by a relative amount. This means that with the $O(1)$ scheduler, calling `nice(1)` to increment the nice level of a task has different effects depending on the initial value.

The last problem was that tasks with negative nice values were not responsive enough: this was problematic for multimedia applications, which had to resort to run under `SCHED_FIFO` rather than `SCHED_NORMAL`. This approach caused another major problem because `SCHED_FIFO`, as we stated earlier, is not starvation proof.

2.2.2 Rotating Staircase DeadLine

Con Kolivas proposed a new scheduler that tries to solve the problems of the $O(1)$ scheduler. The goal was to design a scalable scheduler that was completely fair to all processes while allowing the best possible interactivity.[4]

The *Rotating Staircase DeadLine scheduler* (RSDL) assigns a quota of runtime based on the priority of the task. The tasks at the highest priority level are then executed round-robin with each other. When a task finishes its quota, it is moved to the next priority level and it is given a new quota. The entire priority level also has an assigned quota, when that quota expires, all the process in that priority level are moved to the next one. When a task finishes all its quotas at each priority level, it is moved to the expired queue, then, when all the tasks have finished their quota, the expired queue becomes active and the process starts again.

RSDL doesn't measure the sleep time of the tasks to identify interactive tasks. Tasks that spend most of the time sleeping will consume a little portion of their quota. When they get woken up, they will probably be at a high priority level. Meaning that, in most cases, they only have to wait for the task that is currently running. This guarantees a low latency for interactive tasks, which will rise to high priorities in a natural way, getting rid of the heuristics.

2.3 Completely Fair Scheduler (CFS)

The RDSL scheduler never made it into the kernel, but the CFS scheduler, which was developed by Ingo Molnár[5], was inspired by RSDL.

Like RSDL, CFS does not use fixed timeslices and does not use any heuristic method to calculate the priority of a task. It tries to model an ideal multitasking CPU on real hardware. Ideally every task receives $\frac{1}{n}$ of the processor's time, with n being the number of runnable tasks. This results in simpler code that can handle nice values better than the previous scheduler; the preemption time is no longer fixed like in the $O(1)$ scheduler, but it is variable.[6] This approach solves the problem found in the $O(1)$ scheduler: the behaviour of nice levels is more consistent and independent from the tick-rate; increasing the nice value by one has the same effect regardless of the starting value. Each process gets assigned a portion of the CPU depending on its weight, which is determined by the nice of the task.

2.3.1 Weight function

According to the comments in the code of `kernel/sched/core.c`, and the kernel documentation, the weight w is roughly equivalent to

$$w(n) = \frac{1024}{(1.25)^n}. \tag{2.1}$$

with n being the nice of the task, and 1024 being the weight of a task with nice 0. To avoid computing this function every time it is needed, there is a pre-calculated table which maps nice values to the corresponding weight. Below, the code from `kernel/sched/core.c` is reported. In this code fragment, the developer's comments are useful to understand how this formula works.

```c
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
static const int prio_to_weight[40] = {
/* -20 */ 88761, 71755, 56483, 46273, 36291,
/* -15 */ 29154, 23254, 18705, 14949, 11916,
/* -10 */ 9548, 7620, 6100, 4904, 3906,
/* -5 */ 3121, 2501, 1991, 1586, 1277,
/* 0 */ 1024, 820, 655, 526, 423,
/* 5 */ 335, 272, 215, 172, 137,
/* 10 */ 110, 87, 70, 56, 45,
/* 15 */ 36, 29, 23, 18, 15,
};
```

The expression of Eq. (2.1) is designed in such a way that an increase of 1 in the nice value roughly translates to a 10% increase in assigned fraction of CPU. For example, if there are only 2 processes with the same nice value, they will get both 50% of CPU time. If the nice of one of the processes is increased by one, then it will get 55% of CPU time while the other will get only 45%. Figure 2.3.1 reports the plot of the function of Equation (2.1). In the developer's comments in the code above it's said that "to achieve that we use a multiplier of 1.25". Why exactly does 1.25 cause the "10% effect"? Let's try to reverse engineer the formula in order to understand how it was built. The expression of Eq. (2.1) can be written in a more readable format as follows:

$$w(n) = \frac{2^{10}}{\left(\frac{5}{4}\right)^n} \tag{2.2}$$

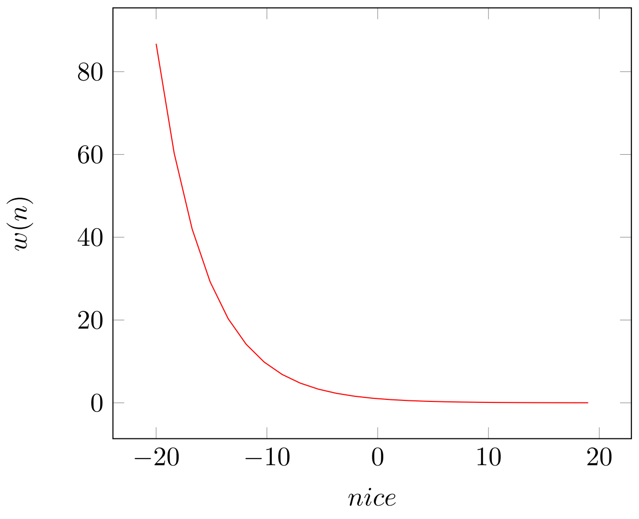


Figure 2.2: Plot of the weight as function of the nice value.

The fraction of CPU time (timeslice) given to a task is equal to its weight divided by the sum of all the other tasks' weights. Let's suppose we have only two processes and they have a difference in nice d, then the CPU percentage of the process with nice n is:

$$CPU_{\%} = \frac{w(n)}{w(n) + w(n+d)} \tag{2.3}$$

By replacing the expression for $w(n)$ of Eq. (2.2) and by setting $\alpha = \frac{5}{4}$, we find

$$\begin{aligned} CPU_{\%} &= \frac{w(n)}{w(n) + w(n+d)} \frac{\dfrac{2^{10}}{\alpha^n}}{\dfrac{2^{10}}{\alpha^n} + \dfrac{2^{10}}{\alpha^{n+d}}} = \\ &= \frac{1}{\dfrac{\alpha^n}{2^{10}} \left(\dfrac{2^{10}}{\alpha^n} + \dfrac{2^{10}}{\alpha^{n+d}} \right)} = \frac{1}{1 + \dfrac{\alpha^n}{\alpha^{n+d}}} = \frac{1}{1 + \dfrac{1}{\alpha^d}} \end{aligned}$$

We have now a function to calculate the $CPU_{\%}$ of a process given a difference d in nice, which is

$$CPU_{\%}(d) = \frac{1}{1 + \left(\dfrac{4}{5}\right)^d}. \tag{2.4}$$

This function is plotted in Figure 2.4. Finally, by computing the derivative of Equation (2.4) at $d = 0$ we find the value of $0.5 \log(5/4) \approx 0.11157$. This is the 10% increase that was referred by the comments in line 7.

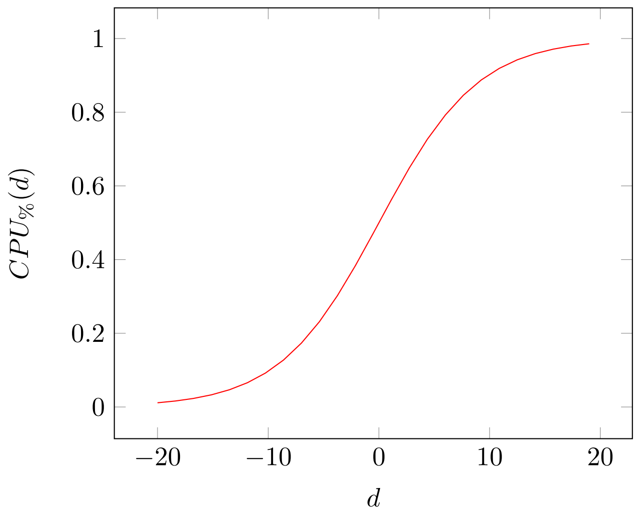


Figure 2.3: Plot of the fraction of CPU assigned to one process among two, assuming they have a difference d of the nice level.

Since this value is not exactly 10%, it is possible to correct the original formula according to the precise value. With the new formula, we can compute the `prio_to_weight` table again, and then recompile the kernel with the new table: this way, the “10% effect” would be even more precise.

2.3.2 Assigned time and virtual runtime

We have defined a function that assigns a weight to each nice level. Let’s see how this weight is used to decide which task to run and for how long. The amount of time that a task can spend on the CPU is determined by 4 values:

- the *Target latency*, which is a tunable value and it’s the period of the scheduler. During this period the scheduler tries to schedule every task once.
- the *Minimum granularity*, which is the minimum amount of time that can be assigned to a task, this is done to prevent small timeslices that would result in a higher switching cost. It is also used to control the behavior of the scheduler. A small value is better for desktop systems that require low latencies, while a large value is better for server workloads.
- the weight of the task, calculated with the weight function discussed in the previous section.
- the total weights of all the task on the run queue.

Given these values, the time assigned to a task is equal to:

$$assigned\_time = target\_latency * \frac{task\_weight}{total\_weight} \tag{2.5}$$

Virtual runtime Remember that CFS tries to model an ideal multi-tasking CPU where each task runs for the same amount of time. In order to know which task deserves to be run next, every task keeps track of the total amount of time that is has spent running. The simplest solution for the scheduler would be to choose the task with the smallest total runtime. But this approach ignores the priorities of the tasks. Instead the absolute runtime of each task is weighted with the weight value discussed before. The weighted time is called the virtual runtime, abbreviated to vruntime. The general formula for calculating it is:

$$vruntime = delta\_exec * \frac{weight\_of\_nice\_0}{task\_weight} \tag{2.6}$$

Where `delta_exec` is the absolute time that the task spent running and `weight_of_nice_0`, as the name suggests, is the weight corresponding to a task with nice zero.

If the nice value of the task is 0, the fraction has value 1 and the virtual runtime is equivalent to the actual time spent running on the CPU. If the nice value is less than 0, then the virtual runtime will be smaller than the actual runtime. This means that the virtual runtime of an high priority task will increase more slowly than the vruntime of a low priority one. So, in order to keep all the virtual runtimes at the same level the high priority task will have to run for more time. On the other hand, if the nice value is more than 0, the virtual runtime will increase faster.

Running the next task As we said before, the goal of CFS is to be as fair as possible with all the task. This means keeping all the task's vruntime as close as possible to each other. Following this logic, the task that deserves more than anyone to be executed next, is the one with the smallest vruntime.

2.3.3 Adding and removing tasks from the runqueue

When a task goes to sleep, it is removed from the runqueue. While sleeping, the virtual runtime of a task remains the same, while the other tasks will continue to increase their own vruntime. If the task is woken up and re-inserted into the runqueue with the same virtual runtime it had before, it would have a significantly higher priority compared to the task that remained on the runqueue: this creates a situation of unfairness, where a sleeping task gets too much priority. A similar situation happens with a newly created task, since it cannot simply have zero as the initial value. If that were the case, the older tasks would not get any CPU for a long time, and the new task would stay on the CPU until its virtual runtime reaches the one of the older tasks.

To avoid this unfairness, the scheduler keeps track of the *minimum virtual runtime* present in the run queue. That is, the virtual runtime of the most deserving active task. Every time that a task is chosen for execution, the minimum is updated. When new tasks are added to the runqueue their virtual runtime is updated to keep things fair. When a sleeping task is woken up, a similar solution is applied: before the task is queued again, the scheduler checks that its virtual runtime is at least equal to the current minimum; if it is not, then it is set to the minimum and the task is inserted into the runqueue. This way the difference in virtual runtime between all the active tasks remains small and no task gets an unfair treatment. When a new task is created via `fork()` and inserted into the runqueue, it inherits the virtual runtime from the parent: this prevents the exploit where a task can take control of the CPU by continuously forking itself.

2.4 Multiprocessing

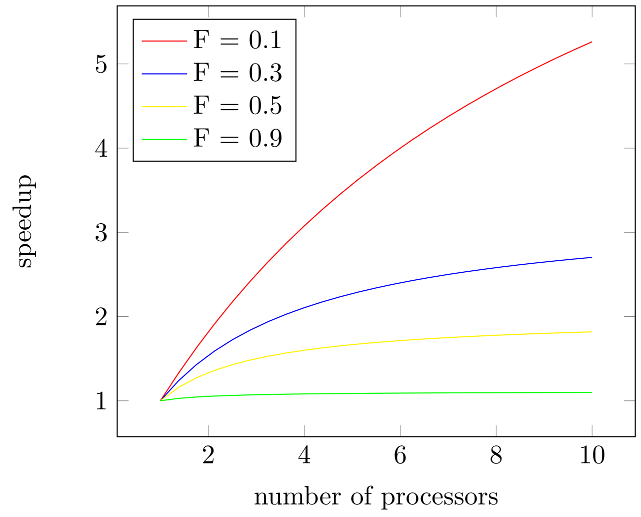
As mentioned in Chapter 1, nowadays most systems have more than one processor. Over the years, the frequencies of processors have been increased in order to achieve better performances, but there are physical limitations to this. Once we hit the limit in CPU frequency, the best way to improve the performance of a system is to add more processors. This allows us to run more processes in parallel and improve the performance, but processes often share resources together and need to communicate with each other (IPC): this factor limits the performance gained by parallel execution.

