Università degli Studi di Torino

DIPARTIMENTO DI INFORMATICA
Corso di Laurea in Informatica

TESI DI LAUREA

Adding core scheduling support to rt-app and analyzing efficiency

Candidato:
Alessio Mana
Matricola 913380

Relatore:
Enrico Bini
University of Turin

Dario Faggioli
SUSE

Anno Accademico 2021/2022

Contents

List of Figures

List of Tables

Acknowledgments

Part of Chapter 1 is borrowed from the thesis by Marco Perronet [26], Stefano Chiavazza [15], and Stefano De Venuto [17]. Part of Chapter 2 is borrowed from the thesis by Marco Perronet [26], Stefano Chiavazza [15], Francesco Ciraolo [16] and Antonio Casoli [13]. Part of Chapter 3 is borrowed from the thesis by Marco Edoardo Santimaria [27]. Part of Chapter 4 is borrowed from the thesis by Marco Perronet [26] and Lorenzo Brescia [12]. Part of Chapter 5 is borrowed from the thesis by Lorenzo Brescia [12] and Giuseppe Eletto [19].

I'd like to thank my family, friends, supervisors and co-workers who supported and encouraged me during this part of my life.

Declaration of originality:

> *"Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata."*

Abstract

Core scheduling was recently introduced in Linux for security reasons, as it was discovered that processes running on core siblings can steal information from the other, as they share cache memory. The core scheduling in Linux, illustrated in detail in Chapter 3, allows forcing the scheduler to execute only processes authorized between them that share the core scheduling cookie on the core siblings. `rt-app`, an application for the generation of synthetic workload described in Chapter 6, did not yet provide this functionality, which is useful instead for carrying out tests on the limit cases of core scheduling.

Therefore, the core scheduling functionality in rt-app was implemented in this work, allowing tests to be carried out on all possible cases. Subsequently, we have carried out numerous tests to verify the security of the core scheduling even in cases that are very difficult to schedule and even with requests that are impossible to comply with. In any case, the scheduler managed to complete these tests, even though many quirks in the execution were highlighted due to the very specific requests. All these tests and their conclusions are covered in Chapter 7.

Chapter 1

Basics of Linux kernel

1.1 Introduction

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* and it 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 doing 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 1.2.4. 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"[30], 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 pre-defined 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 pre-defined 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.

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` (`strace` is explained better later4.2.1).

```
% 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
```

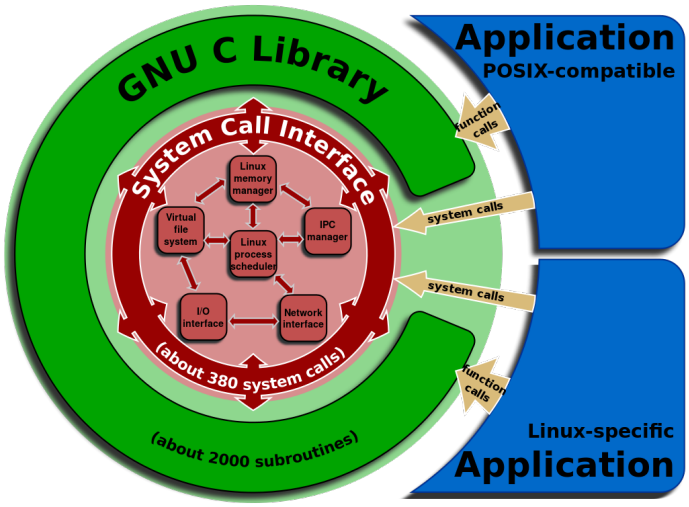


Figure 1.1: Kernel space (in red) and user space (in green and blue)

```
------ ----------- ----------- --------- --------- ----------------
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 Figure 1.1.

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

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 [14]. 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. The choice of checking assertion (and possibly going to kernel panic if 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 maintainers.

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 is concerns 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 machine1.2, 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*". 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 available 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

```
lorenzo@localhost:~/Scrivania/vm_Brescia> ulimit -s
8192
```

Figure 1.2: The maximum stack size on my machine

encounters along its path (Figure 1.3). This will make the process nonexistent for the kernel and cause a memory leak. 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.

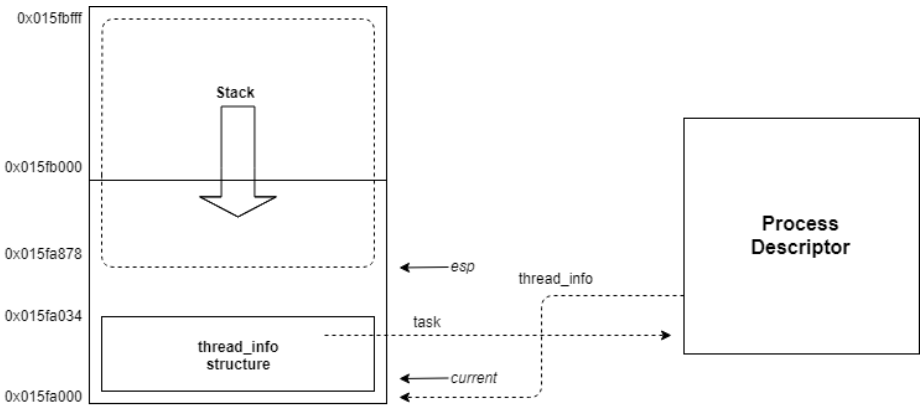


Figure 1.3: The kernel stack inside its small address space of two pages, it grows downward towards low memory.

It's important to note that there are special processes called *kernel threads* 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 1.4).

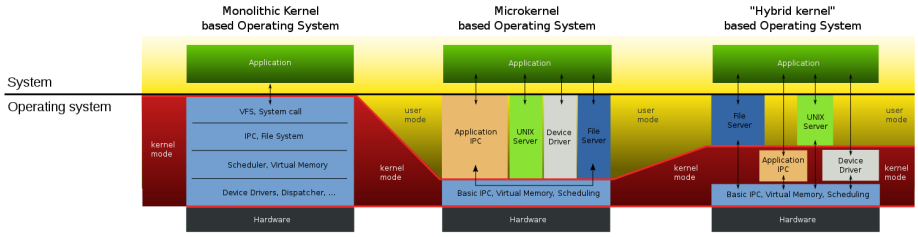


Figure 1.4: The most popular kernel designs and their differences

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. 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 environment, 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 architecture 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 through heavy use of IPC. More precisely, the task of moving in and out of the kernel to move data between 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 can make 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.

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 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 it 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"[31]. 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 1.5 is a zoom into the kernel mechanisms inside the red part of Figure 1.1. The names in the picture are structs, functions or source code files.

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

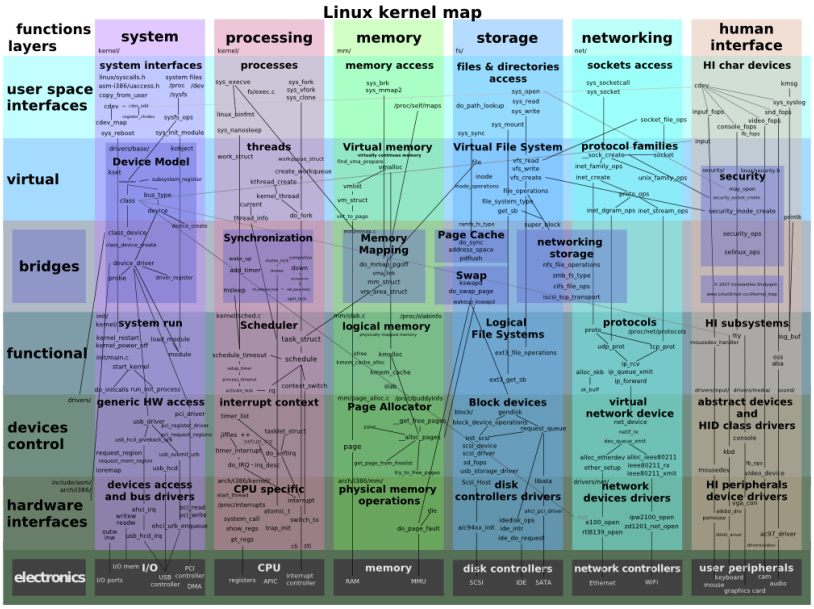


Figure 1.5: Kernel subsystems map[28]

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 only has 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 resembles 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 (in section 4.2.1,`getpid()` it's used for track a process).

You can already see how "process" and "thread" are being used interchange-

ably. As anticipated, this terminology can be confusing so let's clarify. 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 1.3.2, 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.

```
1  //stack size for cloned child
2  const int STACK_SIZE = 65536;
3  //start and end of stack buffer area
4  char *stack, *stackTop;
5  // ... define child startup function "do_something" ...
6  stack = malloc(STACK_SIZE);
7  stackTop = stack + STACK_SIZE; //stack grows downward
8
9  //spawns a new thread
10 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 parent, while the other flags clone filesystem information (such as working directory), open files and signal handlers. The flag `SIGCHLD` at line 12 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.

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.

```
1  // Code from ./include/linux/sched.h
2  struct task_struct {
3          /* -1 unrunnable, 0 runnable, >0 stopped: */
4          volatile long state;
5          void *stack;
6          /* Current CPU: */
7          unsigned int cpu;
8          // A boolean, "on_runqueue"
9          int on_rq;
10         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;
};
```

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 `volatile` 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 1.6 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.

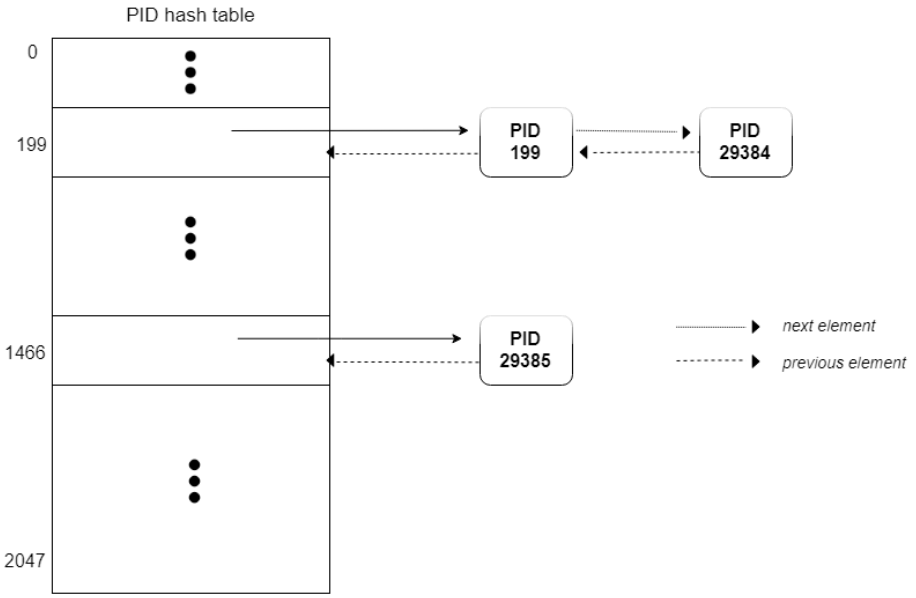


Figure 1.6: Pid hash table, pids 199 and 29384 are both hashed to 199

```
// Code from ./include/linux/pid.h
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
};
```

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 1.7). 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 1.7 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 list, and since it's circular there's really no head structure that points to the first element.

```
// Code from ./include/linux/pid.h
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
};
```

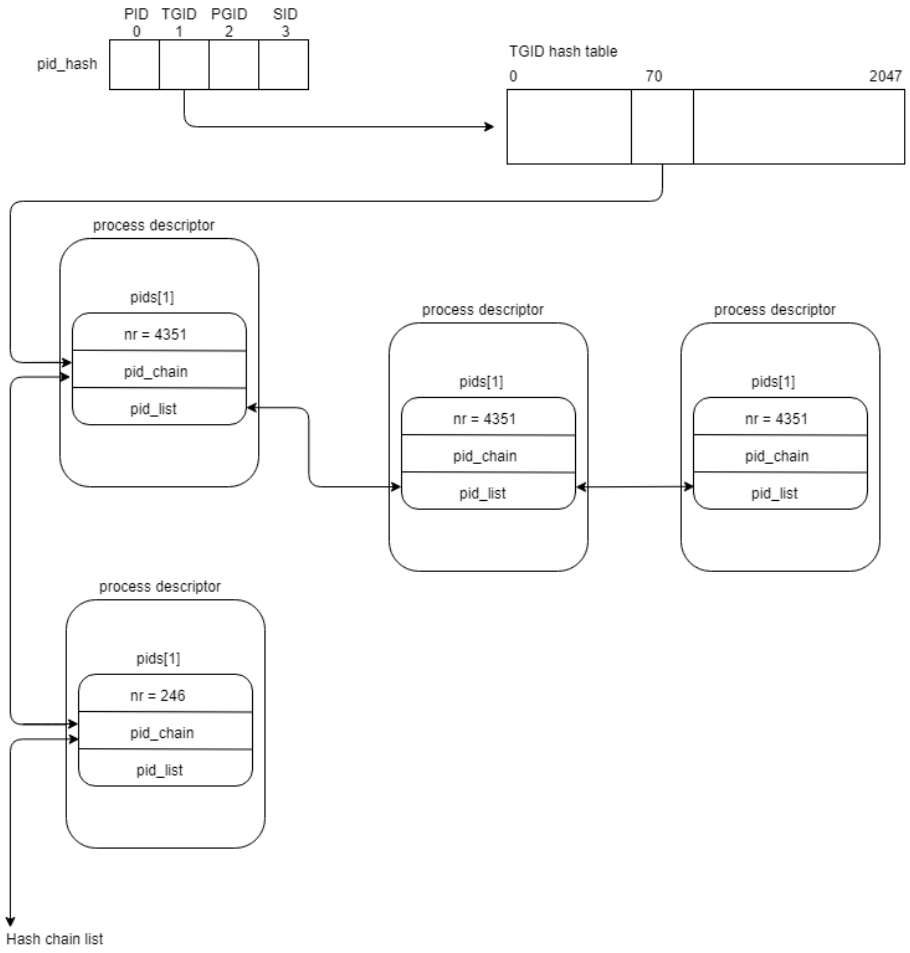


Figure 1.7: Hash table for the TGID pid type

The implementation of `struct pid` is slightly different from what was presented, with other nested structures and a different linkage to the hash table.

1.3.2 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 (see in Section 5.2 Figure5.5 a real example, on my machine, of this event). 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()` 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.