Theoretical performance gain

The performance gained with a multiprocessing system depends on the parallelizability of the processes: a process that has to wait for the result of another computation cannot be parallelized. Gene Amdahal's law [8] predicts the maximum theoretical improvement of multiprocessing:

$$\mathit{speedup} = \frac{1}{F + \frac{1-F}{N}} \tag{2.7}$$

where F is a factor that represents the portion of the calculation that cannot be parallelized and N is the number of processors.



The graph shows how the performance gained by adding processors depends on the type of the computation. A computation that cannot be parallelized much (high value of F) will have a low performance increase with more processors.

2.4.1 Load balancing

Let's consider a single CPU with multiple cores. To achieve the best possible performance, the workload should be spread evenly between the cores, and there shouldn't be any core that does significantly less work than another. It is up to the scheduler to keep the load of each core balanced.

How is the load measured? The first option is to use the weights of the task. The load of a core would be the sum of the weights of its tasks, but this approach doesn't work very well. Suppose that we have 4 tasks with the same priority, 2 are CPU intensive and 2 spend most of their time sleeping, and we want to balance the load between 2 processors. If we use only the weights of the tasks to balance, both the CPU intensive tasks could get assigned to one core, making it always busy, while the other core would be almost always idle as both of his tasks spend most of their time waiting. This approach doesn't take into consideration the nature of the tasks (CPU bound vs I/O bound). To effectively measure the load, we need to keep track of the amount of time that a task spends sleeping. Hence, the load of a task becomes a combination of its weight and its average CPU utilization, and the load of the a core is the sum of the loads of its tasks. Since the CPU utilization of a task can vary, its load is constantly updated.

Migrating tasks

The scheduler periodically checks that the load of all the CPUs is balanced. In case it is not, the scheduler tries to migrate the tasks from one CPU to another: this operation is called *task migration*. The migration of a task could be expensive and, in some cases, it could be more efficient to not move the task at all. The memory access design of the CPU plays an important role and can change radically how expensive it is to migrate tasks. In a *uniform memory access* (UMA) architecture there is one memory, and the cost of accessing it is the same for all the cores. In this case the tasks can be migrated between cores without any constraints, but having a single memory is not efficient since two cores cannot access the memory at the same time. Modern processors solve this problem by giving each core its own memory, but this creates a more complex structure. In *non-uniform memory access* (NUMA) the cost of accessing the memory is no longer the same for all the cores, but depends on where the desired data is located. Specifically, memory can be local or foreign relatively to a core, and the migration cost changes drastically between the two. This means that sometimes it is more efficient to keep a task on the same core, even if the cores are not balanced.

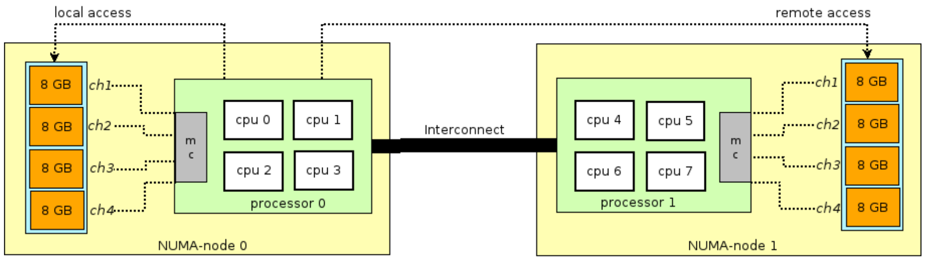


Figure 2.4: Local and foreign memory in a NUMA architecture

The same system could use a mix of the two structures: the cores could be divided in groups with their own memory. Accessing the memory of another group is more expensive, but inside a group all the cores can access the memory at same cost. Inside a group tasks can be migrated more frequently, while migration between groups should be less frequent. To efficiently use this structure the system gives the possibility of defining different migration policies.

Domains The structure of the system is described by two entities: scheduling domains and CPU groups. A scheduling domain is a set of CPUs, and a CPU group is a partition of the CPUs of a domain. A CPU inside a domain cannot belong to two groups of the same domain and every CPU has to be part of a group. The scheduler does not balance the load between single CPUs, instead it tries to balance the loads among the CPU groups of a domain.

Let's take as an example a system with 2 pairs of processors. A pair shares the same memory and its processors can exchange tasks without cost. On the other hand, moving tasks between pairs is more expensive. This could be a real scenario, where a pair represents a single physical processor with inside two hyperthreaded CPUs. Each pair is represented by a domain, which contains two

CPU groups, one for each processor. The groups contains only one CPU in this case. Inside this domain balancing can happen very often since it can be done without additional cost, and it is triggered by small differences in load between the groups of the domain.

To allow the tasks to be migrated from one domain to another, there is also an higher level domain that includes all the CPUs. It is partitioned in two CPU groups, one for each pair. The two groups are kept balanced, but the scheduler does not try to balance the load inside a group. This domain tries to balance the load much less often and it is more tolerant to imbalances in the load. There could also be other layers of domains: for example, in a NUMA system there could be another domain representing a NUMA node, with a different migration policy. By using this domain structure, the scheduler can take in consideration the organization of the system when balancing the CPUs, allowing for more efficient choices.

Chapter 3

Implementation of the Linux scheduler

3.1 Structs and their role

task_struct As mentioned in chapter 1, each task in the system is represented by a `task_struct`, which contains all the information about it. Here we list some of the fields used by the scheduler, and then cover them in detail.

- `thread_info`. This struct is architecture dependant and can contain various fields. The most important for the scheduler is the `flags` field. These flags are used to keep track of requests or signals from and for the task. `TIF_SIGPENDING` means that the process has pending signals, `TIF_MEMDIE` means that the process is being killed to reclaim memory. In particulare, there are two flags that are crucial for the scheduler: `TIF_NEED_RESCHED` and `TIF_POLLING_NRFLAG`. The first indicates that scheduling must be performed, and the second that the idle process is polling the `TIF_NEED_RESCHED` flag.
- `state`. A long that represents the state of the task: -1 unrunnable, 0 runnable, > 0 stopped.
- `on_rq`. Indicates if the process is on the runqueue.
- `static_prio` and `normal_prio`. The first is the priority used for real-time scheduling policies. If the task is scheduled under one of the normal scheduling policies `static_prio` is set to zero and `normal_prio` is used. This number is an internal representation used in the kernel, different from the priority value shown in user space.
- `sched_entity`. This struct contains all the other information needed by the scheduler.

- Pointer to `sched_class`. The scheduling class discussed in chapter 2. Contains a circular list of all scheduling classes.
- Pointer to `mm_struct`. This struct represents the slice of virtual memory used by the process. Here, "`mm`" stands for *memory management*.

thread_info This structure is very small and contains only basic information useful for scheduling, and each process has its own `thread_info`. In figure 1.2 from chapter 1 it is shown that this struct is located at the bottom of kernel stacks. The small size of this struct is due to the fact that it has to fit in the small kernel stacks while not occupying too much space. Today, the implementation of `thread_info` has changed: it is now embedded directly in `task_struct`.

```c
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info  thread_info;
#endif
// ...
```

With the previous implementation, `thread_info` and `task_struct` referenced each other through pointers. As shown in this code, if `CONFIG_THREAD_INFO_IN_TASK` is defined, the field is not a pointer. With this new implementation, `thread_info` is not located in the kernel stack. In order to access it, a pointer to the current task's `thread_info` is saved, and its `task_struct` can be accessed with the "`container_of()`" trick discussed in chapter 1. The implementation is very architecture dependent and this is specific to x86.

sched_entity The `sched_entity` struct represent a *schedulable entity*. From the scheduler's perspective, there is no difference between a process, a thread or even a whole group of processes: they are all just schedulable entities represented by this struct, and scheduler is oblivious to their real nature. Because of this, the scheduler works only with schedulable entities and not `task_struct`s. For example, runqueues contain schedulable entities.

A `sched_entity` may be organized in a hierarchy of entities, this is done to allow group scheduling and it is optional. Suppose that there are more users using a system and we want to share the CPU equally between them, but they have different processes running. With normal CFS, each task is present in the runqueue and it is scheduled independently. By organizing the entities in hierarchies, we can group together the tasks and schedule them as a single schedulable entity. In the previous example we would have only two entities on the runqueue, one for each user, and a corresponding runqueue containing all the task of that user. The scheduler decides which of the two entities to schedule, as if there were only two tasks running, and then repeats the process for the sub-queue. As stated earlier, a `sched_entity` does not always correspond to a process; only the leafs of this structure correspond to a task.

The most important fields that compose `sched_entity` are:

- `load_weight`. It contains the weight of the entity. The role of the weight is discussed in chapter 2.
- `struct rb_node run_node`. Represents a node inside a red-black tree. The next section shows that the runqueue is organized as a red-black tree. This field is embedded in this struct because the implementation is similar to lists.
- `on_rq`. Indicates if the entity is on the runqueue.
- `exec_start`. The time at which the task started its execution on the CPU.
- `sum_exec_runtime`. The total time that the task has actually spent running on the CPU before being weighted.
- `vruntime`. The weighted total time that the task has spent running.
- `struct sched_entity *parent`. The parent node in the hierarchy.
- `struct cfs_rq *cfs_rq`. The runqueue on which this entity is queued.
- `struct cfs_rq *my_q`. If this is a group, this is the sub-runqueue of this group. The children of this entity are queued here.

These fields are specific to CFS, where the runqueue is a red-black tree (`rb_node` and `cfs_rq`), and the timeslice is based on virtual runtime (`exec_start` and `sum_exec_runtime` are used to calculate vruntime). The other scheduling class have their own entity, such as `sched_rt_entity` and `sched_dl_entity`, defining its specific fields. `sched_entity` is not called `sched_fair_entity` supposedly because the fair class is the default.

sched_class This structure contains function pointers to the scheduler routines, specific to the scheduling class. Each class has routines that do the same thing, but with different implementations.

```
struct sched_class {
        const struct sched_class *next; //Circular list of all classes

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int
         ↪  flags);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int
         ↪  flags);
        void (*yield_task)   (struct rq *rq);
        bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool
         ↪  preempt);
        void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int
         ↪  flags);

        /*
         * It is the responsibility of the pick_next_task() method that will
         * return the next task to call put_prev_task() on the @prev task or
         * something equivalent.
         *
         * May return RETRY_TASK when it finds a higher prio class has
         * runnable tasks.
         */
        struct task_struct * (*pick_next_task)(struct rq *rq,
                                               struct task_struct *prev,
                                               struct rq_flags *rf);
    // ...
    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
    void (*update_curr)(struct rq *rq);
    // ...
}
```

When scheduling, the class list is visited, starting from the higher priority class: if there is no process to schedule the next class is visited. This keeps going until something to schedule is found, then the correct function pointers are used to carry out the scheduler's jobs.

This structure is a good reference to understand the code because it essentially lists all the main scheduler routines, which are covered in the next section.

mm_struct TODO mm switched during context switch

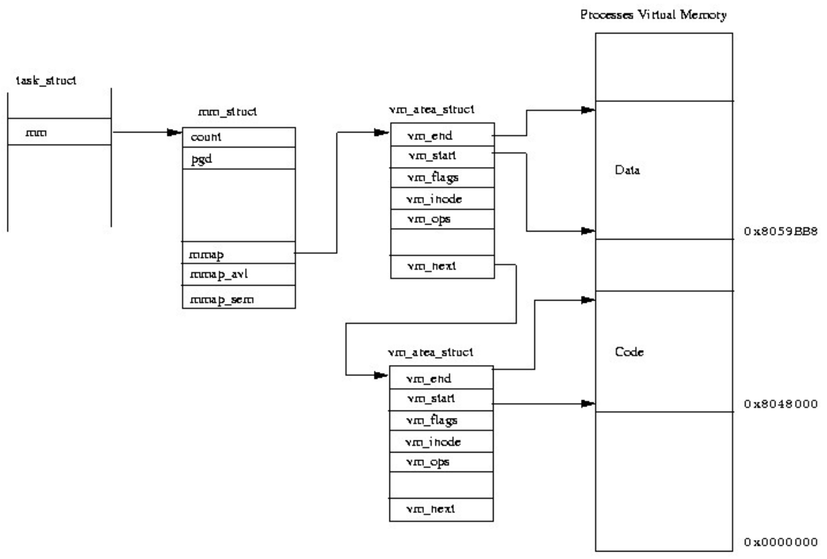


Figure 3.1: The structures used for virtual memory `https://www.tldp.org/LDP/tlk/kernel/processes.html`

procfs The information contained in these structures can be accessed from outside the kernel with specific tools (covered in detail in Chapter 4). One approach to accomplish this it to use `procfs`. In short, `procfs` is a special filesystem that contains all the kernel information about processes, making it easy to access it from user space. For example, it is possible to gather scheduling information about process with pid 1196 by executing `cat /proc/1196/schedstat`. This gives the following human-readable output:

```
tilda (1196, #threads: 3)
-------------------------------------------------------------------
se.exec_start                                :      26964101.112552
se.vruntime                                  :       2272975.687429
se.sum_exec_runtime                          :         10181.684621
se.nr_migrations                             :                11160
nr_switches                                  :                40529
nr_voluntary_switches                        :                40104
nr_involuntary_switches                      :                  425
se.load.weight                               :              1048576
se.avg.load_sum                              :              2453334
se.avg.util_sum                              :              2417183
se.avg.load_avg                              :                   45
se.avg.util_avg                              :                   45
se.avg.last_update_time                      :       26964101112552
policy                                       :                    0
prio                                         :                  120
clock-delta                                  :                   50
mm->numa_scan_seq                            :                    0
numa_pages_migrated                          :                    0
numa_preferred_nid                           :                   -1
total_numa_faults                            :                    0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0
group_private=0 group_shared=0
```

Here, "`se`" is the name of the field of type `sched_entity` in `task_struct`.

Red-black tree The runnable tasks are stored inside a red-black tree, which allows more efficiency. A red-black tree is a type of self-balancing binary search tree: this means that the insertion and deletion operations will keep the height of the tree as small as possible. This characteristic is important because the cost of the most common functions (search, insertion and deletion) is proportional to the height of the tree. As stated before, the task to run next is the one with the smallest vruntime. In a red-black tree, this corresponds to the task in the leftmost node. When picking the next task the tree is not traversed, instead, the leftmost node is cached in a dedicated variable in order to retrieve it faster.

3.2 Time keeping

Many functions of the kernel need to keep track of the passing of the time. The scheduler, for example, needs to know for how long a task has been running in order to know when to preempt it and give the control of the CPU to another task. But there are many other functions that are time-driven, so it's important to understand how the system keeps track of the time.

The kernel requires an hardware timer to manage the passing of time. This timer can be implemented in different ways and there can be more timers in a system, but the general idea is the same: the timer sends an interrupt at a fixed frequency known by the kernel, this way it also knows the time between two timer interrupts and can perform periodic actions such as updating the system uptime. The period between two interrupt is called a *tick* and the frequency is called *tick rate*, this value inside the kernel is defined as **HZ**. If the value of HZ is 100, it means that the frequency is 100HZ and there is a tick every 1/100 seconds, or 10 milliseconds.

Choice of the tick rate

As we will see later, many parts of the system are time-dependant, so they rely on the timer interrupt. Changing the frequency of the interrupt can have a large impact on the behavior of the system, there are pros and cons to larger an smaller values oh HZ.

A larger HZ means that the timer interrupt has a finer resolution, which means that all the timed events also have a higher resolution and also allows to improve the accuracy of those events. As an example, let's consider process preemption, with an higher tick rate we can improve the accuracy and reduce the scheduling latency. Suppose that a process has 2 milliseconds left of its timeslice and the timer interrupt has just occurred, with an HZ of 100 the next interrupt will occur in 10 milliseconds, giving the process 8 extra milliseconds, it also means that another task has to wait for more time in the runqueue and this can be an issue for time-sensitive tasks. With a larger HZ, 1000 for example, in the worst case scenario, the latency is only 1 millisecond.

There is also a drawback to increasing the tick rate: the interrupt handler gets executed more often and results in less processor time for the tasks. The actual impact of the increased overhead is debatable and depends on the speed of the processor.

The value of HZ depends on the hardware and on the configuration of the machine. On i386 HZ had a value of 100, with linux version 2.6 the value was raised to 1000 and later was made a configurable parameter and the default became 250. With the time were also introduced other, more accurate, timers, making an high HZ less necessary. It is also possible to configure the kernel as *tickless* (NO_HZ), this means that the interrupt is no longer at fixed intervals, but it's dynamically scheduled as needed, which is helpful for power savings.

Jiffies

Jiffies is the number of ticks that have occurred since the system started, very time that the interrupt handler is executed, the value of *jiffies* is increased by one. This means that if we know the value of HZ and the value of jiffies we can theoretically convert from seconds to jiffies, simply by doing `seconds * HZ` and from jiffies to seconds with: jiffies / HZ

sched_clock() An important function of the kernel is `sched_clock()`, it returns the system's uptime in nanoseconds. An architecture may provide an implementation, but if it is not provided, the system will use the jiffies counter to calculate the system's uptime.

The scheduler uses this function to determine the absolute time that the current task has been running. If the system uses the jiffies counter to determine this value the maximum resolution of `sched_clock()` depends on the value of HZ. In this case the choice of HZ becomes more relevant. The default implementation of `sched_clock()` using jiffies is this:

```c
unsigned long long __weak sched_clock(void)
{
        return (unsigned long long)(jiffies - INITIAL_JIFFIES)
                                        * (NSEC_PER_SEC / HZ);
}
```

Interrupt Handler

As we said before, the interrupt handler gets called at fixed intervals (unless configured with NO_HZ). The handler perform many important functions, some of this actions depend on the architecture of the system, but some are independent and are executed by the `periodic_tick()` function (in `kernel/time/tick_common.c`):

```c
static void tick_periodic(int cpu)
{
        if (tick_do_timer_cpu == cpu) {
                write_seqlock(&jiffies_lock);

                /* Keep track of the next tick event */
                tick_next_period = ktime_add(tick_next_period, tick_period);

                do_timer(1);
                write_sequnlock(&jiffies_lock);
                update_wall_time();
        }

        update_process_times(user_mode(get_irq_regs()));
        profile_tick(CPU_PROFILING);
}
```

The two most important functions here are `do_timer(1)` and `update_process_times`. The first one increments the value of jiffies and updates the load statistics for the system.

The second is the most important for the scheduler, it updates various statistics and updates the runtime of the current task, if the task has finished its timeslice, it prompts a reschedule. We will see later in more detail how this works.

update_process_times()

This function updates the time that the current process has been running, the process changes if the process was running in user mode or in kernel mode. Remember that `task_struct` has two different fields `utime` and `stime` to keep track of time spent in user space and in kernel space respectively.

The other important thing this function does is invoking the `scheduler_tick` function.

3.3 Scheduler routines

We will now analyze in more detail how the scheduling works and the most important functions involved. In the last section we saw how the system can perform periodic functions at fixed intervals to measure the passing time.

Path that `schedule()` takes. TODO

```bash
#!/bin/bash
echo function_graph > /sys/kernel/debug/tracing/current_tracer
# removing noise to see just the actual function trace
#trace on cpu0 only (actually disables tracing on other cpus)
echo 1 > /sys/kernel/debug/tracing/tracing_cpumask
#show comments on all exit points
echo funcgraph-tail > /sys/kernel/debug/tracing/trace_options
# function to trace
echo $* > /sys/kernel/debug/tracing/set_graph_function
# clear previous trace
echo > /sys/kernel/debug/tracing/trace
sleep 3
cat /sys/kernel/debug/tracing/trace
```

`selected schedule() calls from the script here`

3.3.1 scheduler_tick()

As we mentioned in the previous section the `scheuler_tick()` function is called by the interrupt handler at every tick, this means that it is called with a frequency of HZ.

```c
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        struct rq *rq = cpu_rq(cpu);
        struct task_struct *curr = rq->curr;
        struct rq_flags rf;

        sched_clock_tick();

        rq_lock(rq, &rf);

        update_rq_clock(rq);
        curr->sched_class->task_tick(rq, curr, 0);
        cpu_load_update_active(rq);
        calc_global_load_tick(rq);
        psi_task_tick(rq);

        rq_unlock(rq, &rf);

        perf_event_task_tick();

#ifdef CONFIG_SMP
        rq->idle_balance = idle_cpu(cpu);
        trigger_load_balance(rq);
#endif
}
```

The first important function here is `sched_clock_tick()`, this function updates the per-CPU `sched_clock_data` struct. To update this time, it uses the `sched_clock()` function discussed in the section before 3.2.

The `sched_rq_clock()` updates the `clock_task` field inside the `task_stuct`, this is used later by `update_curr()`.

It then invokes the `task_tick` function for the current task that is running to update the statistics of the task. This action depends on the scheduling class that the current process is using. We will now see how the `fair_sched_class` works.

task_tick_fair

The runtime statistics for the current process are stored in the `sched_entity` associated with it. The scheduler updates its parameters and then check if the current task is to be rescheduled.

Remember that entities can organized in hierarchies. We can now analyze the `task_tick_fair` function:

```
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int
↪   queued)
{
        struct cfs_rq *cfs_rq;
        struct sched_entity *se = &curr->se;

        for_each_sched_entity(se) {
                cfs_rq = cfs_rq_of(se);
                entity_tick(cfs_rq, se, queued);
        }

        if (static_branch_unlikely(&sched_numa_balancing))
                task_tick_numa(rq, curr);

        update_misfit_status(curr, rq);
}
```

This function fetches the `sched_entity` of the current process running, this is going to be a leaf of the hierarchy, and, if the task isn't part of a group, it is also the root. Remember that the `sched_entity` struct point to two runqueues: `cfs_rq` and `my_q`, the first is the runqueue on which the entity is scheduled, the second is the runqueue that belongs to that group. When the entity is a leaf the second field is empty.

For each entity in the hierarchy starting from the leaf, it calls:

- `cfs_rq_of(se)`, which returns the runqueue on which the entity is scheduled, if the system is configured without group scheduling there is only one runqueue.
- then `entity_tick(cfs_rq, se, queued)`, which updates the statistics of that entity and checks if it has finished its time and if another entity deserves to run. This means that if a process inside a group still deserves more time, but the entire group has finished its time and another group deserves the CPU, the task is preempted anyway. `entity_tick` also updates the load statistics used to balance the load between CPUs.

update_curr()

The first part of `entity_tick()`'s job, updating the runtime statistics, it's done by the function `update_curr()`. This function takes as argument only the runqueue on which the current task is running, returned by: `cfs_rq_of(se)`. From the runqueue it gets the `sched_entity` of the current task running. It also gets the current time from the runqueue's clock.

```
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;

        if (unlikely(!curr))
                return;

        delta_exec = now - curr->exec_start;
        if (unlikely((s64)delta_exec <= 0))
                return;

        curr->exec_start = now;

        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);

        if (entity_is_task(curr)) {
                struct task_struct *curtask = task_of(curr);

                trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
                cgroup_account_cputime(curtask, delta_exec);
                account_group_exec_runtime(curtask, delta_exec);
        }

        account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

`curr->exec_start` is the time at which the entity was selected by the scheduler to be executed. We can calculate `delta_exec`, the amount of time that the task has spent running, simply by subtracting `exec_start` from the current time, this value is then used to update the total runtime of the entity (can correspond to a group or a task) and the longest execution.

The virtual runtime is calculated by
`__calc_delta(delta, NICE_0_LOAD, &se->load);`.

If the newly calculated vruntime is also the smallest in the runqueue, the field `cfs_rq->min_vruntime()` is updated.

`update_curr()` also triggers the event `sched_stat_runtime`, this event signals when a process's virtual runtime is updated.

It is also possible that a group of tasks has a limit on the amount of CPU it can use. The function `account_cfs_rq_runtime` updates `cfs_rq->remaining_runtime`, where `cfs_rq` is, again, the runqueue of the current group, and also triggers a reschedule if the remaining time reaches zero.

check_preempt_tick()

The job of `check_preempt_tick()` is to check if the current task has to be preempted. The first step is calculating `ideal_runtime`, as the name suggests, this is the time that is assigned to the task. It's calculated by `__calc_delta(slice, se->load.weight, load)` where the first argument is the target latency, the second is the weight of the entity and the third is the total load of the current runqueue. This function calculates the `assigned_time` of formula 2.5.

If the task has run for more than the ideal runtime, the function `resched_curr()` is called and it will cause the preemption of the task.

The other way it can trigger a reschedule is if the difference between the virtual runtime of the current task and the smallest virtual runtime in the runqueue is bigger than the ideal runtime.

resched_curr() This function simply sets the `TIF_NEED_RESCHED` which indicates that the current task needs to be rescheduled. When resuming the execution of a user process, the flag is check and if it is set, the schedule function is called, this will select the next task to run.

3.3.2 Adding a task to the runqueue

A task can be added to the runqueue in 2 cases: when it is created, when it is woken up after sleeping.

Let's start by describing what happens when a new task is created via the `fork()` system call.

A new process is created

When a process calls `fork()` its `task_struct` is duplicated and a new process is started. This is done by the `_do_fork()` function that also triggers the `sched_process_fork` event. It then calls the function `wake_up_new_task()` that takes as argument the newly created `task_struct`.

This function sets the state of the task as `TASK_RUNNING`, then, after initiating some values used for scheduling statistics, invokes the `activate_task()` function that will proceed in activating the task. After the task has been activated and placed on the runqueue, `wake_up_new_task` triggers the `sched_wakeup_new` event. Lastly this function calls `check_preempt_curr()` that has the job of preempting the current task if another is more deserving.

check_preempt_curr() This function compares the newly created task with the one currently running. The first thing it has to check is the scheduling class. A task of a lower priority scheduling class cannot preempt one of an higher class. On the other hand if the new task has an higher priority, the `resched_curr()` function is called that will cause the task to be rescheduled.

When the two tasks have the same scheduling class, the decision depends on the class. If they use the `fair_sched_class` the `check_preempt_wakeup()` function in `fair.c` is called.

If the current task's policy is `SCHED_IDLE` and the new task has a different policy, then the old task is preempted. Also, a batch or an idle task cannot preempt a non-idle task. If the tasks have the same policy, the virtual runtime of the current task is updated and then compared with the virtual runtime of the new task.