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 1.2.1 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 and 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;
};
```

```
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;
```

```
12      unsigned short        gsindex;
13  #endif
14      // ...
15      /* Floating point and extended processor state */
16      struct fpu      fpu;
17  };
```

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 is specific to x86 it can still change depending on whether the architecture is 32 or 64 bits. 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.

1.3.3 CPU Idle Management

Logical CPUs, simply referred to as "CPUs" in what follows, are regarded as **idle** by the Linux kernel when there are no tasks to run on them except for the special "idle" task.

Tasks can be in various states. In particular, they are *runnable* if there are no specific conditions preventing their code from being run by a CPU as long as there is a CPU available for that (for example, they are not waiting for any events to occur or similar). When a task becomes runnable, the CPU scheduler assigns it to one of the available CPUs to run and if there are no more runnable tasks assigned to it, the CPU will load the given task's context and run its code (from the instruction following the last one executed so far, possibly by another CPU).

The special "idle" task becomes runnable if there are no other runnable tasks assigned to the given CPU and the CPU is then regarded as idle. In Linux idle CPUs run the code of the "idle" task called the **idle loop**. That code may cause the processor to be put into one of its idle states, if they are supported, in order to save energy, but if the processor does not support any idle states, or there is not enough time to spend in an idle state before the next wakeup event, or there are strict latency constraints preventing any of the available idle states from being used, the CPU will simply execute more or less useless instructions in a loop until it is assigned a new task to run.

The idle loop code takes two major steps in every iteration of it. First, it calls into a code module referred to as the **governor** that belongs to the CPU idle time management subsystem called *CPUIdle* to select an idle state for the CPU to ask the hardware to enter. Second, it invokes another code module from the *CPUIdle* subsystem, called the **driver**, to actually ask the processor hardware to enter the idle state selected by the governor.

The role of the governor is to find an idle state most suitable for the conditions at hand. For this purpose, idle states that the hardware can be asked to enter by logical CPUs are represented in an abstract way independent of the platform or the processor architecture and organized in a one-dimensional (linear) array. That array has to be prepared and supplied by the *CPUIdle* driver matching the platform the kernel is running on at the initialization time. This allows *CPUIdle* governors to be independent of the underlying hardware and to work with any platforms that the Linux kernel can run on.

There are four *CPUIdle* governors available, `menu`, `TEO`, `ladder` and `haltpoll`. Which of them is used by default depends on the configuration of the kernel and in particular on whether or not the scheduler tick can be stopped by the idle loop. Available governors can be read from the available_governors file, and the governor can be changed at runtime. The name of the *CPUIdle* governor currently used by the kernel can be read from the `current_governor_ro` or `current_governor` file under `/sys/devices/system/cpu/cpuidle/` in sysfs. As an example, in the current PC:

```
root@user:~# cat /sys/devices/system/cpu/cpuidle/available_governors
ladder menu teo
```

```
root@user:~# cat /sys/devices/system/cpu/cpuidle/current_governor
menu
```

Which *CPUIdle* driver is used, on the other hand, usually depends on the platform the kernel is running on, but there are platforms with more than one matching driver. The name of the *CPUIdle* driver currently used by the kernel can be read from the `current\_driver` file under `/sys/devices/system/cpu/cpuidle/` in the same filesystem as before. As an example, in the current PC:

```
root@user:~# cat /sys/devices/system/cpu/cpuidle/current_driver
acpi_idle
```

For the CPU idle time management purposes all of the physical idle states supported by the processor have to be represented as a one-dimensional array of `struct cpuidle_state` objects each allowing an individual (logical) CPU to ask the processor hardware to enter an idle state of certain properties.

```
1  struct cpuidle_state {
2          char            name[CPUIDLE_NAME_LEN];
3          char            desc[CPUIDLE_DESC_LEN];
4
5          s64             exit_latency_ns;
6          s64             target_residency_ns;
7          unsigned int    flags;
8          unsigned int    exit_latency; /* in US */
9          int             power_usage; /* in mW */
10         unsigned int    target_residency; /* in US */
11
12         int (*enter)    (struct cpuidle_device *dev,
13                          struct cpuidle_driver *drv,
14                          int index);
15
16         int (*enter_dead) (struct cpuidle_device *dev, int index);
17
18         /*
19          * CPUs execute ->enter_s2idle with the local tick or entire
    ↪  timekeeping
20          * suspended, so it must not re-enable interrupts at any point (even
21          * temporarily) or attempt to change states of clock event devices.
22          *
23          * This callback may point to the same function as ->enter if all of
24          * the above requirements are met by it.
25          */
26         int (*enter_s2idle)(struct cpuidle_device *dev,
27                             struct cpuidle_driver *drv,
28                             int index);
29 };
```

The more useful fields to explore in order to continue are:

desc Description of the idle state.

latency Exit latency of the idle state in microseconds.

name Name of the idle state.

power Power drawn by hardware in this idle state in milliwatts.

residency Target residency of the idle state in microseconds.

In addition to the parameters above, the objects representing idle states each contain a pointer to the function to run in order to ask the hardware to enter that state.

Also, for each `struct cpuidle_state` object, there is a corresponding `struct cpuidle_state_usage` one containing usage statistics of the given idle state.

```
1  struct cpuidle_state_usage {
2          unsigned long long      disable;
3          unsigned long long      usage;
4          u64                     time_ns;
5          unsigned long long      above; /* Number of times it's been too deep
           ↪  */
6          unsigned long long      below; /* Number of times it's been too
           ↪  shallow */
7          unsigned long long      rejected; /* Number of times idle entry was
           ↪  rejected */
8  #ifdef CONFIG_SUSPEND
9          unsigned long long      s2idle_usage;
10         unsigned long long      s2idle_time; /* in US */
11 #endif
12 };
```

And again, the more useful fields:

above Total number of times this idle state had been asked for.

below Total number of times this idle state had been asked for.

disable Whether or not this idle state is disabled.

time Total time spent in this idle state by the given CPU in microseconds.

usage Total number of times the hardware has been asked by the given CPU to enter this idle state.

rejected Total number of times a request to enter this idle state on the given CPU was rejected.

All the information contained in these two structures is exposed by the kernel via *sysfs*. In fact, for each CPU in the system, there is a `/sys/devices/system/cpu/cpu<N>/cpuidle/` directory that contains a set of subdirectories, one for each idle state objects defined for the given CPU. The larger the number in its name, the deeper the (effective) idle state represented by it. Each of them contains a number of files (attributes) representing the properties seen previously.

1.4 Timekeeping mechanisms

Timekeeping is the process or activity of recording how long something takes. We need "instruments" to measure time. The Linux kernel has several abstractions to represent such devices:

Clocksource The purpose of the clock source is to provide a timeline for the system that tells you where you are in time. For example issuing the command 'date' on a Linux system will eventually read the clock source to determine exactly what time it is.

Typically the clock source is a monotonic, atomic counter which will provide n bits which count from 0 to $(2^n) - 1$ and then wraps around to 0 and start over. It will ideally **never** stop ticking as long as the system is running. It may stop during system suspend.

The clock source shall have as high resolution as possible, and the frequency shall be as stable and correct as possible as compared to a real-world wall clock. It should not move unpredictably back and forth in time or miss a few cycles here and there.

Clockevent device Clock events are the conceptual reverse of clock sources: they take a desired time specification value and calculate the values to poke into hardware timer registers.

Clock events are orthogonal to clock sources. The same hardware and register range may be used for the clock event, but it is essentially a different thing.

sched_clock() This function is similar to clocksource, but should be "cheaper" to read (meaning that one can get its value fast), as sched_clock() is used for task-scheduling purposes and scheduling happens often. That means sacrifice accuracy and other characteristics for speed.

In order to generate a clocksource many hardware devices are available, like Programmable Interrupt Timer (PIT), RealTime Clock (RTC), Advanced Programmable Interrupt Controller (APIC) and High Precision Event Timer (HPET), but the detailed specification of them is beyond the current scope of indroduction and the thesis itself. The only one that will be discussed is the **Time-Stamp Counter (TSC) 1.4.1**.

A full list of clocksources available on a machine along with the one used can be retrieved by interacting with the related *sysfs*, by respectively reading the `available_clocksource` and `current_clocksource` files in the `/sys/devices/system/clocksource/clocksource0` folder. As an example, on the current machine:

```
1 root@user:~# cat /sys/d⌋
  ↪ evices/system/clocksource/clocksource0/available_clocksource
2 tsc hpet acpi_pm
```

```
3 root@user:~# cat /sys/d┘
  ↪ evices/system/clocksource/clocksource0/current_clocksource
4 tsc
```

1.4.1 Time-Stamp Counter (TSC)

The Time-Stamp Counter (TSC) is used to count processor-clock cycles. The TSC is cleared to 0 after a processor reset and it will be incremented at a rate corresponding to the baseline frequency of the processor (which may differ from actual processor frequency in low power modes of operation). Each time the TSC is read, it returns a monotonically-larger value than the previous value read from the TSC. When the TSC contains all ones, it wraps to zero. The TSC in a 1-GHz processor counts for almost 600 years before it wraps.

The counter's architecture includes the following components:

- **TSC flag** A feature bit that indicates the availability of the time-stamp counter.
- **IA32_TIME_STAMP_COUNTER MSR** The MSR used as the counter.
- **TSD flag** A control register flag used to enable or disable the time-stamp counter.

The TSC register be read using one of the special read time-stamp counter instructions, RDTSC (Read Time-Stamp Counter) or RDTSCP (Read Time-Stamp Counter and Processor ID).

The former reads the Time-Stamp Counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound, the latter returns atomically not just the TSC, but also an indicator which corresponds to the processor number. This can be used to index into an array of TSC variables to determine offset information in SMP systems where TSCs are not synchronized.

Of course, the regular RDMSR and WRMSR instruction can be used, treating the Time-Stamp Counter as an ordinary MSR.

When used in virtual enviroments, an important mechanism can be used, the Timestamp-counter offsetting (TSC offsetting). It is a feature that allows VMM software to specify a value (the TSC offset) that is added to the TSC when it is read by guest software. A VMM can use this feature to provide guest software with the illusion that it is operating at a time later or earlier than that represented by the current TSC value.

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 Elements of scheduling decisions

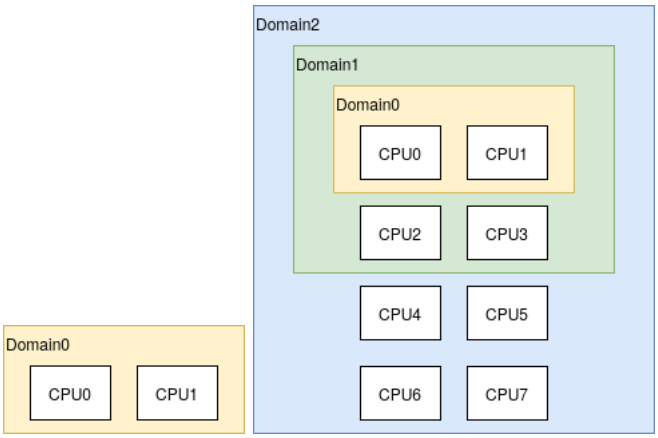
2.1.1 Domains

The domain struct is a key component in the linux scheduler. Each domain keeps two main elements: a spans array and a sched_group circular list. Above each CPU is built a hierarchy of domains, which cluster increasingly more cores and manages the related balance.

Spans Each domain clusters a cores' quantity, growing according to its level in the hierarchy. A domain's span needs to include his child's span and can extends it. The balance doesn't take place between all the CPUs' ready-queue but among the sched_groups.

Sched groups Sched_groups of a domain are partitions of its span, so each core in the span is in one and only one group. The balance is performed among them, involving the more busy and the less one of each group and balancing their ready-queue.

Hierarchy At the base level, the domains' hierarchy is based on a couple of close cores at least, meaning that there can't be a domain with a single core (2.1a). Above each domain is placed another one which has more cores, until a domain containing all CPUs in the system is made (2.1b).



(a) First step (b) Complete

Figure 2.1: Scheduler domains' hierarchy

Something often omitted is that the hierarchy shown in 2.1b can be either the CPU0 or CPU1 one, but not both: each core owns a private domains hierarchy. This implies that even if CPU0 and CPU1 are in the same domain, named Domain0 in the picture, they hold two different actual structs that represent it. This behavior helps to finer tune the domains balancing with the flags described in Section 2.3.2.

On the other hand, the groups involved in the domains structs are the same across all the system. Let's take as an example the same hierarchy shown in fig 2.1b and the groups shown in 2.2. Group0 and Group1, references of CPU0's first domain, are the same in CPU1's counterpart and a similar situation happens with the second domain of the first 4 cores: they all share the same Group2 and Group3 pointers. In this way all the domains are coherent and are able to balance processes load among the same groups.

2.1.2 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 quanta. 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:

- `dl_sched_class` – `kernel/sched/deadline.c`

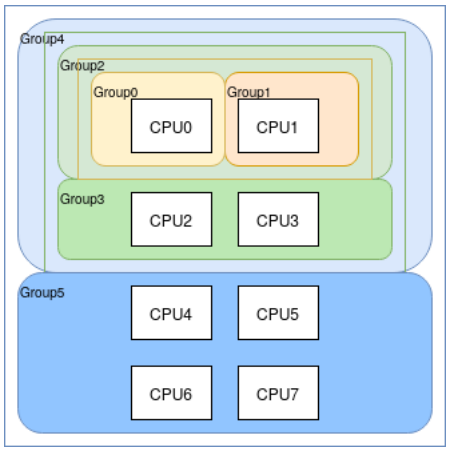


Figure 2.2: A complete domains' hierarchy

- `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 preemptive. 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.

2.1.3 Priorities

Each task has a priority associated with it. The first priority levels corresponds to real-time tasks, which are scheduled with the `SCHED_FIFO` or `SCHED_RR` pol-

icy. 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 reason why it's called "nice" is that it represents how nice the process is being to all the others: being very nice (positive nice) means having a low priority.

The behavior of different nice values depends on the version of the scheduler. This has been a problem for Linux for years, because nice's behaviour used to advantage servers over desktop[2].

2.2 Multiprocessing

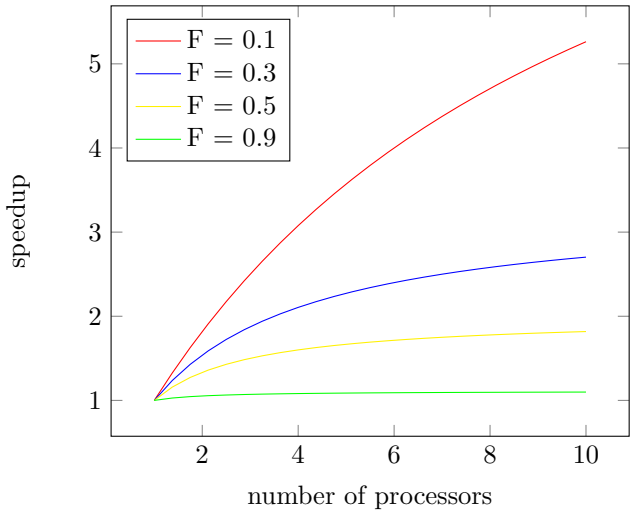
As mentioned in Chapter 1, nowadays most systems have more than one processor. Over the years, the frequencies of processors have been increased in order to achieve better performances, but there are physical limitations to this. Once we hit the limit in CPU frequency, the best way to improve the performance 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 [11] predicts the maximum theoretical improvement of multiprocessing:

$$speedup = \frac{1}{F + \frac{1-F}{N}} \tag{2.1}$$

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.

2.2.1 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 core is the sum of the loads of its tasks. Since the CPU utilization of a task can vary, its load is constantly updated.

The load then is used as a reference to take important decisions in order to try using the computational resources in the most effective way. This decisions are taken by the load-balancer, in practice represented by the scheduler which must decide every time on which cpu the task should be executed on. Since the

definition of a balanced system may vary according to the needs, these decisions must also be tuned accordingly, and to do so, the load balancer may slightly adjust the load value for the evaluation phase. This will be explained more in depth in the section about the attribute responsible for this evaluation 2.3.1.

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.

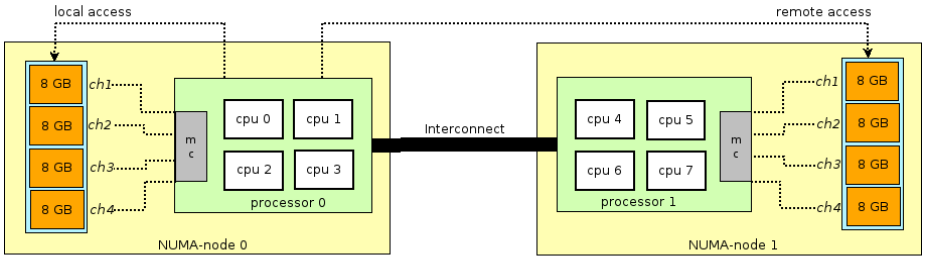


Figure 2.3: Local and foreign memory in a NUMA architecture

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.

2.3 Scheduling domains

As introduced in Section 2.1.1, the scheduling domains play a crucial role when trying to balance the different loads among the CPUs. They allow a finer control on scheduling decisions over the whole topology, making possible to adopt different strategies or policies for different groups of cores.

2.3.1 Load-balancing threshold

There is an attribute inside every scheduling domain struct called `imbalance_pct`, which is a number representing loosely speaking "how imbalanced (always in terms of load) a domain is allowed to be". To be more precise this value is used as a threshold for many cases, but primarily to decide whether a balancing operation is needed or not. Let's see in practice how changes in this value affect the domain balancing from the fair `sched_class` by looking at the code of two functions (developers comments have been left for more clarity):

```
1  /*
2  * group_has_capacity returns true if the group has spare capacity that could
3  * be used by some tasks.
4  * We consider that a group has spare capacity if the  * number of task is
5  * smaller than the number of CPUs or if the utilization is lower than the
6  * available capacity for CFS tasks.
7  * For the latter, we use a threshold to stabilize the state, to take into
8  * account the variance of the tasks' load and to return true if the available
9  * capacity is meaningful for the load balancer.
10 * As an example, an available capacity of 1% can appear but it doesn't make
11 * any benefit for the load balance.
12 */
13 static inline bool
14 group_has_capacity(struct lb_env *env, struct sg_lb_stats *sgs)
15 {
16         if (sgs->sum_nr_running < sgs->group_weight)
17         return true;
18
19         if ((sgs->group_capacity * 100) >
20         (sgs->group_util * env->sd->imbalance_pct))
21         return true;
22
23         return false;
24 }
25
26 /*
27 *  group_is_overloaded returns true if the group has more tasks than it can
28 *  handle.
29 *  group_is_overloaded is not equals to !group_has_capacity because a group
30 *  with the exact right number of tasks, has no more spare capacity but is not
31 *  overloaded so both group_has_capacity and group_is_overloaded return
32 *  false.
33 */
34 static inline bool
35 group_is_overloaded(struct lb_env *env, struct sg_lb_stats *sgs)
36 {
37         if (sgs->sum_nr_running <= sgs->group_weight)
```

```
38          return false;
39
40          if ((sgs->group_capacity * 100) <
41          (sgs->group_util * env->sd->imbalance_pct))
42          return true;
43
44          return false;
45  }
```

Both functions describe in some way the capacity of a group, which is done by checking if the *effective* group capacity exceeds the `group_util`, an attribute that can be viewed as the current load of the group, multiplied by the `imbalance_pct`; then they return their respective information to the load balancer (in the fair `sched_class` represented by `slect_rq_fair()`) which will decide how to best perform the balancing. We can see that the `imbalance_pct` can be used to leverage the current load, ultimately directly affecting what to make the function return. If, for instance, we set this value to 0, the function `group_has_capacity()` will always (except when the capacity is 0 itself) return true, therefore indicating that the group allows can receive more tasks resulting in a much more frequent activation of the balancer, since the system is told that there's a group which always has capacity (which can be expressed as underloaded). If instead, we were to increase the value, a higher value of capacity will be needed in order to return true. The same reasoning can be applied in the `group_is_overloaded()` function. To summarize, the function realized by the attribute `imabalance_pct` is that of a threshold (which could be seen as percentage) indicating how tolerant to be, relative to the load of a group/cpu, before activating load balancing mechanism. Higher values are more lenient and activate the balancer less often, while lower values are more strict and result in more frequent activations. The default value is set to 125, but by activating some of the flags described in the next sections, this value changes in order to adjust for different environments.

Over the course of the versions, this threshold has been used for more than its simple and original purpose of scheduling activator, and took over some responsibility for power management, specifically in arm specializations of the kernel.

2.3.2 Flags

To accomplish this task, they keep some statistics of the processes being run on the domain, and the options on how to behave when certain events occur. This options are encoded through the use of flags, which are illustrated in the following piece of code from *topology.c*:

```
1  #define SD_LOAD_BALANCE                 0x0001        /* Do load balancing on
   ↪  this domain. */
2  #define SD_BALANCE_NEWIDLE        0x0002        /* Balance when about to become
   ↪  idle */
3  #define SD_BALANCE_EXEC                 0x0004        /* Balance on exec */
4  #define SD_BALANCE_FORK                 0x0008        /* Balance on fork, clone
   ↪  */
5  #define SD_BALANCE_WAKE                 0x0010  /* Balance on wakeup */
6  #define SD_WAKE_AFFINE                0x0020        /* Wake task to waking CPU
   ↪  */
7  #define SD_ASYM_CPUCAPACITY        0x0040  /* Domain members have different CPU
   ↪  capacities */
8  #define SD_SHARE_CPUCAPACITY         0x0080        /* Domain members share CPU
   ↪  capacity */
9  #define SD_SHARE_POWERDOMAIN         0x0100        /* Domain members share power
   ↪  domain */
10 #define SD_SHARE_PKG_RESOURCES         0x0200        /* Domain members share CPU
   ↪  pkg resources */
11 #define SD_SERIALIZE                 0x0400        /* Only a single load
   ↪  balancing instance */
12 #define SD_ASYM_PACKING                 0x0800  /* Place busy groups earlier in
   ↪  the domain */
13 #define SD_PREFER_SIBLING        0x1000        /* Prefer to place tasks in a
   ↪  sibling domain */
14 #define SD_OVERLAP                0x2000        /* sched_domains of this level
   ↪  overlap */
15 #define SD_NUMA                         0x4000        /* cross-node balancing
   ↪  */
```

This flags are set during the initialization of a SD, according to the underlying topology, or, if its definition is missing or some errors occurred, following default values. They each have their own function, but we can divide them into two general categories: those flags which enable or disable some sort of function, usually the load balancing between domains, and those which change values in order to achieve a different behavior of the system, like for instance the threshold used for indicating the load of a cpu and therefore whether a balancing is due or not. This options allow to fine tune the load balancing mechanism which can be useful when coupled with other sections of the kernel such as the power management aspect. In the following paragraphs will be provided a description for each one of these flags.

`SD_LOAD_BALANCE` This can be considered one of the most important flags, mainly used to inform if the SD requires an attempt on balancing the load. It belongs to the enablers category, acting as a switch for load balancing operations. It appears in the following instances inside fair.c:

1. `select_task_rq_fair()`: this function selects in which rq the examined task get moved. In this instance the flag `SD_LOAD_BALANCE`, if set, informs the scheduler to be considered during this choice.

2. `rebalance_domains()`: this function checks each scheduling domain to see if it is due to be balanced, and `SD_LOAD_BALANCE`, as in the previous case

if set, informs the scheduler to be considered during this analysis. This function is invoked from the function `run_rebalance_domains()`, which is triggered when needed from the scheduler tick with a SOFTIRQ signal.

3. `newidle_balance()`: this function is called by schedule() when the current cpu is about to become idle. Its goal is to pull tasks from other CPUs in an attempt to prevent useless idle time and to have a good distribution of the tasks. As previously `SD_LOAD_BALANCE`, if set, informs the scheduler to to take the SD in consideration, when checking every domain, to pull task from.

4. `active_load_balance_cpu_stop()`: this function pushes running tasks off the busiest CPU onto idle CPUs. As before `SD_LOAD_BALANCE`, when set, informs the scheduler to take the SD in exam into account.

`SD_BALANCE_NEWIDLE` The purpose of this flag is to inform the fair scheduler to consider the SD in exam during the previously mentioned function `newidle_balance()`. It is an enabler and is used as a more specific flag for this function.

`SD_BALANCE_EXEC` This flag, together with `SD_BALANCE_FORK` and `SD_BALANCE_WAKE`, can be considered as enablers for their specific moments. Specifically, `SD_BALANCE_EXEC` defines whether to perform the load balancing operation at execve time. This moment is considered a good load balancing opportunity since the process is newly born, meaning that the task has the smallest effective memory and cache footprint. In practice this flag is passed as parameter to the function `select_task_rq()`, communicating that the context is that of execve.

`SD_BALANCE_FORK` This flag, together with `SD_BALANCE_EXEC` and `SD_BALANCE_WAKE`, can be considered as enablers for their specific moments. Specifically, `SD_BALANCE_FORK` defines whether to perform the load balancing operation at fork time. As for exec, this moment is considered a good load balancing opportunity since the process is newly born, meaning that the task has the smallest effective memory and cache footprint. In practice this flag is passed as parameter to the function `select_task_rq()`, communicating that the context is that of fork. Moreover, in the fair sched_class is used to make adjustments in the functions `find_idlest_cpu()` and `find_idlest_group()`, providing the information that the task has been forked. In the rt sched this flag skips the evaluation returning the task_cpu if not set.

`SD_BALANCE_WAKE` This flag, together with `SD_BALANCE_EXEC` and `SD_BALANCE_FORK`, can be considered as enablers for their specific moments. Specifically, `SD_BALANCE_WAKE` defines whether to perform the load balancing operation at wake time(which is the moment in which the task_state becomes RUNNING). As the previous two, it is passed as argument for the

`select_task_rq()` function from the `try_to_wake_up()` function, informing that it's a task being woken up. Each one of the different sched classes has its own approach. Rt and deadline will try to balance if this flag is set, otherwise they would completely skip the evaluation returning the task_cpu.

`SD_WAKE_AFFINE` This flag allows the check for affinity. This means that if the cpu is in the same domain as the one in which the task was previously executed, then it can be considered a valid wake affine target. It is used as an alternative for the idlest cpu. Instead of selecting the idlest cpu, the scheduler looks for domains with the `SD_WAKE_AFFINE` flag set.

`SD_ASYM_CPUCAPACITY` This flag is used to inform that the scheduling domains have different cpu capacity. It is set when the SD are built, based on the informations the topology provided. As of this version the only sched class concerned with this flag is the fair. Based on whether it is set or not, adjust the calculations for the statistics of the scheduling groups, and for picking the busiest queue and SD.

Topology descriptive flags For the following flags to come, let's take a look at the comment of the developers about them:

```
/*
 * SD_flags allowed in topology descriptions.
 *
 * These flags are purely descriptive of the topology and do not prescribe
 * behaviour. Behaviour is artificial and mapped in the below sd_init()
 * function:
 *
 *   SD_SHARE_CPUCAPACITY   - describes SMT topologies
 *   SD_SHARE_PKG_RESOURCES - describes shared caches
 *   SD_NUMA                - describes NUMA topologies
 *   SD_SHARE_POWERDOMAIN   - describes shared power domain
 *
 * Odd one out, which beside describing the topology has a quirk also
 * prescribes the desired behaviour that goes along with it:
 *
 *   SD_ASYM_PACKING        - describes SMT quirks
 */
```

`SD_SHARE_CPUCAPACITY` This flag represents the ability for the CPUs in the domain to share the computational power. It is set by the function powerpc_smt_flags in smp.c.

If this flag is set, makes the value `imbalance_pct` change from the default 125 to 110. This because since there are shared capacity it's easier to perform the balancing between SDs, therefore, we can lower the `imbalance_pct` threshold.

`SD_SHARE_POWERDOMAIN` This flag has no use, it is purely descriptive.

SD_SHARE_RESOURCES Similar to SD_SHARE_CPUCAPACITY in scope and purpose. This flag represents CPU in the domain that are able to share their cache. It is set by the function powerpc_smt_flags in smp.c. If this flag is set, makes the value imbalance_pct change from the default 125 to 117.

Same reasoning as the one for SD_SHARE_CPUCAPACITY applies about the lowering of the threshold, except is a bit higher since the only thing shared is the cache and not the capacity directly.

SD_SHARE_NUMA This flag is used to correctly calculate the idlest group taking in consideration the distance of nodes.As most of the other flags, its influence is relevant only in the fair sched class It is used in update_sd_lb_stats() to properly update the sg_lb_stats structs (structs keeping important informations for the load balancer on the SDs), but it's essential function is to override the normal way the balancer work, by canceling the effects that a NUMA topology would have on it. In the first case, this flags sets the value returned by the function migrate_degrades_locality(), whose purpose is to return whether the migration of a task is affected by locality, to -1, telling the balancer that the task isn't affected by locality, allowing a more artificial control dictated by the topology as the next one. In the other case in fact, this flags artificially adds imbalance to the one calculated to prevent unnecessary migrations to remote nodes, that may result more favorable to the load balancer than what they really are, since the real distance could render the balancing operation more harmful than beneficial. This is done inside the find_idlest_group() function.

SD_SERIALIZE This flag is used to coordinate and synchronize the rebalancing operation of domains (in this version seems to be used only by the fair sched class). If set, the domain tries to acquire the spinlock and if successful calls the function load_balance() while also setting the time for the next rebalance. In practice this flag won't allow multiple instances of load balancing active at the same time from different domains.

SD_ASYM_PACKING This flag is set by the function powerpc_smt_flags() in smp.c, if the cpu has the feature CPU_FTR_ASYM_SMT. This allows the system to detect the presence of interleaved big-cores at boot up, helpin gin spreading threads in a better way across the cores, as can be seen during the choice of the busiest sd in fair, specifically in the function update_sd_pick_busiest().

SD_PREFER_SIBLING This flag is checked during the update of the SD statistics used for load balancing (specifically during the function update_sd_lb_stats() in the fair sched class). If set, this flag informs its preference in moving trying to move their task to the sibling first, preference which is taken into account only if the siblings have the capacity to accept this tasks, otherwise the balancer proceeds as usual.

SD_OVERLAP This flag is set by the function build_sched_domains(), according to the topology, which informs when the domains are overlapped. It is used

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

by the function `update_group_capacity()` in the fair sched class to change the assumption that child groups span the current group, and adjust the operation required for a correct update of the capacities.

2.4 Objectives of the scheduler

2.4.1 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.[2] Nonetheless, Linux tries to achieve flexibility by implementing scheduling classes. Ingo Molnár, 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."[25]. 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.

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 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 system performance. 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.

2.5 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.

2.5.1 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. 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 or as a percentage of the total time of a cycle.

Interactive tasks and heuristics Dynamic priority bonuses and penalties are based on interactivity heuristics. This heuristic is implemented by keeping track of how long a task goes to sleep.[21] A process that is marked as interactive (`SCHED_NORMAL` rather than `SCHED_BATCH`) will be reinserted in the active queue after it has expired its current timeslice. This is done to improve the response time of the system. A task is also marked as interactive depending on its dynamic priority and its nice value. It is easier for a task with a higher priority to become interactive compared to task with a lower one. A task with a nice value of -20 is 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 of the dynamic priority.

This approach 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.

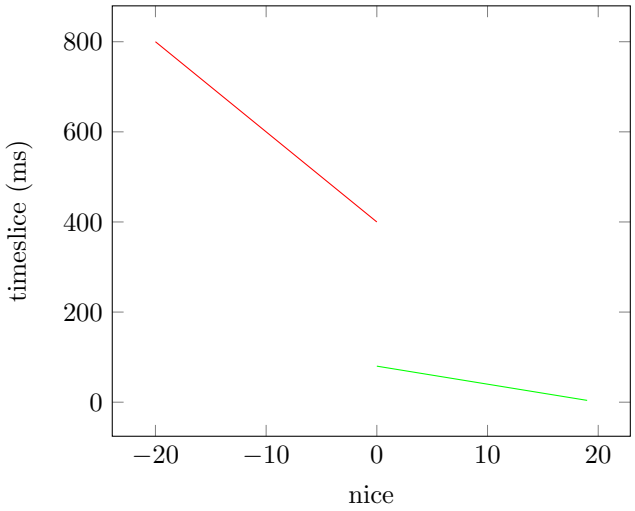


Figure 2.4: Nice to timeslice conversion

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.[2] 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 thrashing. The value of the minimum timeslice can be adjusted, but it is still `HZ` dependent.

Another problem, that can be noticed in the graph of Figure 2.4, 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()` 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.

2.5.2 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.[23]

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.

2.6 Completely Fair Scheduler (CFS)

The RDSL scheduler never made it into the kernel, but the CFS scheduler, which was developed by Ingo Molnár[25], 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.[1] This approach solves the problem found in the $O(1)$ scheduler: the behavior of nice levels is more consistent and independent of 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.

2.6.1 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

$$w(n) = \frac{1024}{(1.25)^n}. \tag{2.2}$$

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.