```
static int
wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
{
        s64 gran, vdiff = curr->vruntime - se->vruntime;

        if (vdiff <= 0)
                return -1;

        gran = wakeup_gran(se);
        if (vdiff > gran)
                return 1;

        return 0;
}
```

If the current task has a smaller virtual runtime than the new one, the task is not preempted. Otherwise it is preempted if the difference between the virtual runtimes is big enough. This is done to avoid too frequent rescheduling.

The scheduler has a tunable parameter called `wakeup_granularity` that controls the minimum difference necessary to preempt the task. This value is not directly used, instead it's first weighted by the weight of the new task with the `calc_delta_fair()` function. This is the same function used to calculate the virtual runtime. The new task is used, instead of the current one, because this penalizes tasks with smaller weights:

- if the new task has a smaller weight than the current one, then the corresponding weighted granularity will be larger. This means that it is harder for the new task to preempt the current one.
- on the contrary, if the new task has a larger weight, it will be easier to preempt the current task.

If the current task needs to be rescheduled, the `check_preempt_wakeup()` function, calls `set_next_buddy()` that tells to scheduler to favor the newly added task when choosing the next task. Finally the `resched_curr()`3.3.1 function is called.

A task is woken up

Waking up a process is similar to creating a new one: the task is first inserted into the runqueue and then the system checks if the currently running task needs to be rescheduled.

Before inserting the task into the runqueue, the `sched_waking` event is triggered and the task's state is set to `TASK_WAKING`. The task is then inserted into the runqueue by `activate_task()`. Once it is on the runqueue the `task_struct->on_rq` field is set to `TASK_ON_RQ_QUEUED`. Finally the function `ttwu_do_wakeup()` is called, which checks if the current task can be preempted with `check_preempt_curr()`3.3.2, sets the task's state to `TASK_RUNNING` and triggers the `sched_wakeup` event.

enqueue_task()

Whether the task was just created or it was woken up from sleep, his `task_struct` has to be added to the runqueue. This is done by the `enqueue_task()` function. It takes as arguments a pointer `rq` to the runqueue in which to insert the task, a pointer `p` to the `task_struct` and an int representing the flags. It updates the `cfs_rq` utilization statistics and invokes the `enqueue_entity()` function that:

- calls `updats_curr()` adds `cfs_rq->min_vruntime` to `se->vruntime`, otherwise the new task would have too large boost compared to running tasks. Note that the virtual runtime is decreased when dequeuing a task.
- updates the load statistics of the entity and its group
- if `ENQUEUE_WAKEUP` calls `check_spread`
- calls `__enqueue_entity()` that insert the entity in the red-black tree representing the runqueue.

schedule()

context_switch()

Chapter 4

Tracing with ftrace

Kernel debugging is a big challenge even for the most experienced kernel developers. The problem is that if the system has, for example, latencies or synchronization issues (undetected race conditions), it's really hard to pinpoint where they're coming from. Which subsystems are involved? In which conditions does the problem arise? When the system is running, not always there is a way to know the answer. Ftrace is a debugger designed specifically to solve the issue and make the developer's life easier. It's also a great educational tool, not just to peek at what happens in the kernel, but also to help approach the source code by observing the function flow.

The name comes from "function tracer", which is one of its features, but it has many others. Each mode of tracing is simply called a *tracer*, and each one comes with many options to tweak it. They can do function tracing, event tracing, measure context switch time or the time in which interrupts are disabled. Ftrace is also very extensible because it's possible to write new tracers that can be added like a module.

As anticipated, one of the objectives of the thesis is to document events related with scheduling. To understand what they do and why they are useful, it's necessary to understand ftrace, which is the tool that uses them.

4.1 How does it work?

Tracing means recording events that occur at runtime in order to analyze the code's behavior and performance. More generally, this is called *software profiling* ans it's implemented with different techniques. In our case, it's achieved by the means of *code instrumentation*, which consists in adding instructions to the source code or its binary in order to profile it. There are two main ways of using ftrace, and they use two different instrumentation techniques:

- Function tracing, using `gcc`'s code instrumentation mechanism activated by compiling with the `-pg` option.

- Event tracing, using *tracepoints* in the source code.

Function tracing uses a form of dynamic profiling: this means that the tracing instructions can be toggled at runtime in the binary executable, without the need to recompile the code. The way this works is that, while compiling, `gcc` adds extra `NOP` assembly instructions at the beginning of every function. The position of these instructions is then saved in the binary itself, so that it will be possible to change these `NOP`s into something else. This is exactly what ftrace does: it toggles tracing by changing these instructions at runtime; they are converted to `JMP` instructions to tracing functions, and then back to `NOP` to disable tracing. For this reason, this instrumentation technique is called *runtime injection*. This approach has two main advantages:

- Since we can toggle tracing at runtime, there is zero overhead when it's disabled (so 99% of the time).
- It's possible to filter what is being traced: we could dynamically activate tracing only on functions from a single subsystem, or on one function alone.

Event tracing, on the other hand, is a little different and it's less efficient than function tracing because it doesn't use runtime injection. Instead, it uses tracepoints directly in the c code, which makes it static. Tracepoints are simply direct calls to tracing function, which will gather some information through parameters and then write it in the trace output, along with the event name. Since this mechanism is static, the whole kernel must be recompiled to toggle the tracepoints: this is done by simply toggling the `CONFIG_TRACEPOINTS` macro in the configuration before compiling. Let's take two scheduling events as an example: `sched_stat_runtime` and `sched_migrate_task`. The first happens in a specific point of the scheduler code and contains core scheduling information about a given process; the second happens upon migration of a task and contains information such as the CPU the thread was migrated to. They are called in the code like this:

```
// curtask and curr are task_structs, delta_exec is the difference in runtime
↪   since the last timer interrupt.
trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
// p is a task_struct, new_cpu is the CPU the thread migrated to
trace_sched_migrate_task(p, new_cpu);
```

When these events happen, they will appear in the trace output like this:

```
# The format is name, pid, cpu, timestamp, event name, event information
AudioIPC-1849  [002] 21448.743195: sched_stat_runtime:   comm=AudioIPC Callba pid=1849 runtime=104248 [ns] vruntime=34625082783
AudioIPC-1849  [002] 21448.743174: sched_migrate_task:   comm=AudioIPC Client pid=26778 prio=120 orig_cpu=3 dest_cpu=0
```

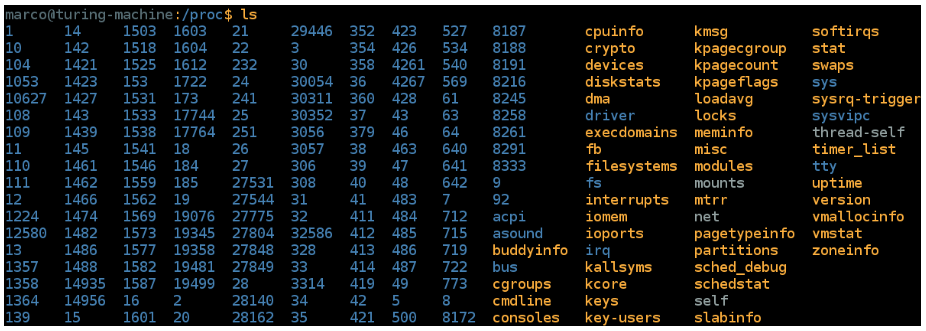
Notice how all the information can be found through the parameters: command, pid, runtime and priority can all be found directly into the `task_struct`.

So what do the tracing functions do, exactly? Ftrace uses a ring buffer to store all the events that are happening at runtime, so these functions will write new events in the buffer. Essentially, the producer is the kernel and the consumer is the user, which will read from user space (we will shortly see how). Because it's a circular buffer, all the old entries are overwritten if they are not read in time: this happens all the time with events because, since boot, they are written in the buffer even if nobody is reading. Another scenario in which entries are lost is when they are getting written faster than they are read: this is common when we trace every single function/event without any filters. It's generally good practice to filter as much as possible to avoid losing entries, which is possible with dynamic tracing, but not with static tracing. It's true that we can easily filter the trace output, but with static tracing the entries will be written in the buffer anyway, resulting in potential overwriting: this is why with function tracing we have true filtering, but not with event tracing.

4.2 Interfacing with ftrace

While tracing, the events that need to be monitored are so frequent that an extremely lightweight mechanism is needed. Ftrace offers this possibility because it's self-contained and entirely implemented in the kernel, requiring no user space tools whatsoever. We said earlier that the ftrace output, which is produced from the kernel, is read from user space: how can we read it without a specialized program?

On Unix, system calls are not the only way to interact with the kernel. Another solution is to use a dedicated special filesystem on which the kernel and the user can easily read/write: this creates a sort of shared memory between the user and the kernel. This practice is very common on Unix-like systems such as Linux; so common, in fact, that kernel process information is (almost) always accessed like this. This is done through the `procfs` filesystem, which is found in `/proc`, as shown in figure 4.1: every information about processes is stored here and it's fully accessible from user space. You can see that there is some generic information and also per-process information, with a folder for each current pid.

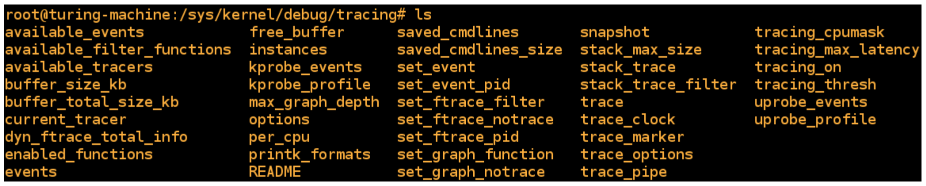


```
marco@turing-machine:/proc$ ls
1      14     1503  1603   21     29446  352  423   527   8187       cpuinfo      kmsg          softirqs
10     142    1518  1604   22     3      354  426   534   8188       crypto       kpagecgroup   stat
104    1421   1525  1612   232    30     358  4261  540   8191       devices      kpagecount    swaps
1053   1423   153   1722   24     30054  36   4267  569   8216       diskstats    kpageflags    sys
10627  1427   1531  173    241    30311  360  428   61    8245       dma          loadavg       sysrq-trigger
108    143    1533  17744  25     30352  37   43    63    8258       driver       locks         sysvipc
109    1439   1538  17764  251    3056   379  46    64    8261       execdomains  meminfo       thread-self
11     145    1541  18     26     3057   38   463   640   8291       fb           misc          timer_list
110    1461   1546  184    27     306    39   47    641   8333       filesystems  modules       tty
111    1462   1559  185    27531  308    40   48    642   9          fs           mounts        uptime
12     1466   1562  19     27544  31     41   483   7     92         interrupts   mtrr          version
1224   1474   1569  19076  27775  32     411  484   712   acpi       iomem        net           vmallocinfo
12580  1482   1573  19345  27804  32586  412  485   715   asound     ioports      pagetypeinfo  vmstat
13     1486   1577  19358  27848  328    413  486   719   buddyinfo  irq          partitions    zoneinfo
1357   1488   1582  19481  27849  33     414  487   722   bus        kallsyms     sched_debug
1358   14935  1587  19499  28     3314   419  49    773   cgroups    kcore        schedstat
1364   14956  16    2      28140  34     42   5     8     cmdline    keys         self
139    15     1601  20     28162  35     421  500   8172  consoles   key-users    slabinfo
```

Figure 4.1: The procfs special filesystem

If we were to write a user application to display processes; the alternative approach to get this information would be to use a special-purpose syscall, which is what BSD and MacOS do: the syscall will return a kernel structure with all the information that needs to be parsed. The approach used by Linux is more straightforward: the information is (mostly) in human-readable form, so you simply read the files in `/proc` and parse the results as strings. By doing this, you don't need to use any syscall, except, of course, `open()` and `read()` to interact with the filesystem. On Linux, when you use commands like `ps`, `top` or `pgrep`, what they do internally is to query `procfs`. You could always do the same operation manually by doing something like `cat /proc/1337/info_that_you_need | grep specific_info`, but it would be tedious: this is why utilities like `ps` are essentially front-ends for the user.

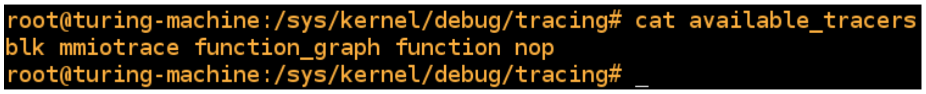
There are also other specialized filesystems, for example `sysfs`, which contains system information; but what iterests us is `debugfs`, which contains kernel debug information: it's here that we can interact with ftrace. This filesystem is mounted by executing `mount -t debugfs nodev /sys/kernel/debug/`: since there is not an actual device that is being mounted, we use "`nodev`" as target device; `/sys/kernel/debug/` is the target mount point. In figure 4.2 you can see the trace folder located in this filesystem. To interact with ftrace you simply write in these files with `echo your_value > file`: by doing this you can toggle options and set parameters before/during the trace.