```
/*
 * 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. (2.2) 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%. The figure below reports the plot of the function of Equation (2.2). 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. (2.2) can be written in a more readable format as follows:

$$w(n) = \frac{2^{10}}{\left(\frac{5}{4}\right)^n} \tag{2.3}$$

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:

$$CPU_{\%} = \frac{w(n)}{w(n) + w(n+d)} \tag{2.4}$$

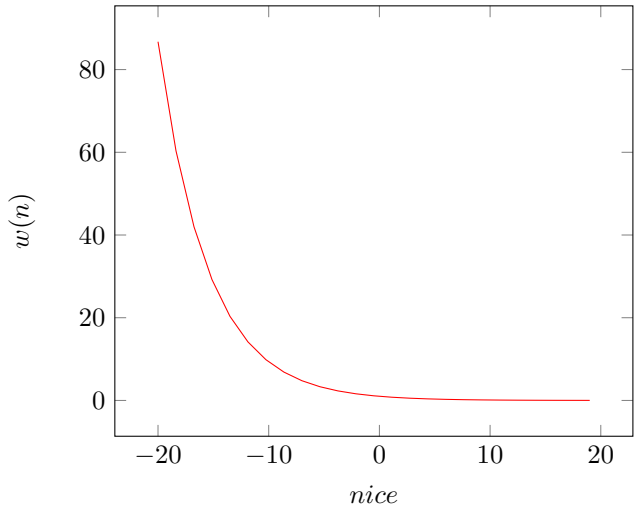


Figure 2.5: Plot of the weight as function of the nice value.

By replacing the expression for $w(n)$ of Eq. (2.3) and by setting $\alpha = \frac{5}{4}$, we find

$$CPU_{\%} = \frac{w(n)}{w(n) + w(n+d)} = \frac{\dfrac{2^{10}}{\alpha^n}}{\dfrac{2^{10}}{\alpha^n} + \dfrac{2^{10}}{\alpha^{n+d}}} =$$

$$= \frac{1}{\dfrac{\alpha^n}{2^{10}}\left(\dfrac{2^{10}}{\alpha^n} + \dfrac{2^{10}}{\alpha^{n+d}}\right)} = \frac{1}{1 + \dfrac{\alpha^n}{\alpha^{n+d}}} = \frac{1}{1 + \dfrac{1}{\alpha^d}}$$

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

$$CPU_{\%}(d) = \frac{1}{1 + \left(\dfrac{4}{5}\right)^d}. \tag{2.5}$$

This function is plotted in the figure below. Finally, by computing the derivative of Equation (2.5) 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.

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.

2.6.2 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

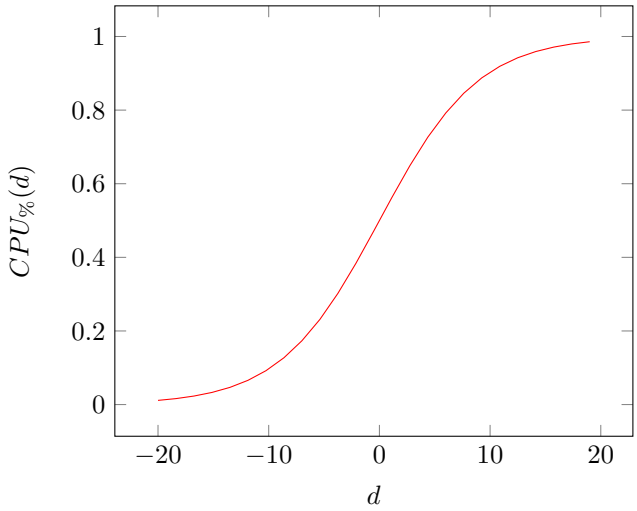


Figure 2.6: Plot of the fraction of CPU assigned to one process among two, assuming they have a difference d of the nice level.

amount of time that a task can spend on the CPU is determined by 4 values:

- The *target latency*, also called the *scheduler period*. It is defined as a period of time in which each task runs once, for at least a fixed amount of time: the *minimum granularity*.
- The *minimum granularity* 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. This amount is a tunable value called `sysctl_sched_min_granularity` 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 weight of all the task on the run queue.

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

$$assigned\_time = target\_latency \frac{task\_weight}{total\_weight} \tag{2.6}$$

Without $target\_latency$, the formula becomes the percentage of CPU time given to the process.

$$CPU_{\%} = \frac{task\_weight}{total\_weight} \tag{2.7}$$

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 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:

$$vruntime = delta\_exec \frac{weight\_of\_nice\_0}{task\_weight} \tag{2.8}$$

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

For nice values less than 0 the weighting factor is < 1: vruntime is less than the real time spent and it grows slower than real time. If the nice value is 0, the vruntime is equivalent to the actual time spent running on the CPU. This means that the virtual runtime of a 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.

Running the next task Being as fair as possible with all the task means keeping all the tasks' vruntimes 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.

2.6.3 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.

Chapter 3

Core scheduling in Linux

This chapter is an introduction to core scheduling in Linux. Core scheduling is an extremely new concept in the Linux kernel (core scheduling was added in the Linux kernel since version 5.14), thus making it subject of frequent changes and revision. While the theory will remain valid and will not change significantly, the specific APIs described in this document may become obsolete soon after the publication date.

3.1 Motivation to core scheduling

Since the discovery of hardware level vulnerabilities which exploited speculative execution such as Spectre[22], Meltdown[24] and L1TF[32], there has been a lot of research in finding ways to mitigate such vulnerabilities. One of the proposed ways was (and still is the best one in certain conditions) to disable SMT altogether. This could be doable in systems where security is more important than economical feasibility. For instance: in a datacenter, it would make no sense to disable SMT just to mitigate some vulnerabilities. The tradeoff would be too great to even consider it. The other way to tackle the issue is to mitigate it with software: core scheduling is in fact a way to mitigate hardware level vulnerabilities by acting with the software that runs on the CPU. Even if core scheduling was born as a way to mitigate hardware related vulnerabilities, other use cases are being found. Take for example a VPS[1]. What might occur is that, process A is run on the same physical core as process B. If B is extremely CPU intensive, A will take longer to complete the execution, thus billing more CPU time to the owner of process A. By using core scheduling, such behavior would not manifest, hence fairness between customers will be maintained [20].

[1] VPS stands for Virtual Private Server. It is a server that instead of running directly on dedicated hardware is virtualized and can be run alongside other VPSs

3.2 What is core scheduling

Core scheduling is a policy which requires that if two different tasks are running on two sibling[2] threads, then they must be in the same core scheduling group: only mutually trusted tasks should be able to run on the same core on two siblings. This is achieved through the actions of the scheduler. Since the scheduler is the part of the Operating System responsible for deciding in which core and which thread to run a process, it is the most logical place where core scheduling should be implemented. Core scheduling is thought to be among the software solutions to mitigate speculative execution attacks, as the only other task running on the same core, is one that already has access to the memory of the other task. Unfortunately, core scheduling makes the scheduling algorithm not work conservative. In other words, some logical cores (threads) might be put to idle even though the run queue is not empty. Core scheduling is implemented with the concept of "cookie". The idea is that if two processes are in the same core scheduling group, then they must be sharing the same cookie. *In core scheduling, we do not schedule single CPU cores, but we try to rearrange tasks running in sibling threads so that only threads in the same core scheduling group are allowed to run in parallel on two sibling threads.* The cookie management is the biggest drawback in the algorithm: it brings overhead, which must be evaluated to ensure that core scheduling is a viable solution to successfully mitigate speculative attacks.

[2]Siblings thread are the two thread that are in the same physical core. They are seen as two separate CPUs by the OS, but are in fact two threads on the same core

3.3 Linux kernel implementation

In the first implementations, CGROUPS was used to implement core scheduling. After some experimentation with CGROUPS, developers choose to implement it as an extension of the CFS scheduling algorithm and, as such, it can be either compiled or not into the kernel. As core scheduling is not strictly linked to a technology, but is rather a concept, it is still possible to implement it with CGROUPS. When core scheduling is compiled into the kernel (which happens as default with the latest releases of the kernel), a variable to store the cookie value is kept inside the `task_struct structure`[3] if the cookie is not set (i.e. it is set to 0), then all the controls required by core scheduling will not take place, otherwise core scheduling will check that the two chosen tasks (which are going to be run on sibling threads) have the same cookie. If not, one of the two sibling threads is put to idle or another task with the same cookie will be chosen from the runqueue, even if it does not have the highest priority. All these checks can be seen in the scheduler source code[4].

```
1 static inline bool cookie_match(struct task_struct *a, struct task_struct *b){
2     if (is_task_rq_idle(a) || is_task_rq_idle(b))  return true;
3     return a->core_cookie == b->core_cookie;
4 }
```

To be more precise, the function [5] that is listed above is used to check whether two tasks are compatible with each others. It is also important to note that a configuration in which a CPU thread is set to IDLE and the other sibling thread is running a task with core scheduling enabled is considered valid for core scheduling, as the idle thread is the only one considered to be part of evry core scheduling groups.

Cookies are not actually just numbers. A technique called `refcount` is used. Whenever a cookie is created, a `kmallok`[6] is carried out to allocate memory in the kernel address space and store a counter of type `refcount_t`. This counter tells how many instances are pointing to the area itself, then the cookie of the task is set to the result of the cast to unsigned long of the pointer to that area. Every time a new task is assigned to that cookie, the value of the counter is increased. If a task is cleared from that cookie, then the counter will be decreased. Once the value arrives at zero, the memory is deallocated, destroying the cookie value. This is the reason why it is not possible to allocate a custom cookie value to a task.

The API to interact with core scheduling is the function prctl. This function is not only used for core scheduling but, as stated by the kernel documentation [7], prctl is used as a general API to

[3] https://elixir.bootlin.com/linux/v5.16.16/source/include/linux/sched.h#L781
[4] https://elixir.bootlin.com/linux/v5.16.16/source/kernel/sched/core.c#L5663
[5] https://elixir.bootlin.com/linux/v5.16.16/source/kernel/sched/core.c#L5638
[6] https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html

manipulates various aspects of the behavior of the calling thread or process.

The prototype of prctl is the following:

```
1 int prctl(int option, unsigned long arg2, unsigned long arg3,unsigned long
  ↪  arg4, unsigned long arg5);
```

The interpretation of the parameters changes, according to the value of `option`, but for core scheduling are as follows[7]:

- option: `PR_SCHED_CORE`
- arg2: Command for operation, must be one among:
  - `PR_SCHED_CORE_GET` get core_sched cookie of pid.
  - `PR_SCHED_CORE_CREATE` create a new unique cookie for pid.
  - `PR_SCHED_CORE_SHARE_TO` push core_sched cookie to pid.
  - `PR_SCHED_CORE_SHARE_FROM` pull core_sched cookie from pid.
- arg3: pid of the task for which the operation applies.
- arg4: pid_type for which the operation applies. It is one among:
  - `PR_SCHED_CORE_SCOPE_THREAD`
  - `PR_SCHED_CORE_SCOPE_THREAD_GROUP`
  - `PR_SCHED_CORE_SCOPE_PROCESS_GROUP`

  For example, if arg4 is `PR_SCHED_CORE_SCOPE_THREAD_GROUP`, then the operation of this command will be performed for all tasks in the task group of pid.[8]
- arg5: user space pointer to an unsigned long for storing the cookie returned by `PR_SCHED_CORE_GET` command. Should be 0 for all other commands[9].

[7]The description of the parameters are taken directly from `https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html`

[8]If the operation applies to only a single task, then the correct value is PR_SCHED_CORE_SCOPE_THREAD.

[9]It is not strictly required to be 0: the variable is simply not used, and checks are not made, so it could be also left to a user space variable pointer.

Chapter 4

Tracing events in Linux

4.1 Introduction

It is important to understand the difference between simple event logging and tracing. The former is often used by system administrators to catch and resolve high-level issues (e.g. failed installation of programs or intrusion detection). It must be easy, so the logs must not be "noisy". On the other hand, tracing is consumed primarily by developers and logs low-level information (e.g. thrown exception). Since it handles lower level information it is necessarily "noisy", this means that reading the logs is not always intuitive or simple.

In operating system based computing environments a significant amount of a process' behaviour is defined by its interface with the operating system. This interface typically defines the process' environment such as the current directory, the input/output operations including operations of files, the execution of sub-processes and inter-process communication. Logging and interpreting the transactions between a process and the operating system it runs on, can provide data that can be used for a variety of purposes. Some of them are [29]:

Debugging A listing of a program's operating system calls allows the programmer to analyse its behaviour at a low but well defined level. Program errors can be explained in terms of the operating system calls that were (or were not) issued, and these can in turn point to the source of the error.

Profiling System calls can consume a significant amount of program run time, since the state of the machine must be saved and restored between calls. A listing of the system calls can provide hints on areas of a program that can be optimised to enhance a program's speed.

Program verification A log of a program's transactions with the operating system can be used to verify a program against its specifications or a run of a previous version. In addition, the log can be used to detect the use of non-portable functions, or programs that have been infected by viruses.

In other words, tracing is mainly used to understand what is happening and how a given system behaves meanwhile performing a certain operation. To do tracing you must always, for better or worse, be able to intercept some operations and make sure that a "trace" remains (from here the name) somewhere (e.g. file, buffer in memory). What operations are traced and how they actually do so define the tracing mechanism itself. But is tracing really useful? Computer systems, both at the hardware and software-levels, are becoming increasingly complex. In the case of Linux, used in a large range of applications, from small embedded devices to high-end servers, the size of the operating system kernels increases, libraries are added, and major software redesign is required to benefit from multi-core architectures, which are found everywhere. As a result, the software development industry and individual developers are facing problems which resolution requires to understand the interaction between applications and all components of an operating system [18].

4.2 Tracing tools

Before starting to investigate the tracing made directly by the operating system with *ftrace*, it may be interesting to briefly analyze some tools that allow a higher level of tracing. *strace* is a utility which allows you to trace the system calls that an application makes. When an application makes a system call, it is basically asking the kernel to do something (e.g. file access). Meanwhile *ftrace* is a tool used during kernel development and allows the developer to see what functions are being called within the kernel.

4.2.1 strace

strace is a diagnostic, debugging and instructional userspace utility for Linux. It is used to monitor and tamper with interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state. System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. The operation of strace is made possible by the kernel feature known as *ptrace* [8].

Let us have this simple source code:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main(int argc, char** argv)
6  {
7      int testInteger;
8      pid_t pid = getpid();
9
10     printf("My PID is: %d\n", pid);
11     printf("Enter an integer: ");
12     scanf("%d", &testInteger);
```

```
13      printf("Number = %d\n", testInteger);
14
15      return 0;
16  }
```

Once done, it will first print its *pid*. At this point, opening another terminal and giving the command:

```
$ strace -p pid
```

And the output will be seems like:

```
strace: Process 1530 attached
read(0, "5\n", 1024)                    = 2
write(1, "Number = 5\n", 11)            = 11
lseek(0, -1, SEEK_CUR)                  = -1
exit_group(0)                           = ?
+++ exited with 0 +++
```

In this specific example, there are mainly two system calls. Before, the process reads an integer with `read()` (in this case: 5) and then it prints the same integer with ther `write()`; where `read()` and `write()` are the system calls.

4.2.2 ltrace

ltrace intercepts and records dynamic library calls which are called by an executed process and the signals received by that process. It can also intercept and print the system calls executed by the program, like *strace*. Keeping in mind the same source code used by strace, let us see what happens with *ltrace*. The procedure is very similar but this time the output will be:

```
printf("Number = %d\n", 2)              = 11
+++ exited (status 0) +++
```

Unlike strace, this time `printf()` does not mean a system call but a library call. Both tools, *strace* and *ltrace*, can be useful for developers to analyze high-level software and understand where problems arise. But they certainly do not say anything about what happens inside the kernel, for this reason step by step we will now analyze a much more powerful tool useful for the described purpose.

4.3 Tracing with ftrace

Before starting the discussion of ftrace it is good to specify that it is not the only existing tool, but there are many others that are based more or less on the same principles (e.g. *perf events* [6], *eBPF* [3], *sysdig* [9], *LTTng* [5]).

Kernel debugging is a big challenge even for experienced kernel developers. For example, one difficulty is that if the system has latencies or synchronization issues (undetected race conditions), it is really hard to pinpoint where these issues are originated. Which subsystems are involved? In which conditions does the problem arise? When the system is running, there is not always a way to know the answer. *ftrace* is a debugger designed specifically to solve the issue and to ease the developer's debugging effort. Also, it is a great educational tool, not just to peek at what happens in the kernel, but also to help approach the source code.

The name comes from "function tracer", which is one of its features among many others. Each mode of tracing is simply called a *tracer*, and each one comes with many options to be tuned. Therefore *ftrace* is also very extensible because it is possible to write new tracers that can be added like a module.

4.3.1 Interfacing with ftrace

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

The 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 in this way. This is done through the `procfs` filesystem, which is found at `/proc`, as shown in Figure 4.1: every information about processes is stored here and it is 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.

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 so, no syscall is needed, except, of course, `open()` and `read()` to interact with the filesystem. On Linux, when commands such as `ps`, `top` or `pgrep` are invoked, they internally query the `procfs` filesystem. 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 convenient 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 and allows interatcion with ftrace. This filesystem is mounted by executing `mount -t debugfs nodev /sys/kernel/debug/`: since

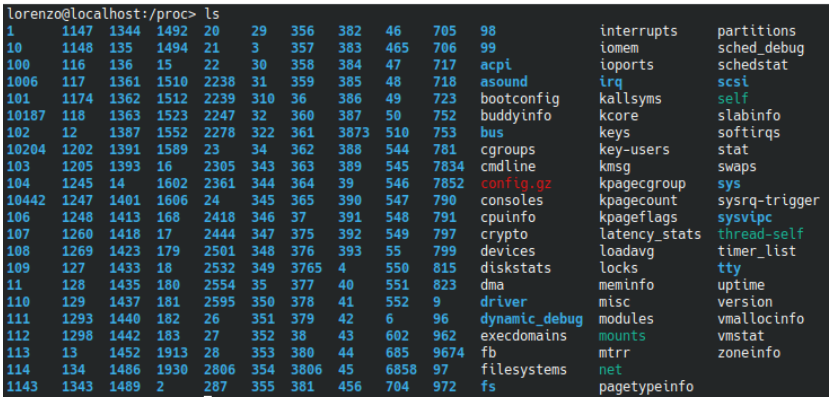


Figure 4.1: The procfs special filesystem

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.

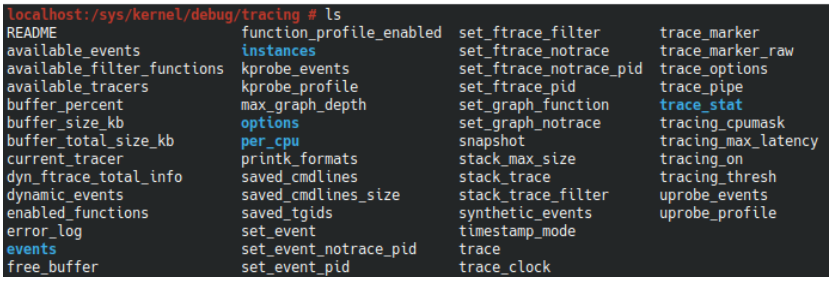


Figure 4.2: Tracing folder inside the debugfs special filesystem

The purpose of some of these files is not to set options. Rather, it is to list available options. For instance, in Figure 4.3, the available tracers are listed. These are essentially tracing modes: we activate one by doing `echo function > current_tracer`, which will immediately start to 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.

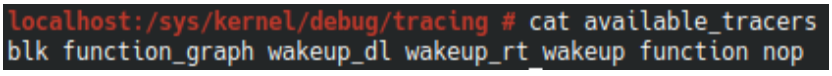


Figure 4.3: Types of tracers on by distribution (OpenSUSE Tumbleweed)

Basically, we can interact with *ftrace* using the filesystem. Later, we also analyze other methods like `trace-cmd` (Section 4.3.4) and `kernelshark` (Chapter 5).

```
$ cd /sys/kernel/debug/tracing
$ echo function > current_tracer
$ cat trace
```

4.3.2 Function tracing

Let us write a simple script that traces any input process.

```
1 #!/bin/bash
2 # traceprocess.sh
3 echo $$ > /sys/kernel/debug/tracing/set_ftrace_pid
4 echo function > /sys/kernel/debug/tracing/current_tracer
5 exec $1
```

`$$` is the variable that contains the pid of the script itself, and `$1` is the first argument 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 tracer to the function tracer
3. Execute the input program
4. The executed program will replace the process of the script itself, so the command passed as first argument to the script will be traced

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`.

```
localhost:/sys/kernel/debug/tracing # head -n 20 trace
# tracer: function
#
# entries-in-buffer/entries-written: 204971/5796026   #P:4
#
#                      _-----=> irqs-off
#                     / _----=> need-resched
#                    | / _---=> hardirq/softirq
#                    || / _--=> preempt-depth
#                    ||| /      delay
# TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#    | |         |   ||||      |         |
    ls-12284   [000] ....  2755.250236: security_inode_permission <-link_path_walk.part.0
    ls-12284   [000] ....  2755.250236: open_last_lookups <-path_openat
```

```
ls-12284   [000] ....  2755.250236: lookup_fast <-open_last_lookups
ls-12284   [000] ....  2755.250236: __d_lookup_rcu <-lookup_fast
ls-12284   [000] ....  2755.250236: step_into <-open_last_lookups
ls-12284   [000] ....  2755.250236: __follow_mount_rcu <-step_into
ls-12284   [000] ....  2755.250236: do_open <-path_openat
ls-12284   [000] ....  2755.250236: complete_walk <-do_open
ls-12284   [000] ....  2755.250236: unlazy_walk <-complete_walk
```

As expected, we only see function traced during the execution of `ls`. This information is not that useful by itself, but what is useful, instead, are the timestamps: with these, it is easy to detect latencies in the kernel. By using `kernelshark` (more details in Chapter 5) the trace can be plotted to visualize the latencies; also, this may be used to estimate which actions cause most overhead. Another way of doing this just with ftrace is to use the `function_graph` tracer: it is 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.

```
localhost:/sys/kernel/debug/tracing # head -n 150 trace
# tracer: function_graph
#
# CPU   DURATION                  FUNCTION CALLS
#  |      |   |                     |   |   |   |
   2)   0.102 us    |    } /* text_poke_flush */
   2)               |    text_poke_loc_init() {
   2)   0.109 us    |      insn_init();
   2)               |      insn_get_length() {
   2)               |        insn_get_immediate.part.0() {
   2)               |          insn_get_displacement.part.0() {
   2)               |            insn_get_sib.part.0() {
   2)               |              insn_get_modrm.part.0() {
   2)               |                insn_get_opcode.part.0() {
   2)               |                  insn_get_prefixes.part.0() {
   2)   0.100 us    |                    inat_get_opcode_attribute();
   2)   0.101 us    |                    inat_get_opcode_attribute();
   2)   0.100 us    |                    inat_get_opcode_attribute();
   2)   0.704 us    |                  }
```

This is small piece of a trace using *function_graph*. Function duration is located at every leaf function and function exit point (}). Is important to keep always in mind that the buffer can be filled and some entries could be lost: this is very common if you trace everything without filtering. To mitigate this we can trace on a single CPU, instead of all 4. This approach has three advantages:

- The output has not function calls interleaved between the CPUs, which breaks the flow of function calls
- Since fewer entries are traced, the buffer is not filled and many will not be lost

- There is a performance gain: tracing every single function call generates significant overhead.

In general, it is better to narrow the filters as much as possible. For example, it would be good to trace only the function that we are interested in, and on one CPU only.

4.3.3 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* (as shown in Figure 4.4). Now let us focus on `kvm` events. Figure 4.5 shows its contents: there is a folder for each event, containing information about it and a switch to enable/disable it4.6.

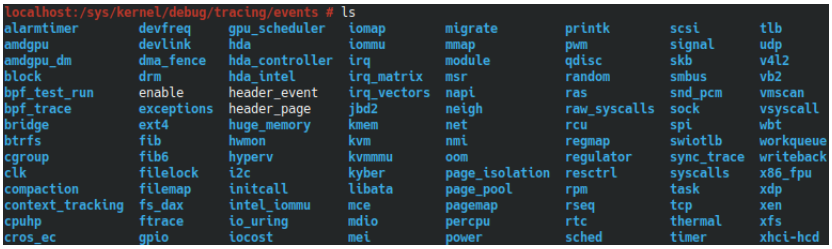


Figure 4.4: List of macro-event's section

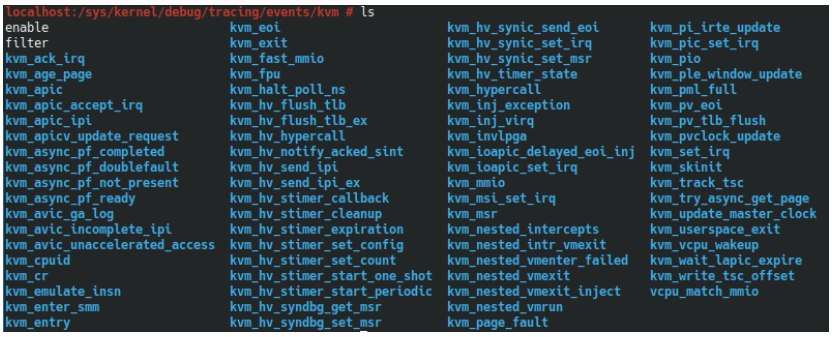


Figure 4.5: Every event associated with kvm

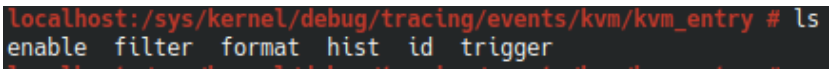


Figure 4.6: Control files for the `kvm_entry` event

Let us now see how event tracing is enabled and how to filter events. Events are not related to 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 does not trace anything), but we could also trace events while tracing functions by enabling any other tracer.

```
# enable kvm events
$ echo nop > /sys/kernel/debug/tracing/current_tracer
$ echo 1 > /sys/kernel/debug/tracing/events/kvm/enable
# enable just the kvm_entry events
$ echo nop > /sys/kernel/debug/tracing/current_tracer
$ echo 1 > /sys/kernel/debug/tracing/events/
↪  kvm/kvm_entry/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 does not 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 `kvm entry` event:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 360039/5452116   #P:4
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| /     delay
#           TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#              | |         |   ||||      |         |
       CPU 0/KVM-10145   [002] d...   663.282125: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282127: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282129: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282132: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282135: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282138: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282142: kvm_entry: vcpu 0
    # ... many more entries ...
```

In this trace the virtual cpu 0 (vCPU0) starts executing the virtual machine code on the host CPU2. So the virtual CPUs of the guests are treated exactly like all the other processes of the host. In other words, from the host's point

of view, virtual CPUs are nothing more than processes, therefore they will be scheduled together with all the other processes on the machine. Obviously you can trace the `kvm_exit` event and its operation is similar to the previous one. In particular, after this event the host code starts running again on the cpu.

4.3.4 trace-cmd

After figuring out how to use ftrace using the filesystem directly, let us analyze how to do it more immediately. The `trace-cmd` command interacts with the *ftrace* tracer that is built inside the Linux kernel. It interfaces with the *ftrace* specific files found in the debugfs file system under the tracing directory. A command must be specified to tell `trace-cmd` what to do [10].

ftrace allows you to make and specify infinite options, trace-cmd cannot be outdone in fact there are many commands and each of them has many options. The two most important commands are `record` 4.7 and `report` 4.8. Let us see an example, this time by tracing the `kvm_exit` event.

```
# Interfacing through a command-line program
$ sudo trace-cmd record -e kvm:kvm_exit
$ sudo trace-cmd report
```

The `record` command starts capturing until we stop it with the usual `ctrl+c` (interrupting signal). A `trace.dat` file with all the traced information will then be created to the current folder (Figure 4.7 shows the output of `trace-cmd record`). After the recording phase, we can read the `trace.dat` file by the `report` command (Figure 4.8 shows the output of `trace-cmd report`).

```
lorenzo@localhost:/HDD/tesi/dat> sudo trace-cmd record -e kvm:kvm_exit
Hit Ctrl^C to stop recording
^CCPU0 data recorded at offset=0x7e0000
    8192 bytes in size
CPU1 data recorded at offset=0x7e2000
    8192 bytes in size
CPU2 data recorded at offset=0x7e4000
    4096 bytes in size
CPU3 data recorded at offset=0x7e5000
    4096 bytes in size
```

Figure 4.7: Output of the command *trace-cmd record*

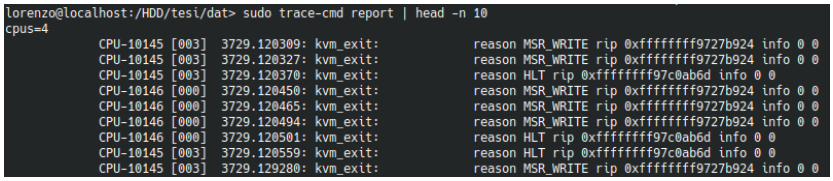


Figure 4.8: Output of the command *trace-cmd report*

Chapter 5

KernelShark

5.1 Introduction

Reading a trace file in text format can be tricky, especially when we are interested in several (and short) time intervals that are far from each other. Furthermore, a short tracing session can result in a considerably large trace file. To get around this hide'n'seek that one has to play when searching for a particular event, in 2010, **Steven Rostedt** developed *KernelShark*.
It is an application which allows the user to graphically analyze the output traces of `trace-cmd` command [4].

Summarizing to interact with ftrace is possible:

Use filesystem 4.3.1 It is expensive from the point of view of commands to give, but it is still a primitive way of obtaining information from tracing.

Use trace-cmd 4.3.4 It is less expensive to interact directly with the filesystem but the output is given on the command line, which is not always convenient for analyzing large amounts of data

Use kernelshark Finally, this graphic tool not only allows you to more comfortably analyze the data starting from a `.dat` file generated with `trace-cmd` command, but you can also directly generate the `.dat` file by setting all the options through the graphic menus 5.1.

5.2 Usage of Kernelshark

As you can see, you can select the plugin (or rather *current_tracer*; in the Figure 5.1 `nop`) and decide to trace just specific events. Therefore if we select `apply`, on the right side we can see the command which trace-cmd can perform through the command line; in this case:

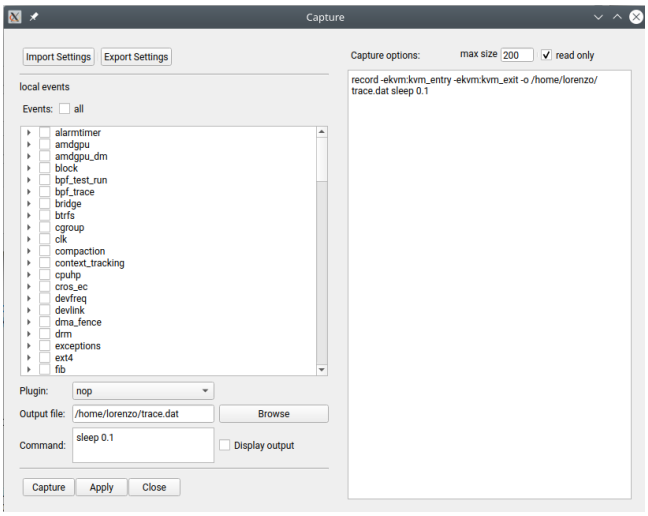


Figure 5.1: Record's tab of kernelshark