```
root@turing-machine:/sys/kernel/debug/tracing# ls
available_events          free_buffer      saved_cmdlines       snapshot            tracing_cpumask
available_filter_functions instances       saved_cmdlines_size  stack_max_size      tracing_max_latency
available_tracers         kprobe_events    set_event            stack_trace         tracing_on
buffer_size_kb            kprobe_profile   set_event_pid        stack_trace_filter  tracing_thresh
buffer_total_size_kb      max_graph_depth  set_ftrace_filter    trace               uprobe_events
current_tracer            options          set_ftrace_notrace   trace_clock         uprobe_profile
dyn_ftrace_total_info     per_cpu          set_ftrace_pid       trace_marker
enabled_functions         printk_formats   set_graph_function   trace_options
events                    README           set_graph_notrace    trace_pipe
```

Figure 4.2: Tracing folder inside the debugfs special filesystem

Some of these files' purpose is not to set options, but rather to list available options. For instance, in figure 4.3, we list the available tracers. These are essentially tracing modes: we activate one by doing `echo function > current_tracer`, which will immediately start the trace with the "function" tracer. We can then see the trace output by simply executing `cat trace`. Most of the other files are used for filtering what is being traced, which we will see in detail in the upcoming section.



```
root@turing-machine:/sys/kernel/debug/tracing# cat available_tracers
blk mmiotrace function_graph function nop
root@turing-machine:/sys/kernel/debug/tracing# _
```

Figure 4.3: Types of tracers, only a few are available by default on by distribution (Debian)

Let's now see different ways to interact with ftrace:

```
# Interfacing through an application program
sudo trace-cmd record -p function -P 622
sudo trace-cmd report
# Interfacing through the filesystem
cd /sys/kernel/debug/tracing
echo 622 > set_ftrace_pid
echo function > current_tracer
cat trace
```

This is a good example of the different ways of communication from user to kernel space. In this code, both approaches trace the process with pid 622, and they essentially do it in the same way because `trace-cmd` simply queries `debugfs`, just like `ps` queries `procfs`. We will use the second approach because it shows explicitly how we interface with the kernel, but in practice it's sometimes easier to use tools like `trace-cmd`. Another useful tool is `kernelshark`, which has a GUI to show graphs of the traces done through `trace-cmd`.

4.3 Ftrace usage

4.3.1 Function tracing

Let's write a simple script that traces any input process.

```
#!/bin/bash
# traceprocess.sh
echo $$ > /sys/kernel/debug/tracing/set_ftrace_pid
# echo every function to filter
echo __do_page_fault >
↪  /sys/kernel/debug/tracing/set_ftrace_filter
echo function > /sys/kernel/debug/tracing/current_tracer
exec $*
```

`$$` is the variable that contains the pid of the script itself, and `$*` are the arguments of the script: in this case, the process to trace. The way it works is very simple:

1. Set this pid as the one that will be traced
2. Set the `__do_page_fault` as the only function to trace
3. Set the tracer to the function tracer

4. Execute the input program
5. The executed program will be a child of the script itself, so its pid will automatically be added to `set_ftrace_pid` and it will be traced

In the kernel, every routine that starts with "`do_`" is an interrupt handler: in this case, we traced the interrupt handler for page faults by using a filter. Usually, we would see every kernel function that the input process calls, which is sometimes a big and uninformative output that needs filtering. The trace output can be found in the file `/sys/kernel/debug/tracing/trace`, or can be viewed as it gets written in `/sys/kernel/debug/tracing/trace_pipe`. The following is an output of `./traceprocess.sh ls`, which traces `ls`.

```
# tracer: function
#
# entries-in-buffer/entries-written: 92/92   #P:4
#
#                      _-----=> irqs-off
#                     / _----=> need-resched
#                    | / _---=> hardirq/softirq
#                    || / _--=> preempt-depth
#                    ||| /     delay
#   TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#      | |       |   ||||       |         |
      ls-4973  [000] d... 14386.659663: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.659718: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.659743: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.659784: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.659800: __do_page_fault <-page_fault
      # ... many more page faults ...
      ls-4973  [000] d... 14386.662473: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.662871: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.662881: __do_page_fault <-page_fault
      ls-4973  [000] d... 14386.662900: __do_page_fault <-page_fault
```

As expected, we only see page faults (for a total of 92). This information is not that useful by itself, but what is useful, instead, are the timestamps: with these, it's easy to detect latencies in the kernel. By using kernelshark you can plot the trace in order to make latencies obvious; doing this can also be interesting because it lets you see which actions cause most overhead. Another way of doing this just with ftrace is to use the `function_graph` tracer: it's similar to the `function` tracer, but it shows the entry and exit point of each function, creating a function call graph. Instead of timestamps it shows the duration of each function execution. The symbols `+`, `!` `#` are used whenever there is an execution time greater than 10, 100 and 1000 microseconds. As we know, scheduling and thread migration cause a lot of overhead, so we can try to use `function_graph` to see it.

```
 2)               |           schedule() {
 2)   0.033 us    |             rcu_note_context_switch();
 2)   0.028 us    |             _raw_spin_lock();
 2)               |             deactivate_task() {
 2)   0.032 us    |               update_rq_clock.part.84();
 2)               |               dequeue_task_fair() {
 2)               |                 dequeue_entity() {
 2)               |                   update_curr() {
 2)   0.030 us    |                     update_min_vruntime();
 2)   0.060 us    |                     cpuacct_charge();
 2)   0.654 us    |                   }
 2)   0.029 us    |                   clear_buddies();
 2)   0.033 us    |                   account_entity_dequeue();
 2)   0.041 us    |                   update_cfs_shares();
 2)   0.027 us    |                   update_min_vruntime();
 2)   2.188 us    |                 }
 2)   0.030 us    |                 hrtick_update();
 2)   2.767 us    |               }
 2)   3.362 us    |             }
 2)               |             pick_next_task_fair() {
 2)   0.028 us    |               __msecs_to_jiffies();
 2)   0.353 us    |             }
 2)               |             pick_next_task_idle() {
 2)               |               put_prev_task_fair() {
 2)               |                 put_prev_entity() {
 2)   0.029 us    |                   check_cfs_rq_runtime();
 2)   0.323 us    |                 }
 2)   0.608 us    |               }
 2)   0.036 us    |               update_idle_core();
 2)   1.208 us    |             }
 2)   0.033 us    |     finish_task_switch();
 2) ! 114.732 us  |   } /* schedule */
 2) ! 115.042 us  | } /* schedule_preempt_disabled */
```

This is small piece of a trace of every function call on my system, without function or process filters. Function duration is located at every leaf function and function exit point (`}`): as you can see `schedule()` takes longer to execute than the other functions; there is also a `!` because it's more that 100 microseconds. As we said earlier, the buffer can be filled and some entries will be lost: this is very common if you trace everything without filtering, which is what we did here.

The ftrace documentation says "The function name is always displayed after the closing bracket for a function if the start of that function is not in the trace buffer". In our case, this means that the exit point "`} /* schedule */`" is not referring to the initial `schedule()` entry point! Even though we can see the overhead of the function, the actual entry point is not there because it couldn't get written in the trace. To mitigate this we can trace on a single CPU, instead of all 4. This approach has three advantages:

- The output won't have function calls interleaved between the CPUs, which breaks the flow of function calls
- Since fewer entries are traced, the buffer won't be filled and many won't be lost
- There is a performance gain: tracing every single function call generates significant overhead

In general, it's better to narrow the filters as much as possible. For example, it would be good to trace only the function that we're interested in, and on one CPU only: in the next chapter, we will always trace this way in order to reduce noise.

Let's try to trace only the `schedule()` function, on all CPUS, just to see how much time it can take (on my machine, that is). We do this by executing:

```
cd /sys/kernel/debug/tracing/
echo schedule > set_graph_function
cat trace | grep -F "/* schedule */"
```

The output is:

```
+ means that the function exceeded 10 usecs.
! means that the function exceeded 100 usecs.
# means that the function exceeded 1000 usecs.
* means that the function exceeded 10 msecs.
@ means that the function exceeded 100 msecs.
$ means that the function exceeded 1 sec.

 2) + 58.121 us   |  } /* schedule */
 1) + 68.348 us   |  } /* schedule */
 2) @ 991933.0 us |      } /* schedule */
 1) @ 992178.9 us |      } /* schedule */
 1) * 31760.76 us |      } /* schedule */
 3) ! 139.005 us  |  } /* schedule */
 3) $ 1147687 us  |      } /* schedule */
 2) + 49.196 us   |  } /* schedule */
 3) # 1243.739 us |  } /* schedule */
 2) * 97870.13 us |  } /* schedule */
 0) + 39.666 us   |  } /* schedule */
 3) + 63.518 us   |  } /* schedule */
 0) # 1193.975 us |  } /* schedule */
 0) ! 345.386 us  |  } /* schedule */
 3) + 74.291 us   |  } /* schedule */
 0) # 1381.706 us |  } /* schedule */
 0) ! 113.232 us  |  } /* schedule */
 0) ! 106.633 us  |  } /* schedule */
 2) # 1652.529 us |  } /* schedule */
 1) $ 1023917 us  |      } /* schedule */
 3) @ 935896.8 us |      } /* schedule */
 2) $ 1123576 us  |      } /* schedule */
 1) * 67530.47 us |      } /* schedule */
 0) + 88.367 us   |  } /* schedule */
 1) # 1585.574 us |  } /* schedule */
 2) @ 231657.1 us |      } /* schedule */
 2) ! 389.683 us  |  } /* schedule */
 0) + 70.527 us   |  } /* schedule */
 1) # 1461.529 us |  } /* schedule */
 3) ! 120.510 us  |  } /* schedule */
 0) + 90.422 us   |  } /* schedule */
 2) # 1433.207 us |  } /* schedule */
 2) @ 307235.2 us |      } /* schedule */
 3) $ 1063775 us  |  } /* schedule */
```

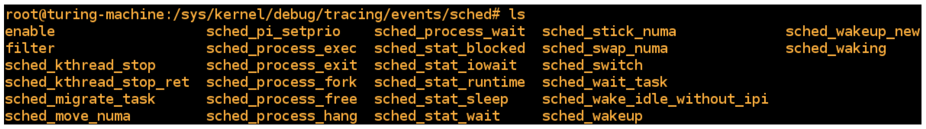
The output is not the average time, but rather an unordered mix with a prevalence of the longest times recorded. The reason is that we looked for commented exit points, so these are actually only the exit points that don't have an entry point in the trace (as stated by the documentation). There are some cases where it took more than 1 second ("`$`") to execute: these are extreme cases where the schedule got interruped and some other kernel task was done in the meantime, so it didn't actually take a whole second **just** to schedule. If we look for every exit point—not just the orphaned ones—we can see that on average the `schedule()` routine will take less time, but still more than the other functions.

There is also another problem with this trace. Steven Rostedt, the creator of ftrace, said in one of its articles that "Only the leaf functions, the ones that do not call other functions, have an accurate duration, since the duration of parent functions also includes the overhead of the `function_graph` tracer calling the child functions".[9] This means that taking the difference between the timestamp of the entry and exit point is not enough, since the overhead of ftrace is not taken into account. The same article says "By using the `set_ftrace_filter` file, you can force any function into becoming a leaf function in the `function_graph` tracer, and this will allow you to see an accurate duration of that function". If we do that we find out, more accurately, that most of the time it will take between $20\mu s$ and $400\mu s$ to execute `schedule()`.

4.3.2 Event tracing

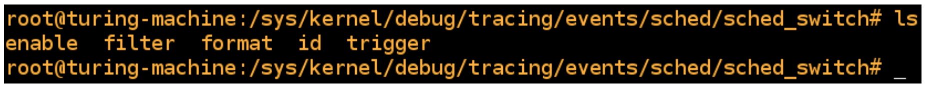
Function tracing is very useful and will come in handy to understand the code, but now we will focus on events. You may have noticed in figure 4.2 that there is a directory called "events". It contains a folder for each *event subsystem*, and the one we're interested in is `sched`, for the scheduling subsystem. Figure

4.4 shows its contents: there is a folder for each event, containing information about it and a switch to enable/disable it. This is essentially a list of the events that we're going to document, even though some of their names are almost self explainatory.



```
root@turing-machine:/sys/kernel/debug/tracing/events/sched# ls
enable                 sched_pi_setprio    sched_process_wait  sched_stick_numa             sched_wakeup_new
filter                 sched_process_exec  sched_stat_blocked  sched_swap_numa              sched_waking
sched_kthread_stop     sched_process_exit  sched_stat_iowait   sched_switch
sched_kthread_stop_ret sched_process_fork  sched_stat_runtime  sched_wait_task
sched_migrate_task     sched_process_free  sched_stat_sleep    sched_wake_idle_without_ipi
sched_move_numa        sched_process_hang  sched_stat_wait     sched_wakeup
```

Figure 4.4: Every event associated with scheduling



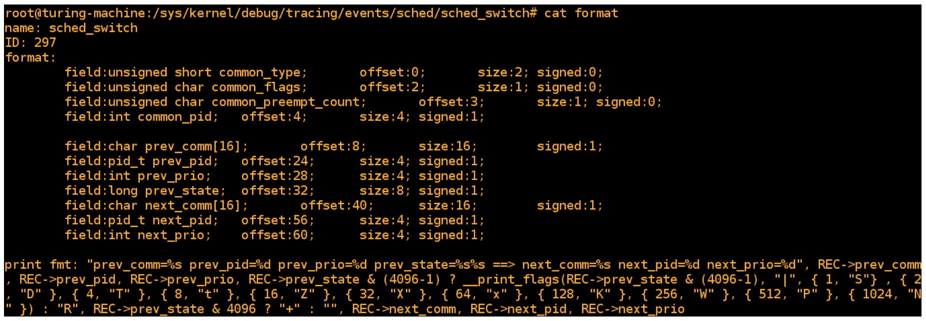
```
root@turing-machine:/sys/kernel/debug/tracing/events/sched/sched_switch# ls
enable  filter  format  id  trigger
root@turing-machine:/sys/kernel/debug/tracing/events/sched/sched_switch# _
```

Figure 4.5: Control files for the sched_switch event

So what are events, exactly? As anticipated, event tracing is much more static compared to function tracing. What this means is that event *tracepoints* are directly embedded in the code and are called just like functions, so they cannot be toggled at runtime and you need to recompile the whole kernel to change/disable them. From this perspective, events are really similar to regular prints, but in practice events are much more efficient than `printk()`.

Steven Rostedt explains pretty well why that is the case: "`printk()` is the king of all debuggers, but it has a problem. If you are debugging a high volume area such as the timer interrupt, the scheduler, or the network, `printk()` can lead to bogging down the system or can even create a live lock. It is also quite common to see a bug "disappear" when adding a few `printk()`s. This is due to the sheer overhead that `printk()` introduces. Ftrace introduces a new form of `printk()` called `trace_printk()`. It can be used just like `printk()`, and can also be used in any context (interrupt code, NMI code, and scheduler code). What is nice about `trace_printk()` is that it does not output to the console. Instead it writes to the Ftrace ring buffer and can be read via the trace file. Writing into the ring buffer with `trace_printk()` only takes around a tenth of a microsecond or so. But using `printk()`, especially when writing to the serial console, may take several milliseconds per write."[10] `trace_printk()` simply writes a message in the trace buffer, which is exactly what happens with events, just with a pre-defined format and many printed fields. Because of this, what is said in the quote also applies to events. In figure 4.6 you can see how similar to a print an event actually is. Each event has this format file which states fields and print formatting, with the same syntax of `printk()`. In the upcoming section, we will see how to declare these properties for an event from the kernel.

Let's now see how event tracing is enabled and how to filter events. Events are not related with any tracer because tracers are used for dynamic tracing only. If we want to see just the events, then we must use the `nop` tracer (which



```
root@turing-machine:/sys/kernel/debug/tracing/events/sched/sched_switch# cat format
name: sched_switch
ID: 297
format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3;       size:1; signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;

        field:char prev_comm[16];       offset:8;       size:16;        signed:1;
        field:pid_t prev_pid;   offset:24;      size:4; signed:1;
        field:int prev_prio;    offset:28;      size:4; signed:1;
        field:long prev_state;  offset:32;      size:8; signed:1;
        field:char next_comm[16];       offset:40;      size:16;        signed:1;
        field:pid_t next_pid;   offset:56;      size:4; signed:1;
        field:int next_prio;    offset:60;      size:4; signed:1;

print fmt: "prev_comm=%s prev_pid=%d prev_prio=%d prev_state=%s%s ==> next_comm=%s next_pid=%d next_prio=%d", REC->prev_comm
, REC->prev_pid, REC->prev_prio, REC->prev_state & (4096-1) ? __print_flags(REC->prev_state & (4096-1), "|", { 1, "S"} , { 2
, "D" }, { 4, "T" }, { 8, "t" }, { 16, "Z" }, { 32, "X" }, { 64, "x" }, { 128, "K" }, { 256, "W" }, { 512, "P" }, { 1024, "N
" }) : "R", REC->prev_state & 4096 ? "+" : "", REC->next_comm, REC->next_pid, REC->next_prio
```

Figure 4.6: Fields and print format of the sched_switch event

doesn't trace anything), but we could also trace events while tracing functions by enabling any other tracer.

```
# enable scheduling events
echo nop > /sys/kernel/debug/tracing/current_tracer
echo 1 > /sys/kernel/debug/tracing/events/sched/enable
# enable just the sched_switch event
echo nop > /sys/kernel/debug/tracing/current_tracer
echo 1 >
↪  /sys/kernel/debug/tracing/events/sched/sched_switch/enable
```

The "enable" file is located in every folder of the event directory tree. As you can see, the directory hierarchy is used to toggle single events, entire event subsystems, or all the existing events. Be aware that this filter doesn't stop the events from being written in the trace buffer, we are just ignoring them. "You have to recompile the whole kernel to disable specific events" can be paraphrased as "You have to recompile the whole kernel to prevent ftrace from writing specific events in its buffer, even when they are disabled from `debugfs`".

The following is a small piece of a trace of every scheduling event:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 116546/459475   #P:4
#
#                      _-----=> irqs-off
#                     / _----=> need-resched
#                    | / _---=> hardirq/softirq
#                    || / _--=> preempt-depth
#                    ||| /     delay
#   TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#      | |       |   ||||       |         |
  <idle>-0     [000] d...   611.283814: sched_switch: prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==> next_comm=
    Xorg-1450  [000] d...   611.283921: sched_stat_runtime: comm=Xorg pid=1450 runtime=117083 [ns] vruntime=17539094302 [ns]
    Xorg-1450   [000] d...   611.283924: sched_switch: prev_comm=Xorg prev_pid=1450 prev_prio=120 prev_state=S ==> next_comm=swa
  <idle>-0      [000] d...   611.283957: sched_switch: prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==> next_comm=X
    # ... many more entries ...
```

In this trace, the swapper process (pid 0) was switched out to schedule Xorg, which is the display server (essentially, the GUI) of the system, and then back again to the swapper; all in a matter of $143\mu s$. `sched_switch` and `sched_stat_runtime` are the most common scheduling events. The first reports when a process switch happens, by printing information about the old and new process, and the second prints core scheduling information of the running process, such as pid, actual runtime and virtual runtime. The tracepoints for these events look like this in the code:

```
trace_sched_switch(preempt, prev, next);
trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
```

As we said earlier, every information printed out in the trace is found through the parameters.

4.4 Creating new events

Tracepoints are created from within the kernel. At this level, events are seen as structures which carry the information needed for the event. A tracepoint must create this structure, fill it with data and write it in the ftrace ring buffer. So, this buffer is basically an array of binary data, which will be periodically consumed, decoded and printed as a string in the trace output.

Besides the core infrastructure of ftrace, its developers have also made iterfaces to make it easy for kernel developers to create tracepoints. Today, this process is almost completely automated thanks to the `TRACE_EVENT()` macro and other sub-macros called by it. By using `TRACE_EVENT()` it's possible to quickly create tracepoints in the core kernel code, or even in a kernel module, without much boilerplate code. Every existing tracepoint for scheduling events is created by using this macro: these tracepoints are declared in a separate, dedicated, header file, which is then included in the main code. The default path for these headers is `include/trace/events`, so the declarations for the scheduling events are in `include/trace/events/sched.h`. Let's see what's in this header.

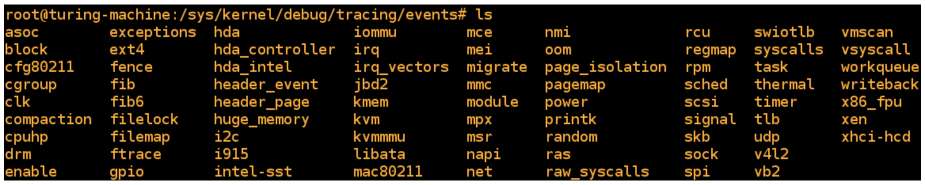
```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM sched
#if !defined(_TRACE_SCHED_H) || defined(TRACE_HEADER_MULTI_READ) // Special
 ↪  guard
#define _TRACE_SCHED_H
// ... some other includes ...
#include <linux/tracepoint.h> //TRACE_EVENT() defined in here, then redefined
 ↪  at the end of this trace header
/*
 * Tracepoint for a task being migrated:
 */
```

```
TRACE_EVENT(sched_migrate_task,
        TP_PROTO(struct task_struct *p, int dest_cpu),
        TP_ARGS(p, dest_cpu),
        TP_STRUCT__entry(
                __array(char, comm,        TASK_COMM_LEN)
                __field(pid_t, pid)
                __field(int, prio)
                __field(int, orig_cpu)
                __field(int, dest_cpu)
        ),
        TP_fast_assign(
                memcpy(__entry->comm, p->comm, TASK_COMM_LEN);
                __entry->pid = p->pid;
                __entry->prio = p->prio; /* XXX SCHED_DEADLINE */
                __entry->orig_cpu = task_cpu(p);
                __entry->dest_cpu = dest_cpu;
        ),
        TP_printk("comm=%s pid=%d prio=%d orig_cpu=%d dest_cpu=%d",
                  __entry->comm, __entry->pid, __entry->prio,
                  __entry->orig_cpu, __entry->dest_cpu)
);
// ... many more event declarations ...
#endif /* _TRACE_SCHED_H */
/* This part must be outside protection */
#include <trace/define_trace.h>
```

The whole file is basically a list of declarations like this one. This is the declaration of `sched_migrate_task`, which was shown in section 4.1. These weird includes at the end of the file and the special guard have something to do with how `TRACE_EVENT()` works internally; we will give a brief overview of that at the end of the section. The macro itself has 6 parameters, with 5 different sub-macros:

- The first is the tracepoint name, the final name will have the format `trace_NAME`.
- `TP_PROTO` and `TP_ARGS` simply define the arguments of the tracepoint
- `TP_STRUCT__entry` defines the struct of the event, with every attribute type and name. There are 2 different sub-sub-macros to define fields and arrays.
- `TP_fast_assign` defines how to fill the event struct, usually with the event parameters. This is not trivial because they can be filled by using other functions, macros or by `memcpy()`, `memmove()` etc...
- `TP_printk` has the same syntax of `printk()` and defines how to print the struct in human-readable form in the trace output. In the examples at the beginning of this section, you can see that `sched_migrate_task` is printed just like that.