```
$ sudo trace-cmd record -ekvm:kvm_entry -ekvm:kvm_exit -o
↪  trace.dat sleep 0.1
```

Now let us understand some examples of trace analysis. In the first example, we can see a very simple switch (*sched_switch* event); Figures 5.2 5.3 5.4.

In the first step, CPU1, which is runnig Thread (pid: 7363) decided to waking (`sched_waking`) our InputThread (pid: 1480) on itself (`target_cpu=001`) with low priority (`prio=120`);

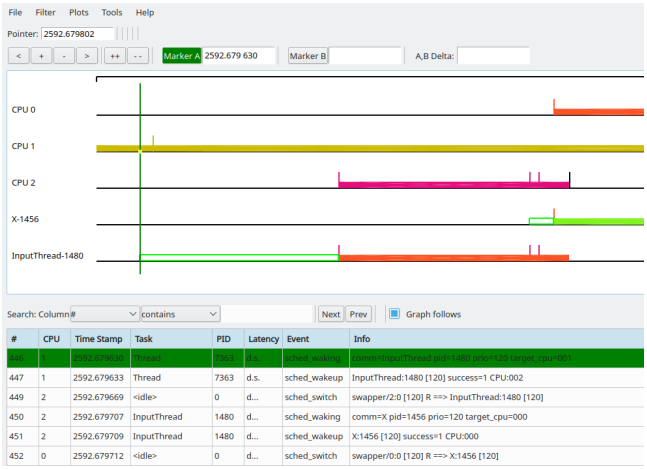


Figure 5.2: Step one of the *sched_switch*

In the second step, the same CPU1 that is running the same process Thread decided to waking up (`sched_wakeup`), this time definitly, the InputThread on CPU2 (`CPU:002`). This choice is probably due to the fact that the CPU1 was busy running Thread while the second was *idle*;

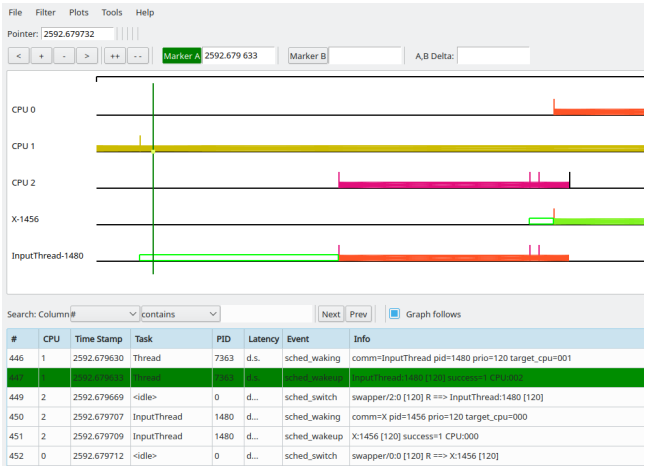


Figure 5.3: Step two of the *sched_switch*

In the last step, finally the CPU2 switch (`sched_switch`) from *idle* to running InputThread.

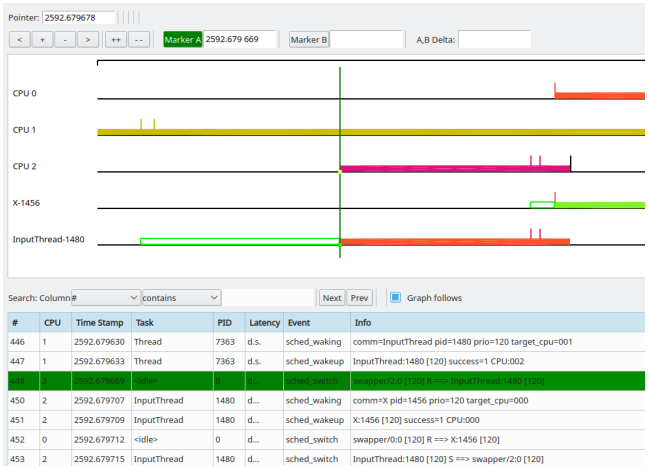
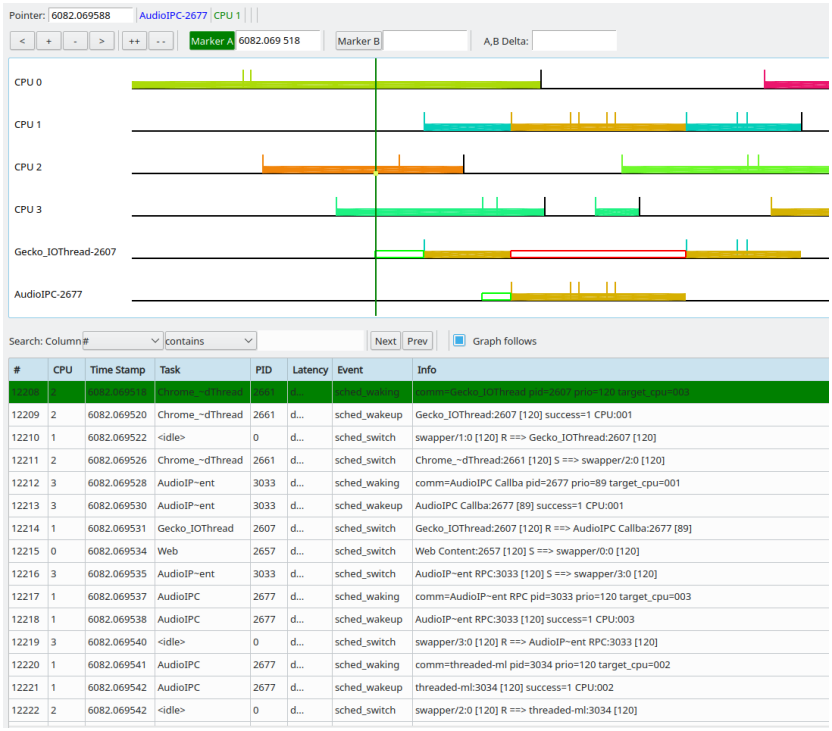


Figure 5.4: Step three of the *sched_switch*

Now that you have become familiar with the GUI, let us present and analyze an example that concerns the events of a preemption; Figure 5.5 In this example, Gecko_IOThread (pid: 2607) wuold like to run but another process, AudioIPC (pid: 2677) has an higher priority.

1. event number `12210`: Gecko_IOThread start to execute on CPU1
2. event number `12213`: a process (AudioIP) that is running on CPU3 decide to waking up AudioIPC on CPU1
3. event number `12214`: Gecko_IOThread, that would like continue use CPU1, switch in favor of AudioIPC. That happens because AudioIPC has an higher priority (89) comapred to Gecko_IOThread (120)

Kernelshark helps us to understand what is happening also through graphic conventions. The unfilled red rectangles mean just what is described now. Instead the unfilled green ones indicate the time from when the process wakes up to when it actually starts to run.



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
12208	2	6082.069518	Chrome_~dThread	2661	d...	sched_waking	comm=Gecko_IOThread pid=2607 prio=120 target_cpu=003
12209	2	6082.069520	Chrome_~dThread	2661	d...	sched_wakeup	Gecko_IOThread:2607 [120] success=1 CPU:001
12210	1	6082.069522	<idle>	0	d...	sched_switch	swapper/1:0 [120] R ==> Gecko_IOThread:2607 [120]
12211	2	6082.069526	Chrome_~dThread	2661	d...	sched_switch	Chrome_~dThread:2661 [120] S ==> swapper/2:0 [120]
12212	3	6082.069528	AudioIP~ent	3033	d...	sched_waking	comm=AudioIPC Callba pid=2677 prio=89 target_cpu=001
12213	3	6082.069530	AudioIP~ent	3033	d...	sched_wakeup	AudioIPC Callba:2677 [89] success=1 CPU:001
12214	1	6082.069531	Gecko_IOThread	2607	d...	sched_switch	Gecko_IOThread:2607 [120] R ==> AudioIPC Callba:2677 [89]
12215	0	6082.069534	Web	2657	d...	sched_switch	Web Content:2657 [120] S ==> swapper/0:0 [120]
12216	3	6082.069535	AudioIP~ent	3033	d...	sched_switch	AudioIP~ent RPC:3033 [120] S ==> swapper/3:0 [120]
12217	1	6082.069537	AudioIPC	2677	d...	sched_waking	comm=AudioIP~ent RPC pid=3033 prio=120 target_cpu=003
12218	1	6082.069538	AudioIPC	2677	d...	sched_wakeup	AudioIP~ent RPC:3033 [120] success=1 CPU:003
12219	3	6082.069540	<idle>	0	d...	sched_switch	swapper/3:0 [120] R ==> AudioIP~ent RPC:3033 [120]
12220	1	6082.069541	AudioIPC	2677	d...	sched_waking	comm=threaded-ml pid=3034 prio=120 target_cpu=002
12221	1	6082.069542	AudioIPC	2677	d...	sched_wakeup	threaded-ml:3034 [120] success=1 CPU:002
12222	2	6082.069542	<idle>	0	d...	sched_switch	swapper/2:0 [120] R ==> threaded-ml:3034 [120]

Figure 5.5: An example of preemption

Finally, let us analyze last example of tracing on the host. It may happen that a process is woken up on one cpu and (during its execution or even before) it is subsequently migrated to another core; that is the case of a `sched_migrate_task` event. In the Figure 5.6 and 5.7, there is an example of this phenomenon.

1. event number `1339`: Chrome is running on CPU3 and decide to waking Gecko_IOThread on CPU1 (5.6)

2. event number `1340`: Chrome also perform the wakeup and confirm the CPU1 for the Gecko_IOThread (5.6)

3. event number `1436`: another process (gfx) that is running in another CPU (2) perform the sched_migrate_task; we can notice that the origin CPU is still CPU1, but the destination CPU is now CPU2 (5.7)

4. shortly after there will be the switch and finally Gecko_IOThread can start its execution on the CPU2

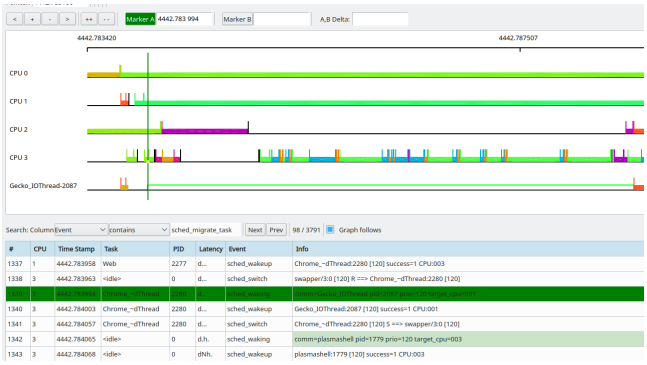


Figure 5.6: Step one migrate task

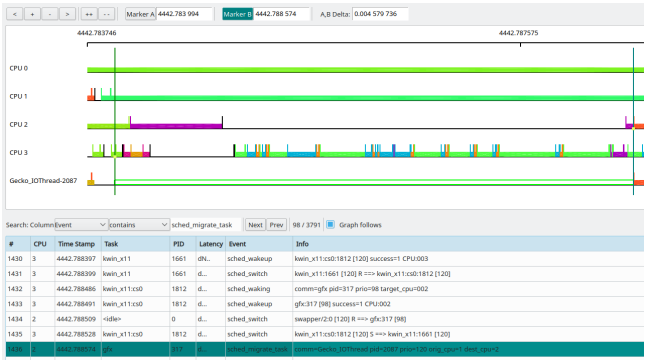


Figure 5.7: Step two migrate task

We can, also, note that kernelshark allows us to use two markers (A and B). The latter can be useful to understand the time that elapses from one specific event to another without doing anything (see in the figures the label: A, B and *Delta*). If we had to use more primitive interfaces (section 4.3.1 or section 4.3.4) than ftrace this work would have cost some time and effort.

Chapter 6

Generating synthetic workloads with rt-app

rt-app is a test application that starts multiple periodic threads to simulate a real-time periodic load. Not only the sleep and run pattern can be emulated but also the dependency between tasks like accessing the same critical resources, creating sequential wake-up or syncing the wake-up of threads.

The use case is described in a JSON-like file. The JSON file that describes a workload is made of 3 main objects: tasks, resources and global objects. You can create multiple tasks in this file, determine how much time every process has to run and select the scheduling policy. This is an example of the JSON file:

```
{
        /*
        * Simple use case which creates a thread that runs 2ms then sleeps 8ms
        * until the use case is stopped with Ctrl+C
        */
        "tasks" : {
                "thread0" : {
                        "loop" : -1,
                        "run" : 20000,
                        "sleep" : 80000
                }
        },
        "global" : {
                "duration" : 2,
                "calibration" : "CPU0",
                "default_policy" : "SCHED_OTHER",
                "logdir" : "./",
                "log_basename" : "rt-app1",
                "ftrace" : "main, task, loop, event",
        }
}
```

6.1 Global options

The following are the default global options, then there is an explanation of each one:

```
"global" : {
                "duration" : -1,
                "calibration" : "CPU0",
                "default_policy" : "SCHED_OTHER",
                "pi_enabled" : false,
                "lock_pages" : false,
                "logdir" : "./",
                "log_size" : "file",
                "log_basename" : "rt-app",
                "ftrace" : "none"
        }
```

duration The *duration* option is an integer that defines the maximum execution time in seconds. When the execution time reaches the duration time all the threads are stopped and the execution ends. If set to -1 there is no limit to the execution time.

calibration See 6.3.

default_policy The *scheduling policy* can be set at thread and phase levels. A default policy is set after the thread creation if nothing has been defined. By setting a policy you can define the scheduling policy of the thread. Accepted values are:

- "SCHED_OTHER"
- "SCHED_IDLE"
- "SCHED_RR"
- "SCHED_FIFO"
- "SCHED_DEADLINE"

If you set "SCHED_DEADLINE" as the policy you can set runtime, period and deadline values for each task. While if you set "SCHED_FIFO" you can set a priority value.

ftrace The *ftrace* option is a list of events that should be traced by ftrace during the execution. After the execution ends, a trace file is generated with all the traced events. The category names supported are:

- **main**: events generated by the main rt-app task
- **task**: events generated by a task

- **run**: events generated by the run of a task
- **loop**: events generated by each loop of a task
- **stats**: events reporting statistics on task's activation

The default value is "none".

pi_enabled The *pi_enabled* option is a boolean. If set to True enables the priority inheritance of mutex. The default value is False.

lock_pages The *lock_pages* option is a boolean. It locks the mem page in RAM. Locking the page in RAM ensures that your RT thread will not be stalled until a page is moved from swap to RAM. The lock of the page is only possible for non-CFS tasks. The default value is True.

logdir The *logdir* option is the path to store the log files. The default path is the current directory (./).

log_basename The *log_basename* option is a string. It is the prefix used for all log files of the use case. "rt-app-" is used by default.

6.2 Thread options

The following is an example of thread options, followed by the meaning of each option:

```
"tasks" : {
        "thread-name" : {
                "loop" : -1,
                "run" : 20000,
                "sleep" : 80000,
                "delay" : 10000,
                "cpus": [0, 2],
                "priority": 0,
                "dl-runtime": 2000,
                "dl-period": 1000,
                "dl-deadline": 1000
        }
}
```

delay The *delay* option is an integer that defines how many μsecs the thread should wait before starting.

loop The *loop* option is an integer that defines how many times the thread body should be executed. The default value is -1.

run The *run* option is an integer that defines how many μsecs the thread should run for each loop. This value is affected by the calibration option.

sleep The *sleep* option is an integer that defines how many μsecs the thread should sleep for each loop

cpus The *cpus* option is an array of integers. It defines CPU affinity on a thread, which means that the thread can only run on selected CPUs. An example is

$$"cpus" : [0, 2, 3]$$

priority The *priority* option is an integer that defines the thread's scheduling priority. The value must be aligned with the range allowed by the policy. The default priority is 0 for sched_other and sched_deadline class and 10 for rt classes.

dl-runtime The *dl-runtime* option is an integer that defines the runtime budget for the deadline scheduling class, expressed in μsecs. The default value is 0.

dl-period The *dl-period* option is an integer that defines the period duration for the deadline scheduling class, expressed in μsecs. The default value is runtime.

dl-deadline The *dl-deadline* option is an integer that defines the deadline parameter for the deadline scheduling class, expressed in μsecs. The default value is period.

6.3 Calibration

One important configuration option is the *calibration*. With this option, you can select which CPU compute the execution time. You can also set a calibration time manually, but if the workload on CPUs changes the execution time can be longer.

To calibrate timing on the CPU, rt-app launches a certain amount of tasks and then measures the elapsed time. Dividing the elapsed time by the number of tasks gets the "single operation unit time" which is the time taken for executing a simple operation. Then it knows how many operations it has to compute to take 1 second of execution. Fixing this value manually can lead to an inaccurate time calculation, but can run the same test regardless of CPU load. This computed value is used to compute runtime.

To waste CPU cycles - to emulate the execution of a load - a loop with some operation is cycled n times until the runtime expires. Setting run option in

configuration options, n value is computed as follows:

$$n := floor(run * 1000/calibration)$$

So a thread executes n times a computation operation, which should take the run time if the calibration option is set properly.

6.4 Scheduling policy

In rt-app is possible to select a scheduling policy and set options for that policy. For example you can set "SCHED_DEADLINE" as scheduling policy and then set "dl-runtime", "dl-period" and "dl-deadline". The scheduling policies are

- "SCHED_OTHER"
- "SCHED_IDLE"
- "SCHED_RR"
- "SCHED_FIFO"
- "SCHED_DEADLINE"

while the policies options are

- "dl-runtime"
- "dl-period"
- "dl-deadline"
- "priority"

6.5 Enable tracing

With "ftrace" option, in rt-app you can enable tracing to trace events during execution. So after execution you can see timings of each loop and each thread allowing to make statistics on this data. Using the following command you can extrapolate from report the ftrace events:

```
1 trace-cm report | grep mark_write
```

As a result, the list of events traced via ftrace will be provided with the related event times and other data. The following figure shows an example of events:

Figure 6.1: Traced events in rt-app

Chapter 7

Adding core scheduling to rt-app

7.1 Core scheduling implementation in rt-app

To test the execution of the test with core scheduling we first needed to implement it in rt-app. To do so we added a new option in threads:

core_scheduling_family: the family of the core scheduling

This setting enables core scheduling in the specified thread. This new option is an integer that:

- if set to **0** no core scheduling cookie is set to the thread
- if > 0 than if two threads have the same *core_scheduling_family* then they will share the core scheduling cookie

To do so we implemented an array of families that will hold the *pid* of the first thread of each family. This way we can generate a new cookie for each of these threads, and then we will copy this cookie to any other thread that will share the *core_cheduling_family*. This way we will have separated families with different cookies. We needed to hold the first thread because new threads can start at any time and they will need the family cookie. The following is a check that is done on each thread:

```
1 if((tdata_orig->core_scheduling_family > 0 &&
  ↪  tdata_orig->core_scheduling_family > opts.core_scheduling_families_count)
  ↪  || tdata_orig->core_scheduling_family < 0) {
2         log_error("Thread core_scheduling_family (%d) bigger then
          ↪  core_scheduling_families_count (%d)",
          ↪  tdata_orig->core_scheduling_family,
          ↪  opts.core_scheduling_families_count);
3         goto exit_err;
4 }
```

The following instead is the main code that implements the core scheduling cookie using *core_scheduling_first_threads* to hold the pids of the first thread for each family:

```
/* Set core scheduling family */
if (data->core_scheduling_family != 0) {
        if (opts.core_scheduling_families_count >=
        ↪  data->core_scheduling_family) {
                pthread_mutex_lock(&core_scheduling_mutex);
                if (core_scheduling_first_threads[data->core_sched
                ↪  uling_family-1] != 0)
                ↪  {
                        if (prctl(PR_SCHED_CORE, PR_SCHED_CORE_SHARE_FROM,
                        ↪  core_scheduling_first_threads[data->core_sched
                        ↪  uling_family-1], PR_SCHED_CORE_SCOPE_THREAD, 0))
                        ↪  {
                                perror("Error copying core scheduling
                                ↪  cookie");
                        }
                } else {
                        core_scheduling_first_threads[data->core_sched
                        ↪  uling_family-1] =
                        ↪  gettid();
                        if (prctl(PR_SCHED_CORE, PR_SCHED_CORE_CREATE,
                        ↪  gettid(), PR_SCHED_CORE_SCOPE_THREAD, 0)) {
                                perror("Error creating core scheduling
                                ↪  cookie");
                        }
                }
                pthread_mutex_unlock(&core_scheduling_mutex);
        }

        u_long test;
        prctl(PR_SCHED_CORE, PR_SCHED_CORE_GET, gettid(),
        ↪  PR_SCHED_CORE_SCOPE_THREAD, &test);
        log_notice("[%d] Core scheduling cookie set to %lu, family %d",
        ↪  data->ind, test, data->core_scheduling_family);
}
```

This implementation has a limit: we cannot edit the core scheduling family during execution, so threads cannot move to another family but they will maintain it for the full run. So we thought of another implementation: we decided to create n "dummy threads" - one for each family - that will do nothing, but they will be used to maintain core scheduling cookies. Their only function is to create a new core scheduling cookie and then pause so that other threads that want to be part of their family can copy this cookie and use it too.

7.2 rt-app configuration for tests

To perform the following tests we used different configurations for each type of test. In most tests there are two threads running for a second, on two separate

cpus in sched_deadline. The following is the JSON for non-siblings core tests at 30%:

```
{
        "tasks" : {
                "thread0" : {
                        "loop" : 1,
                        "run" : 1000000,
                        "cpus" : [0],
                        "dl-runtime" : 30000,
                        "dl-deadline" : 100000,
                        "dl-period" : 100000
                },
                "thread1" : {
                        "loop" : 1,
                        "run" : 1000000,
                        "cpus" : [1],
                        "dl-runtime" : 30000,
                        "dl-deadline" : 100000,
                        "dl-period" : 100000
                }
        },
        "global" : {
                "duration" : -1,
                "calibration" : 23,
                "default_policy" : "SCHED_DEADLINE",
                "logdir" : "./",
                "log_basename" : "rt-app1",
                "ftrace" : "loop, stats"
        }
}
```

The following example is instead the JSON configuration for the test on core siblings with different core scheduling cookie at 30%:

```
{
        "tasks" : {
                "thread0" : {
                        "loop" : 1,
                        "run" : 1000000,
                        "cpus" : [0],
                        "core_scheduling_family": 1,
                        "dl-runtime" : 30000,
                        "dl-deadline" : 100000,
                        "dl-period" : 100000
                },
                "thread1" : {
                        "loop" : 1,
                        "run" : 1000000,
                        "cpus" : [2],
                        "core_scheduling_family": 2,
                        "dl-runtime" : 30000,
                        "dl-deadline" : 100000,
                        "dl-period" : 100000
                }
        },
```