The format defined with `TP_printk` can be seen from user space in the "format" subfolder in the event directory, as seen in figure 4.6. `TRACE_SYSTEM` is the trace system of the events, this will spawn the folder in the events directory and use it for these events: in figure 4.7 you can see the “sched” folder declared in the header, and its content in figure 4.4, where there is also `sched_migrate_task`.



```
root@turing-machine:/sys/kernel/debug/tracing/events# ls
asoc        exceptions  hda             iommu       mce      nmi             rcu     swiotlb   vmscan
block       ext4        hda_controller  irq         mei      oom             regmap  syscalls  vsyscall
cfg80211    fence       hda_intel       irq_vectors migrate  page_isolation  rpm     task      workqueue
cgroup      fib         header_event    jbd2        mmc      pagemap         sched   thermal   writeback
clk         fib6        header_page     kmem        module   power           scsi    timer     x86_fpu
compaction  filelock    huge_memory     kvm         mpx      printk          signal  tlb       xen
cpuhp       filemap     i2c             kvmmmu      msr      random          skb     udp       xhci-hcd
drm         ftrace      i915            libata      napi     ras             sock    v4l2
enable      gpio        intel-sst       mac80211    net      raw_syscalls    spi     vb2
```

Figure 4.7: All of the systems in which events are subdivided

4.4.1 Kernel module to test tracepoints

Let’s create our own kernel module, which will do something that we can trace from outside the kernel.

```c
#include <linux/module.h>
#include <linux/kthread.h>
#define CREATE_TRACE_POINTS
#include "my_module_trace.h"

static int do_stuff(void *arg){
        struct task_struct * p = current;
        u64 count = 0;
        printk(KERN_INFO "Current process is %s with pid %d\n", current->comm,
         ↪   current->pid);
        while (!kthread_should_stop()){
                set_current_state(TASK_INTERRUPTIBLE);
                schedule_timeout(HZ);
                printk("hi! %llu\n", count);
                count++;
                trace_my_event(p, jiffies); //Tracepoint
                p = next_task(p);
        }
        return 0;
}

static struct task_struct *my_tsk;
static int __init init_func(void){
        u32 i = 0, j = 0;
        struct task_struct *p, *t;
        printk(KERN_INFO "Hello world\n");
        for_each_process(p){
                i++;
                for_each_thread(p, t){
                        j++;
                }
        }
        printk(KERN_INFO "There are %d processes\n", i);
        printk(KERN_INFO "There are %d threads\n", j);
        //printk(KERN_INFO "Average threads per process: %s\n", division_hack(j, i));
        printk(KERN_INFO "Current process is %s with pid %d\n", current->comm, current->pid);

        my_tsk = kthread_run(do_stuff, NULL, "my-kthread");
        if (IS_ERR(my_tsk))
                return -1;
        return 0;
}

static void __exit exit_func(void){
        kthread_stop(my_tsk);
        printk(KERN_INFO "Goodbye world\n");
}

module_init(init_func);
module_exit(exit_func);
MODULE_LICENSE("GPL");
```

This is the trace header `my_module_trace.h` included in the module:

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM my_system
#if !defined(_MY_MODULE_TRACE_H) || defined(TRACE_HEADER_MULTI_READ) // Special guard
#define _MY_MODULE_TRACE_H
#include <linux/tracepoint.h> // TRACE_EVENT() is defined here
TRACE_EVENT(my_event,
        TP_PROTO(struct task_struct * t, unsigned long ticks),
        TP_ARGS(t, ticks),
        TP_STRUCT__entry(
                __array(char, name, TASK_COMM_LEN)
                __field(pid_t, pid)
                __field(u64, vruntime)
                __field(unsigned long, ticks)
        ),
        TP_fast_assign(
                memcpy(__entry->name, t->comm, TASK_COMM_LEN);
                __entry->pid        = t->pid;
                __entry->vruntime = t->se.vruntime;
                __entry->ticks = ticks;
        ),
        TP_printk("name=%s pid=%d vruntime=%lli ticks=%li", __entry->name,
        __entry->pid, __entry->vruntime, __entry->ticks)
);
#endif /* _MY_MODULE_TRACE_H */
/* This part must be outside protection */
#undef TRACE_INCLUDE_PATH
#define TRACE_INCLUDE_PATH .
#define TRACE_INCLUDE_FILE my_module_trace
#include <trace/define_trace.h>
```

We can compile by linking the headers of the currently running kernel on the system: these can be found within user space in `/lib/modules/your-kernel-version/build/include`.

We then insert the module with `sudo insmod my_module.ko`, then we print the kernel log with `sudo dmesg`: here we can see our `printk()` output.

```
[  410.661000] Hello world
[  410.661062] There are 160 processes
[  410.661065] There are 437 threads
[  410.661067] Current process is insmod with pid 5637
[  410.661111] Current process is my-kthread with pid 5638
[  411.683260] hi! 0
[  412.707226] hi! 1
[  413.731156] hi! 2
[  414.755236] hi! 3
# ...
```

We essentially used kernel APIs defined in the included headers. `for_each_process()`, `for_each_thread()` and `current` are all macros which are also used in the scheduler code. `init_func()` is the initialization function executed upon module insertion, so when we first print the currently executing process we read `insmod`. We then spawn a kernel thread which goes in a sleep state and wakes up after `HZ` ticks (one second); it then prints hi and throws an event before going into sleep again. We remove the module with `sudo rmmod my_module`, so `exit_func()` is executed, the kthread is terminated and the module removed.

We said earlier that bugs in the kernel can lock the system or in the worst case corrupt your data: this is why kernel modules are usually tested in a virtual machine, and the same goes for core kernel code. For example, in our module lines 11 and 12 are important: if we comment them out, the system completely freezes after a couple of seconds upon module insertion, and then crashes. What happens is that the thread hogs the CPU, starving all other processes: it can do this without being preempted because it's a kernel thread. While kernel threads are scheduled like normal tasks, it's also true that if they don't yield the CPU voluntarily, then they can continue to execute almost indefinitely. This is due to their priority being higher than most processes running on the system. While hogging the CPU, messages like this can be read in the kernel log:

```
[ 2795.881548] NMI watchdog: BUG: soft lockup - CPU#3 stuck for 22s! [my-kthread:2921]
```

Because of `TRACE_EVENT()`, we now have a folder for our trace system. We can trace our event by simply doing `echo 1 > trace/events/my_system/enable` in `debugfs`. The following is the trace output:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 191/191   #P:4
#
#                       _-----=> irqs-off
#                      / _----=> need-resched
#                     | / _---=> hardirq/softirq
#                     || / _--=> preempt-depth
#                     ||| /     delay
#   TASK - PID   CPU#  ||||    TIMESTAMP  FUNCTION
#      |   |        |   ||||       |         |
my-kthread-5638  [001] ....   411.683267: my_event: name=my-kthread pid=5638 vruntime=52137176502 ticks=4294995168
my-kthread-5638  [001] ....   412.707241: my_event: name=swapper/0 pid=0 vruntime=0 ticks=4294995424
my-kthread-5638  [001] ....   413.731163: my_event: name=systemd pid=1 vruntime=50883640967 ticks=4294995680
my-kthread-5638  [001] ....   414.755245: my_event: name=kthreadd pid=2 vruntime=52053628354 ticks=4294995936
my-kthread-5638  [002] ....   415.779197: my_event: name=ksoftirqd/0 pid=3 vruntime=53378623957 ticks=4294996192
my-kthread-5638  [002] ....   416.803364: my_event: name=kworker/0:0H pid=5 vruntime=722463814 ticks=4294996448
my-kthread-5638  [003] ....   417.831200: my_event: name=kworker/u8:0 pid=6 vruntime=44397385655 ticks=4294996705
my-kthread-5638  [003] ....   418.851389: my_event: name=rcu_sched pid=7 vruntime=52699580435 ticks=4294996960
my-kthread-5638  [003] ....   419.875192: my_event: name=rcu_bh pid=8 vruntime=62278851 ticks=4294997216
my-kthread-5638  [001] ....   420.903141: my_event: name=migration/0 pid=9 vruntime=0 ticks=4294997473
my-kthread-5638  [001] ....   421.923231: my_event: name=lru-add-drain pid=10 vruntime=67280643 ticks=4294997728
my-kthread-5638  [001] ....   422.947183: my_event: name=watchdog/0 pid=11 vruntime=-5995840 ticks=4294997984
my-kthread-5638  [001] ....   423.971192: my_event: name=cpuhp/0 pid=12 vruntime=1485464677 ticks=4294998240
my-kthread-5638  [001] ....   424.995245: my_event: name=cpuhp/1 pid=13 vruntime=1137933842 ticks=4294998496
my-kthread-5638  [002] ....   426.019248: my_event: name=watchdog/1 pid=14 vruntime=-2923868 ticks=4294998752
my-kthread-5638  [002] ....   427.043155: my_event: name=migration/1 pid=15 vruntime=0 ticks=4294999008
my-kthread-5638  [000] ....   428.067198: my_event: name=ksoftirqd/1 pid=16 vruntime=53693736956
# ... more entries ...
my-kthread-5638  [003] ....   552.996462: my_event: name=su pid=1598 vruntime=5158411963 ticks=4295030496
my-kthread-5638  [003] ....   554.020544: my_event: name=bash pid=1619 vruntime=66899459134 ticks=4295030752
my-kthread-5638  [003] ....   555.044561: my_event: name=firefox-esr pid=1684 vruntime=68424773099 ticks=4295031008
my-kthread-5638  [003] ....   556.068429: my_event: name=Web Content pid=1741 vruntime=68730788669 ticks=4295031264
my-kthread-5638  [003] ....   557.092489: my_event: name=nemo pid=1837 vruntime=66762475178 ticks=4295031520
my-kthread-5638  [003] ....   558.116460: my_event: name=gvfsd-metadata pid=1852 vruntime=7178686451 ticks=4295031776
my-kthread-5638  [003] ....   559.140545: my_event: name=dconf-service pid=1860 vruntime=7166782392 ticks=4295032032
my-kthread-5638  [003] ....   560.164474: my_event: name=Telegram pid=1912 vruntime=69024257689 ticks=4295032288
my-kthread-5638  [003] ....   561.188485: my_event: name=avahi-autoipd pid=2017 vruntime=56548711450 ticks=4295032544
my-kthread-5638  [003] ....   562.212523: my_event: name=avahi-autoipd pid=2018 vruntime=11116028299 ticks=4295032800
my-kthread-5638  [003] ....   563.236617: my_event: name=dhclient pid=2115 vruntime=66126515993 ticks=4295033056
```

It iterates through the process list, each second printing information about a process. It first traces the kernel threads (low pid), and eventually finds the user processes.

4.4.2 Overview of `TRACE_EVENT()` infrastructure

The macro works by expanding many other sub-macros. What it needs to do is to generate the structs and then generate the code of the tracepoint function, which basically writes the event in the trace ring buffer. To accomplish that, the marco uses a C pre-processor trick that lets you change its behavior while using the same input data. Here is a perfect example from yet another Steven Rostedt article:[11]

```c
#define DOGS { C(JACK_RUSSELL), C(BULL_TERRIER), C(ITALIAN_GREYHOUND) }
#undef C
#define C(a) ENUM_##a
enum dog_enums DOGS;
#undef C
#define C(a) #a
char *dog_strings[] = DOGS;
char *dog_to_string(enum dog_enums dog)
{
        return dog_strings[dog];
}
```

By redefining the sub-macro `C(a)` throughout the program, we change the behavior of `DOGS`: this way, we generate different code with the same data. `TRACE_EVENT()` does the same with its parameters, but performs this trick in a really weird way. Imagine that we had two different headers: `generate_code.h` and `change_behavior.h`.
`generate_code.h`:

1. Uses the macro, expanding it and generating code with a given input data
2. Does `#include change_behavior.h`

`change_behavior.h`:

1. Redefines the sub-macros, changing the original macro's behavior
2. Does `#include generate_code.h`, reincluding the header that just included it, which will generate new code
3. Repeats the process many times

This is exactly what we did with `DOGS`, it's just not as easy to see what happens. With this technique it's easier to change the macro usage because we have two separate files, but the code is way harder to read. Let's use our module to illustrate the mechanism in actual kernel code.
`my_module`:

```
#define CREATE_TRACE_POINTS
#include "my_module_trace.h"
```

`CREATE_TRACE_POINTS` is defined only if the kernel was compiled with the trace option activated.
`my_module_trace.h`:

```
#if !defined(_MY_MODULE_TRACE_H) || defined(TRACE_HEADER_MULTI_READ) //
 ↪   Special guard
#define _MY_MODULE_TRACE_H
#include <linux/tracepoint.h> //TRACE_EVENT() is defined here
TRACE_EVENT(my_event, ..., ..., ..., ..., ...)
// ... All other event declarations ...
#endif /* _MY_MODULE_TRACE_H */
/* This part must be outside protection */
#undef TRACE_INCLUDE_PATH
#define TRACE_INCLUDE_PATH .
#define TRACE_INCLUDE_FILE my_module_trace
#include <trace/define_trace.h>
```

This is the file that uses `TRACE_EVENT()` and the special guard is what makes us able to reinclude it multiple times. Usually, a guard is used to avoid mutiple inclusions. This is because if a function declaration is included twice, then we can't compile since there are two functions with the same name. Since there are no functions in trace headers, but just macros, then it's perfectly safe (and in this case needed) to do multiple inclusions. For reference, this would be the common use of a guard:

```c
#ifndef _MY_MODULE_TRACE_H
#define _MY_MODULE_TRACE_H
// ... Code ...
#endif _MY_MODULE_TRACE_H
```