```
        "global" : {
                "duration" : -1,
                "calibration" : 23,
                "default_policy" : "SCHED_DEADLINE",
                "logdir" : "./",
                "log_basename" : "rt-app1",
                "ftrace" : "loop, stats"
        }
}
```

7.2.1 Calibration value

We have decided to keep a fixed calibration value to keep all tests the same and to prevent the results from deviating due to the calibration. To find the best value, we ran the baseline test several times, checking the total expected and real-time. By making the proportion between these two values we found the value "**23**", with the one-second test it is very close to the expected result.

7.3 Tests

7.3.1 Full load single thread baseline

Execution of a single thread with SCHED_DEADLINE set to 100%, unload CPUs and affinity on CPU0:

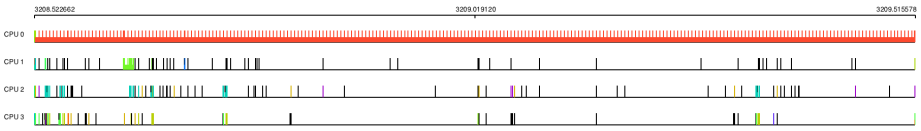


Figure 7.1: Full load single thread baseline

Bandwidth	Period thread-0	Supposed period
100%	0,990860	1

7.3.2 Full load two threads on non-sibling cores

This is a strange result. Two threads on non-sibling cores should not slow down the execution, but strangely the execution is slowed down by 10% as the expected period was 0,99.

Execution of two threads with SCHED_DEADLINE set to 100%, unload CPUs and affinity on CPU0 and CPU1:

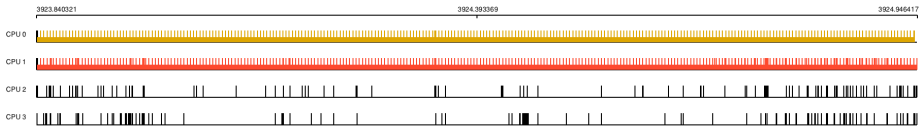


Figure 7.2: Full load two threads on non-sibling cores

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings
100%	1,100580	1,103752	0,99

7.3.3 Full load two threads on core siblings

Here we expected a slowdown, because - as seen in 3.2 - siblings CPUs are on the same physical core. So parallel executions on core siblings will slow down both of them. The slow down factor is $\alpha = 1/1.76 = 0,568$.

Execution of two threads with SCHED_DEADLINE set to 100%, unload CPUs and affinity on CPU0 and CPU2:

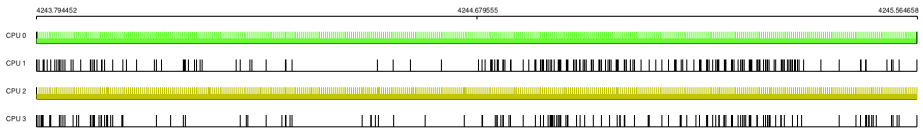


Figure 7.3: Full load two threads on core siblings

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
100%	1,766767	1,767962	0,99	1,77

7.3.4 Core not siblings

In this testing, the supposed period was always $0,99$/bandwidth. The results are all close to the supposed values.

Execution of two threads with SCHED_DEADLINE set to bandwidth, unload CPUs and affinity on CPU0 and CPU1:

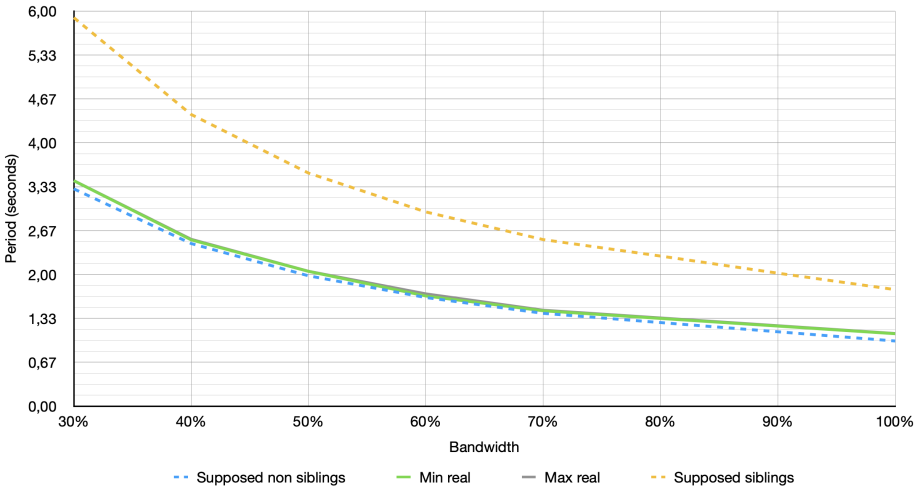


Figure 7.4: Core not siblings plot

Bandwidth	Supposed non siblings	Min real	Max real	Supposed siblings
30%	3,30	3,423969	3,424837	5,90
40%	2,47	2,525753	2,534439	4,43
50%	1,98	2,047301	2,048611	3,54
60%	1,65	1,673601	1,705910	2,95
70%	1,41	1,448606	1,457700	2,53
100%	0,99	1,100580	1,103752	1,77

Table 7.1: Core siblings and different cookie data

bandwidth: 30% SCHED_DEADLINE period set to 30%

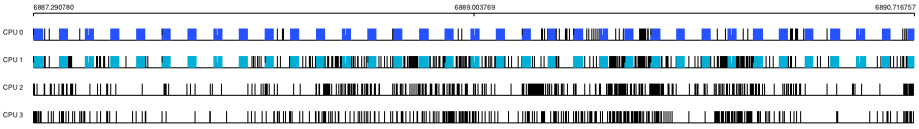


Figure 7.5: Core not siblings 30%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
30%	3,424837	3,423969	$\frac{0,99}{0,3} = 3,3$	$\frac{1,77}{0,3} = 5,9$

bandwidth: 40% SCHED_DEADLINE period set to 40%

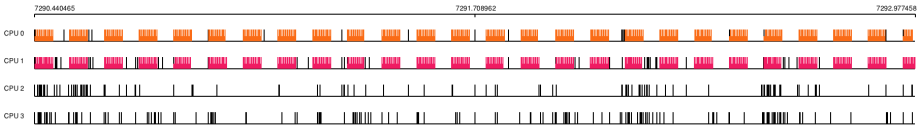


Figure 7.6: Core not siblings 40%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
40%	2,525753	2,534439	$\frac{0,99}{0,4} = 2,47$	$\frac{1,77}{0,4} = 4,43$

bandwidth: 50% SCHED_DEADLINE period set to 50%

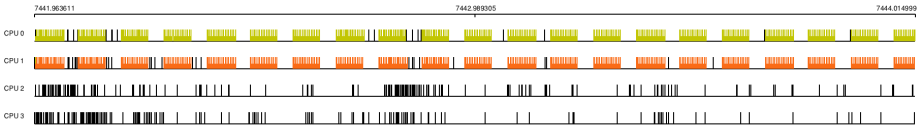


Figure 7.7: Core not siblings 50%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
50%	2,048611	2,047301	$\frac{0,99}{0,5} = 1,98$	$\frac{1,77}{0,5} = 3,54$

bandwidth: 60% SCHED_DEADLINE period set to 60%

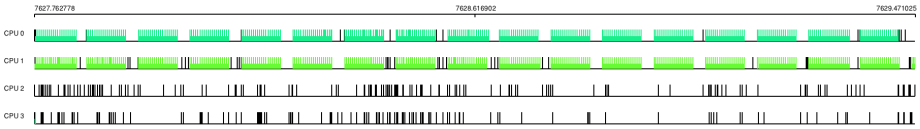


Figure 7.8: Core not siblings 60%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
60%	1,673601	1,705910	$\frac{0,99}{0,6} = 1,65$	$\frac{1,77}{0,6} = 2,95$

bandwidth: 70% SCHED_DEADLINE period set to 70%

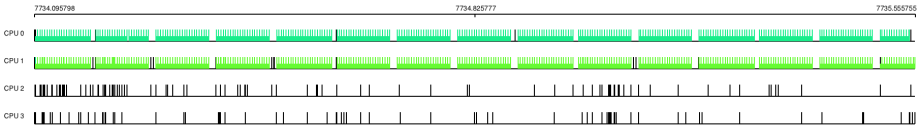


Figure 7.9: Core not siblings 70%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
70%	1,448606	1,457700	$\frac{0,99}{0,7} = 1,41$	$\frac{1,77}{0,7} = 2,53$

7.3.5 Core siblings

These tests show how the execution period depends a lot on the running of a thread on the core sibling. So we can see that basically, the scheduler tends to run the two threads at the same time, slowing down the execution. As we will see in the next tests, using core scheduling, it is possible to force threads to run at different times, speeding up execution.

Execution of two threads with SCHED_DEADLINE set to bandwidth, unload CPUs and affinity on CPU0 and CPU2:

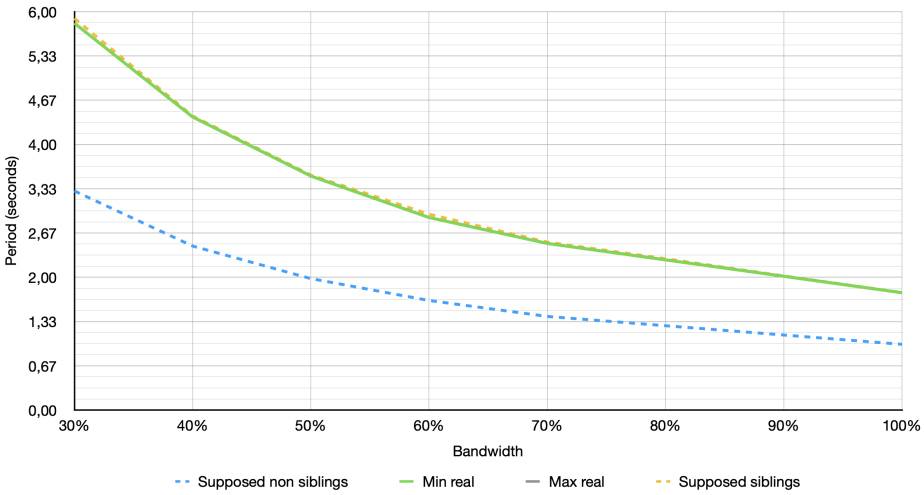


Figure 7.10: Core siblings plot

Bandwidth	Supposed non siblings	Min real	Max real	Supposed siblings
30%	3,30	5,826744	5,828010	5,90
40%	2,47	4,413743	4,415433	4,43
50%	1,98	3,523973	3,525580	3,54
60%	1,65	2,899200	2,900629	2,95
70%	1,41	2,508716	2,510359	2,53
100%	0,99	1,766767	1,767962	1,77

Table 7.2: Core siblings and different cookie data

bandwidth: 30% SCHED_DEADLINE period set to 30%

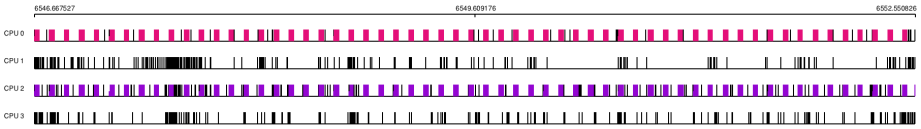


Figure 7.11: Core siblings 30%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
30%	5,826744	5,828010	$\frac{0,99}{0,3} = 3,3$	$\frac{1,77}{0,3} = 5,9$

bandwidth: 40% SCHED_DEADLINE period set to 40%

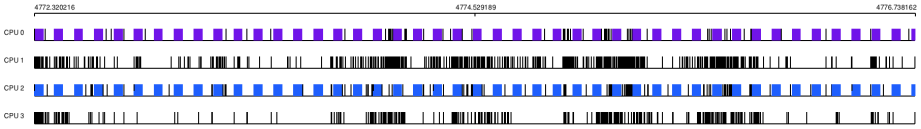


Figure 7.12: Core siblings 40%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
40%	4,413743	4,415433	$\frac{0,99}{0,4} = 2,47$	$\frac{1,77}{0,4} = 4,43$

bandwidth: 50% SCHED_DEADLINE period set to 50%

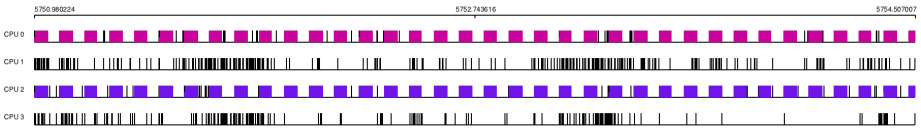


Figure 7.13: Core siblings 50%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
50%	3,523973	3,525580	$\frac{0,99}{0,5} = 1,98$	$\frac{1,77}{0,5} = 3,54$

bandwidth: 60% SCHED_DEADLINE period set to 60%

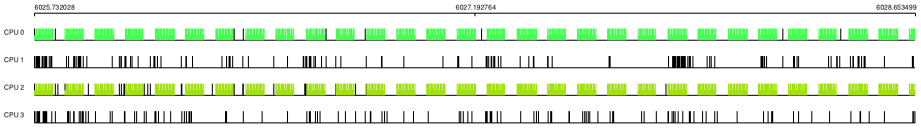


Figure 7.14: Core siblings 60%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
60%	2,899200	2,900629	$\frac{0,99}{0,6} = 1,65$	$\frac{1,77}{0,6} = 2,95$

bandwidth: 70% SCHED_DEADLINE period set to 70%

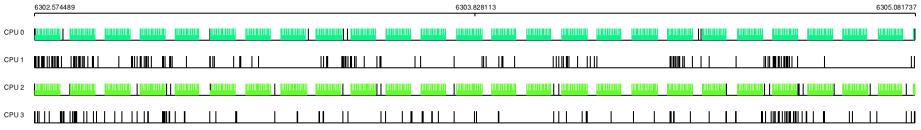


Figure 7.15: Core siblings 70%

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
70%	2,508716	2,510359	$\frac{0,99}{0,7} = 1,41$	$\frac{1,77}{0,7} = 2,53$

7.3.6 Core siblings and different core scheduling cookie

In these tests it is highlighted how using the core scheduling it is possible to better exploit the processor power. Up to 60% bandwidth by setting two different cookies on the threads it is possible to force them to run in series and not in

parallel. This makes better use of the power of the physical core and saves running time. This is an important innovation, which could be used intentionally in the development of applications that need a deadline scheduler. You could calculate the time used by the threads of your application and then apply core scheduling in some cases to speed up execution.

Execution of two threads with SCHED_DEADLINE set to bandwidth, unload CPUs, affinity on CPU0 and CPU2 and different Core Scheduling cookie:

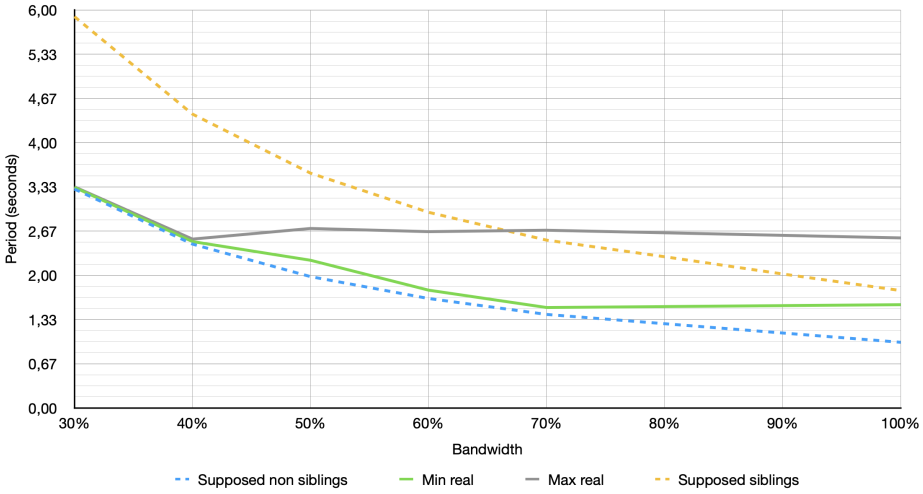


Figure 7.16: Core siblings and different cookie plot

Bandwidth	Supposed non siblings	Min real	Max real	Supposed siblings
30%	3,30	3,326325	3,333944	5,90
40%	2,47	2,508630	2,545869	4,43
50%	1,98	2,226721	2,706311	3,54
60%	1,65	1,776071	2,658218	2,95
70%	1,41	1,514196	2,679692	2,53
100%	0,99	1,557220	2,563799	1,77

Table 7.3: Core siblings and different cookie data

bandwidth: 30% SCHED_DEADLINE period set to 30%

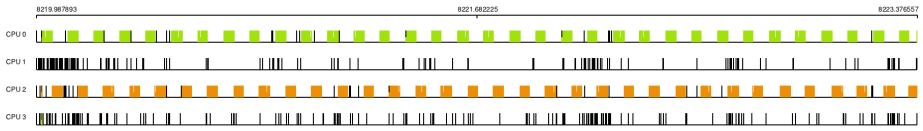


Figure 7.17: Core siblings 30% with different cookie

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
30%	3,333944	3,326325	$\frac{0,99}{0,3} = 3,3$	$\frac{1,77}{0,3} = 5,9$

bandwidth: 40% SCHED_DEADLINE period set to 40%

Here we begin to notice that the periods are no longer regular but are shorter or longer in some moments.

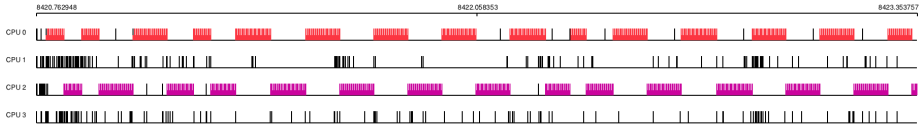


Figure 7.18: Core siblings 40% with different cookie

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
40%	2,545869	2,508630	$\frac{0,99}{0,4} = 2,47$	$\frac{1,77}{0,4} = 4,43$

bandwidth: 50% SCHED_DEADLINE period set to 50%

In this test, the period becomes more and more irregular because we are asking the scheduler for things that cannot be done due to the bandwidth. So, to respect the core scheduling constraint, the sched_deadline bandwidth constraint is not respected.

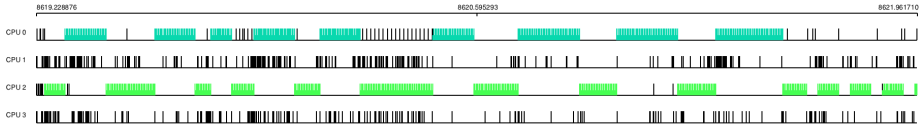


Figure 7.19: Core siblings 50% with different cookie

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
50%	2,226721	2,706311	$\frac{0,99}{0,5} = 1,98$	$\frac{1,77}{0,5} = 3,54$

bandwidth: 60% SCHED_DEADLINE period set to 60%

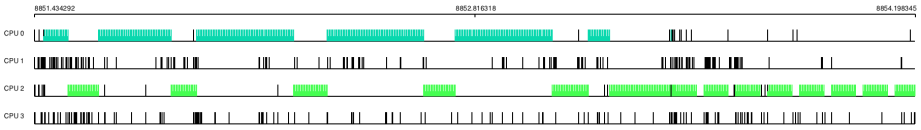


Figure 7.20: Core siblings 60% with different cookie

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
60%	1,776071	2,658218	$\frac{0,99}{0,6} = 1,65$	$\frac{1,77}{0,6} = 2,95$

bandwidth: 70% SCHED_DEADLINE period set to 70%

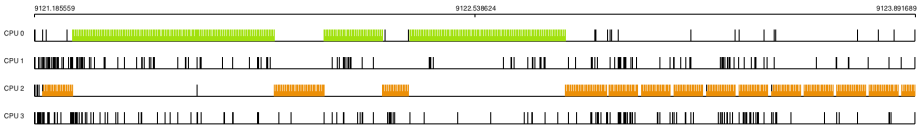


Figure 7.21: Core siblings 70% with different cookie

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
70%	1,514196	2,679692	$\frac{0,99}{0,7} = 1,41$	$\frac{1,77}{0,7} = 2,53$

bandwidth: 100% SCHED_DEADLINE period set to 100%

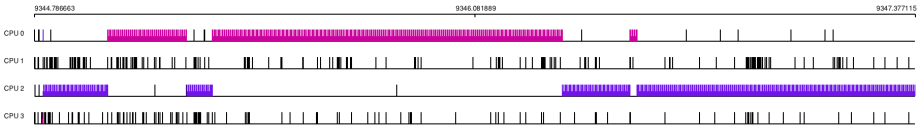


Figure 7.22: Core siblings 100% with different cookie

Bandwidth	Period thread-0	Period thread-1	Supposed non siblings	Supposed siblings
100%	1,557220	2,563799	0,99	1,77

7.4 Conclusions

As shown in figure 7.22 enabling core scheduling with multiple threads can lead to scheduling problems, slowing down execution. In case these threads are forced to run on core siblings they could have the starvation problem, as a thread could be privileged and run constantly without leaving room for his sibling. In the case examined with *SCHED_DEADLINE*, one thread is always preferred over the other with the consequent privileged and faster execution of a thread. At this point, the *SCHED_DEADLINE* parameters are no longer respected, but the constraint of core scheduling is always maintained.

In the case of an execution time of up to 50% the core scheduling allows forcing the threads not to execute simultaneously, allowing to speed up the execution on the computer with a very high slowdown rate on the core siblings. This could also be useful information for developers who, after making further tests on the computer in use, could use core scheduling as a tool to speed up execution in some particular cases.

The tool created to carry out the tests and modified for core scheduling was subsequently sent to the creators of rt-app to be viewed and possibly included as an official release of rt-app for future uses.

Bibliography

[1] Documentation of the linux kernel: Cfs design. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[2] Documentation of the linux kernel: Nice levels implementation. https://www.kernel.org/doc/Documentation/scheduler/sched-nice-design.txt.

[3] eBPF. http://www.brendangregg.com/ebpf.html. Linux Extended BPF (eBPF) Tracing Tools.

[4] Kernelshark. https://kernelshark.org/Documentation.html. kernelShark - a front end reader of trace-cmd output.

[5] LTTng. https://lttng.org. LTTng is an open source tracing framework for Linux.

[6] perf_events. http://www.brendangregg.com/perf.html. perf_events is an event-oriented observability tool, which can help you solve advanced performance and troubleshooting functions.

[7] prctl. https://man7.org/linux/man-pages/man2/prctl.2.html. prctl - The linux manual page.

[8] strace. https://strace.io. strace linux syscall tracer.

[9] Sysdig. https://sysdig.com/blog/sysdig-tracers. Sysdig tracers track and measure spans of execution in a distributed software system.

[10] trace-cmd. https://linux.die.net/man/1/trace-cmd. trace-cmd - interacts with Ftrace Linux kernel internal tracer.

[11] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[12] Lorenzo Brescia. Coupling tracing over host and guest machines. Master's thesis, University of Turin, December 2020.

[13] Antonio Casoli. Linux scheduler internals: impact of load balancing flags. Master's thesis, University of Turin, July 2020.

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

[15] Stefano Chiavazza. Linux scheduler internals: Resource management via cgroups. Master's thesis, University of Turin, July 2019.

[16] Francesco Ciraolo. A tool for testing scheduler domains' flags. Master's thesis, University of Turin, July 2020.

[17] Stefano De Venuto. Synchronization accuracy of hypervisor and virtual machine traces. Master's thesis, University of Turin, November 2021.

[18] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, École Polytechnique de Montréal, 2009.

[19] Giuseppe Eletto. Development of a kernelshark plugin for xen traces analysis. Master's thesis, University of Turin, June 2021.

[20] Dario Faggioli. Core-scheduling for virtualization: Where are we? (if we want it!). `https://youtu.be/6KZOjz3LmGc?t=358`.

[21] Jyothish Jose, Oravanpadath Sujisha, Malayamparambath Gilesh, and Thayyil Bindima. On the fairness of linux o(1) scheduler. In *5th International Conference on Intelligent Systems, Modelling and Simulation*, pages 668–674. IEEE, 2014.

[22] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[23] Con Kolivas. Rsdl completely fair starvation free interactive cpu scheduler. Linux kernel mailing list, `https://lwn.net/Articles/224654/`, 2007.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[25] Ingo Molnár. Modular scheduler core and completely fair scheduler. Linux kernel mailing list, `https://www.lwn.net/Articles/230501/`, 2007.

[26] Marco Perronet. Linux kernel: monitoring the scheduler by trace sched* events. Master's thesis, University of Turin, July 2019.

[27] Marco Edoardo Santimaria. Investigating the behavior and the overhead of core scheduling in linux. Master's thesis, University of Turin, July 2022.

[28] Constantine Shulyupin. Interactive map of linux kernel. `https://makelinux.github.io/kernel/map/`.

[29] Diomidis Spinellis. Trace: A tool for logging operating system call transactions. `https://www2.dmst.aueb.gr/dds/pubs/jrnl/1994-SIGOS-Trace/html/article.html`.

[30] Ken Thompson and Dennis Ritchie. Ken Thompson and Dennis Ritchie Explain UNIX (Bell Labs), about 1980. AT&T Bell Labs promotional film.

[31] Linus Torvalds. Torvalds about hybrid kernels. `https://www.realworldtech.com/forum/?threadid=65915%5C&curpostid=65936`.

[32] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.