At the end of the file, we define two more macros. `TRACE_INCLUDE_FILE` is the name of this file, which is needed later for the reinclusion. `TRACE_INCLUDE_PATH` changes the path of the trace headers, in this case it's `.` to indicate the current folder. This is needed for modules because they never use the default path used for core kernel code (`include/trace/events/`). This information is also needed for the reinclusion, which is performed in the header `define_trace.h` included at the end: this header corresponds to the "`change_behavior.h`" mentioned earlier.
`define_trace.h`:

```c
#ifdef CREATE_TRACE_POINTS // Defined to activate the tracepoints, used here
 ↪   as a guard
/* Prevent recursion */
#undef CREATE_TRACE_POINTS
// ... Redefine the sub-macros to change the behavior ...

#ifndef TRACE_INCLUDE_FILE
# define TRACE_INCLUDE_FILE TRACE_SYSTEM
# define UNDEF_TRACE_INCLUDE_FILE
#endif

#ifndef TRACE_INCLUDE_PATH
# define __TRACE_INCLUDE(system) <trace/events/system.h> //Used to reread
 ↪   system.h (e.g.: sched.h) trace header
# define UNDEF_TRACE_INCLUDE_PATH
#else
# define __TRACE_INCLUDE(system) __stringify(TRACE_INCLUDE_PATH/system.h)
#endif

# define TRACE_INCLUDE(system) __TRACE_INCLUDE(system)

/* Let the trace headers be reread */
#define TRACE_HEADER_MULTI_READ
// Reinclusion: includes the file that just included it
// e.g.: if the subsystem was sched, this just included <trace/events/sched.h>,
 ↪   in our example it's <./my_module_trace.h>
#include TRACE_INCLUDE(TRACE_INCLUDE_FILE)
#ifdef TRACEPOINTS_ENABLED
#include <trace/trace_events.h>
#endif
```

```
// ... Undefine every single macro and sub-macro ...

/* We may be processing more files */
#define CREATE_TRACE_POINTS
#endif /* CREATE_TRACE_POINTS */
```

Before the reinclusion, it's important to undefine `CREATE_TRACE_POINTS`, causing the guard to activate. This is because the reincluded file (`my_module_trace.h`) could include again this file (`define_trace.h`) at the end, causing an infinite loop in compilation. The next sequence generates the path of the file that included this file, taking the information from `TRACE_SYSTEM`, `TRACE_INCLUDE_FILE` and `TRACE_INCLUDE_PATH`. Finally, the file in the generated path is reincluded, causing `TRACE_EVENT()` to generate different code. At the end, `trace_events.h` is included: this header simply does this process again, many times. Its code has this general structure:

```
// Stage 1: change behavior
#include TRACE_INCLUDE(TRACE_INCLUDE_FILE) // Generate
// Stage 2: change behavior
#include TRACE_INCLUDE(TRACE_INCLUDE_FILE) // Generate
// Stage 3: change behavior
#include TRACE_INCLUDE(TRACE_INCLUDE_FILE) // Generate
// ...
```

Each stage simply redefines the sub-macros to generate code for a specific purpose. This is not exactly how the stages are structured, but just as an example, the stages could generate code like this:

- Stage 1: Generates the event struct with the proper fields
- Stage 2: Generates a struct with the offsets of each field in the event struct
- Stage 3: Creates the folder in the event directory of `debugfs`
- Stage 4: Generates a function that prints the raw event in the trace output format
- Stage 5: Generates a function to write the struct in the ring buffer

That's it! The code in this file is not hard to understand, it's just hard to read. There is also a big amount of code, and that's also why we are not going to go through it. For our purposes, that is not really interesting, but what really is interesting is how `TRACE_EVENT()` it's structured (and how hacky it is). For the curious, the path is `include/trace/events/trace_events.h`.

Chapter 5

Scheduler events

A simple script can give us the tracepoint location of every event that we are looking for.

```bash
#!/bin/bash
# find_sched_events.sh
events=$(ls /sys/kernel/debug/tracing/events/sched | grep
↪ "sched_" | sort)
cd /path/to/linux-4.20.13
for i in $events
do
        grep -rin "trace_$i" >> ../events_output
done
```

This gives us the following output:

```
fs/exec.c:1698:                 trace_sched_process_exec(current, old_pid, bprm);

kernel/exit.c:1503:         trace_sched_process_wait(wo->wo_pid);
kernel/exit.c:180:         trace_sched_process_free(tsk);
kernel/exit.c:866:         trace_sched_process_exit(tsk);

kernel/fork.c:2242:         trace_sched_process_fork(current, p);

kernel/hung_task.c:113:         trace_sched_process_hang(t);

kernel/kthread.c:543:         trace_sched_kthread_stop(k);
kernel/kthread.c:554:         trace_sched_kthread_stop_ret(ret);
kernel/kthread.c:554:         trace_sched_kthread_stop_ret(ret);

kernel/sched/core.c:1171:         trace_sched_migrate_task(p, new_cpu);
kernel/sched/core.c:1295:         trace_sched_swap_numa(cur, arg.src_cpu, p, arg.dst_cpu);
kernel/sched/core.c:1358:                 trace_sched_wait_task(p);
kernel/sched/core.c:1659:         trace_sched_wakeup(p);
```

```
kernel/sched/core.c:1798:                       trace_sched_wake_idle_without_ipi(cpu);
kernel/sched/core.c:1813:               trace_sched_wake_idle_without_ipi(cpu);
kernel/sched/core.c:473:               trace_sched_wake_idle_without_ipi(cpu);
kernel/sched/core.c:545:               trace_sched_wake_idle_without_ipi(cpu);
kernel/sched/core.c:1970:       trace_sched_waking(p);
kernel/sched/core.c:2100:       trace_sched_waking(p);
kernel/sched/core.c:2425:       trace_sched_wakeup_new(p);
kernel/sched/core.c:2425:       trace_sched_wakeup_new(p);
kernel/sched/core.c:3469:               trace_sched_switch(preempt, prev, next);
kernel/sched/core.c:3797:       trace_sched_pi_setprio(p, pi_task);
kernel/sched/core.c:5485:       trace_sched_move_numa(p, curr_cpu, target_cpu);

kernel/sched/fair.c:1836:                       trace_sched_stick_numa(p, env.src_cpu, env.best_cpu);
kernel/sched/fair.c:1844:               trace_sched_stick_numa(p, env.src_cpu, task_cpu(env.best_task));
kernel/sched/fair.c:833:               trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
kernel/sched/fair.c:886:               trace_sched_stat_wait(p, delta);
kernel/sched/fair.c:925:                       trace_sched_stat_sleep(tsk, delta);
kernel/sched/fair.c:944:                               trace_sched_stat_iowait(tsk, delta);
kernel/sched/fair.c:947:                       trace_sched_stat_blocked(tsk, delta);
```

There are 24 events and 31 tracepoints.

MP: dove c'è il tracepoint della exit ci sta un panic(). mostralo e citalo nel primo capitolo

5.1 Events in `core.c`

`trace_sched_migrate_task`

`trace_sched_swap_numa`

`trace_sched_wait_task`

`trace_sched_wakeup`

`trace_sched_wake_idle_without_ipi`

`trace_sched_waking`

`trace_sched_wakeup_new`

`trace_sched_switch`

`trace_sched_pi_setprio`

`trace_sched_move_numa`

5.2 Events in `fair.c`

`trace_sched_stick_numa`

`trace_sched_stat_runtime` Mentioned in Chapter 4.

`trace_sched_stat_wait`

`trace_sched_stat_sleep`

`trace_sched_stat_iowait`

`trace_sched_stat_blocked`

5.3 Events in other source files

`trace_sched_process_wait`

`trace_sched_process_free`

`trace_sched_process_exit`

`trace_sched_process_fork`

`trace_sched_process_hang`

`trace_sched_kthread_stop`

`trace_sched_kthread_stop_ret`

5.4 Tracepoints reminder (will remove)

```
ifs:

kernel/sched/fair.c:3817:        if (trace_sched_stat_wait_enabled()    ||
kernel/sched/fair.c:3818:                        trace_sched_stat_sleep_enabled()   ||
kernel/sched/fair.c:3819:                        trace_sched_stat_iowait_enabled()  ||
kernel/sched/fair.c:3820:                        trace_sched_stat_blocked_enabled() ||
kernel/sched/fair.c:3821:                        trace_sched_stat_runtime_enabled())  {
```

Events details + comments found in the code

```
/*
 * Tracepoint for calling kthread_stop, performed to end a kthread:
 */
sched_kthread_stop

/*
 * Tracepoint for the return value of the kthread stopping:
 */
sched_kthread_stop_ret

/*
 * Tracepoints for waking up a task:
 */

/*
 * Tracepoint called when waking a task; this tracepoint is guaranteed to be
 * called from the waking context.
 */
sched_waking:
  ./kernel/sched/core.c:1970:
    try_to_wake_up - wake up a thread
  ./kernel/sched/core.c:2100:
    try_to_wake_up_local - try to wake up a local task with rq lock held

/*
 * Tracepoint called when the task is actually woken; p->state == TASK_RUNNNG.
 * It it not always called from the waking context.
 */
sched_wakeup(./kernel/sched/core.c:1659):
  Mark the task runnable and perform wakeup-preemption.

/*
 * Tracepoint for waking up a new task:
 */
sched_wakeup_new(./kernel/sched/core.c:2425):
 wake_up_new_task - wake up a newly created task for the first time.
   This function will do some initial scheduler statistics housekeeping
   that must be done for every newly created context, then puts the task
   on the runqueue and wakes it.

/*
 * Tracepoint for task switches, performed by the scheduler:
 */
sched_switch(./kernel/sched/core.c:3469):
  context_switch - switch to the new MM and the new thread's register state.
```

```
/*
 * Tracepoint for a task being migrated:
 */
sched_migrate_task(./kernel/sched/core.c:1171):
  set_task_cpu(struct task_struct *p, unsigned int new_cpu)
  Migrate a task to new_cpu.

/*
 * Tracepoint for freeing a task:
 */
sched_process_free(./kernel/exit.c:180):
  TODO

/*
 * Tracepoint for a task exiting:
 */
sched_process_exit(./kernel/exit.c:866):
  process exit

/*
 * Tracepoint for waiting on task to unschedule:
 */
sched_wait_task

/*
 * Tracepoint for a waiting task:
 */
sched_process_wait(./kernel/exit.c:1503):

/*
 * Tracepoint for do_fork:
 */
sched_process_fork: (./kernel/fork.c:2242)
  _do_fork
    It copies the process, and if successful kick-starts
    it and waits for it to finish using the VM if required.

/*
 * Tracepoint for exec:
 */
sched_process_exec(./fs/exec.c:1698):
  exec_binprm(struct linux_binprm *bprm)
  load binaries using bprm(binary parameter)

/*
 * Tracepoint for accounting wait time (time the task is runnable
 * but not actually running due to scheduler contention).
 */
sched_stat_wait

/*
 * Tracepoint for accounting sleep time (time the task is not runnable,
 * including iowait, see below).
 */
sched_stat_sleep

/*
 * Tracepoint for accounting iowait time (time the task is not runnable
 * due to waiting on IO to complete).
 */
sched_stat_iowait

/*
 * Tracepoint for accounting blocked time (time the task is in uninterruptible).
 */
sched_stat_blocked

/*
 * Tracepoint for accounting runtime (time the task is executing
 * on a CPU).
 */
sched_stat_runtime(./kernel/sched/fair.c:829):
  Update the current task's runtime statistics.

/*
 * Tracepoint for showing priority inheritance modifying a tasks
 * priority.
 */
sched_pi_setprio

/*
 * Detect Hung Task
 */
sched_process_hang(./kernel/hung_task.c:113):
  Kernel thread for detecting tasks stuck in D state

/*
 * Tracks migration of tasks from one runqueue to another. Can be used to
 * detect if automatic NUMA balancing is bouncing between nodes
 */
sched_move_numa
```

```
//TODO
sched_stick_numa

//TODO
sched_swap_numa

/*
 * Tracepoint for waking a polling cpu without an IPI.
 */
sched_wake_idle_without_ipi
```

Bibliography

[1] AT&T Bell Labs promotional film, circa 1980 *Ken Thompson and Dennis Ritchie Explain UNIX (Bell Labs)*

[2] Marco Cesati and Daniel P. Bovet. *Understanding the Linux Kernel, Third Edition*. O'reilly, 2005.

[3] Linus Trovalds.
`www.realworldtech.com/forum/?threadid=65915&curpostid=65936`

[4] Con Kolivas. *RSDL completely fair starvation free interactive cpu scheduler*. Linux kernel mailing list, 2007. `lwn.net/Articles/224654/`

[5] Ingo Molnar. *Modular Scheduler Core and Completely Fair Scheduler*. Linux kernel mailing list, 2007. `www.lwn.net/Articles/230501/`

[6] *Documentation of the Linux Kernel*
`www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`

[7] *Documentation of the Linux Kernel*
`www.kernel.org/doc/Documentation/scheduler/sched-nice-design.txt`

[8] Gene M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. 1967. `http://www-inst.eecs.berkeley.edu/ n252/paper/Amdahl.pdf`

[9] Steven Rostedt. *Secrets of the Ftrace function tracer*. lwn.net, 2010. `lwn.net/Articles/370423/`

[10] Steven Rostedt. *Debugging the kernel using Ftrace*. lwn.net, 2009. `lwn.net/Articles/365835/`

[11] Steven Rostedt. *Using the TRACE_EVENT() macro*. lwn.net, 2010. `lwn.net/Articles/383362/`.
The article is a follow up to:
Part 1 – `lwn.net/Articles/379903/`.
Part 2 – `lwn.net/Articles/381064/`.