

**Universität Karlsruhe**  
Fakultät für Informatik  
Institut für Betriebs- und Dialogsysteme (IBDS)  
Lehrstuhl Systemarchitektur

# Design and Implementation of Exchangeability for Linux Schedulers

## STUDY THESIS

Bin Zheng

September 19, 2007

Advisors: Prof. Dr. Frank Bellosa  
Dipl.-Inf. Andreas Merkel



## **Abstract**

The scheduler is one of the most important parts within the kernel of an operating system. For most of today's operating systems, it is built fixed in the kernel and does not allow online replacement. Any improvements to a scheduler or even replacing it with a new one will inevitably require a complete recompilation of the whole kernel and thus also a system restart and service breakdown, which is quite inconvenient especially for commercial servers with heavy loads. Therefore, to solve this problem, the goal of this thesis is to design and implement the online exchangeability of kernel schedulers by realizing them in kernel modules under Linux. Along with this objective, we will define a standard interface between schedulers and the rest of the kernel, design and implement the mechanism of switching between schedulers and finally make tests to verify that the achieved scheduler exchangeability does not cause too much performance overhead.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview . . . . .	1
1.2	Structure . . . . .	2
<b>2</b>	<b>Background and Related Work</b>	<b>3</b>
2.1	Introduction to the Linux Scheduler . . . . .	3
2.2	A Solution by Hewlett-Packard . . . . .	5
<b>3</b>	<b>Proposed Solution</b>	<b>7</b>
3.1	OS Independent Design . . . . .	7
3.2	Implementation for Linux . . . . .	8
3.2.1	Interface Abstraction - Data Structures . . . . .	8
3.2.2	Interface Abstraction - Functions . . . . .	9
3.2.3	Registration and Storage of Schedulers . . . . .	10
3.2.4	User Interface . . . . .	11
3.2.5	Invoking Scheduler Functions . . . . .	12
3.2.6	Switchover between Schedulers . . . . .	12
3.2.7	Implementation . . . . .	15
3.2.8	Boot Scheduler . . . . .	17
<b>4</b>	<b>Test and Verification</b>	<b>19</b>
4.1	Test Scheduler . . . . .	19
4.2	Function Test . . . . .	20
4.3	Performance Test . . . . .	20
4.3.1	Kernel Compilation . . . . .	22
4.3.2	SPEC Benchmark . . . . .	23
<b>5</b>	<b>Conclusion</b>	<b>25</b>



# Chapter 1

## Introduction

### 1.1 Overview

Most of the operating systems come with a scheduler as a fixed part of their kernels, for example, Windows, Linux and BSD. Ideally, this universal scheduler should be able to achieve different kinds of objectives (e.g. fairness, high throughput, good user interactivity) under all circumstances (e.g. uniprocessor, multiprocessor, heavily loaded, lightly loaded). Regretfully, such an omnipotent scheduler does not exist. All schedulers have their own weaknesses. Furthermore, there will always be new requirements which the current schedulers can not accommodate, for example, energy awareness. In such cases, an improved scheduler has to be compiled with the whole kernel, which will be later installed into a system to replace the old one running an unsuitable scheduler. Restart of the system and service breakdown are inevitable.

On the other hand, modifying code of a running kernel is nowadays possible via loadable modules. Hardware drivers are a good example. Delivered as modules, compiled driver code can be dynamically linked into and unlinked from the kernel and function the same way as any other parts of it. This feature allows users to load and unload hardware drivers any time when needed. Moreover, unlike some older unix systems, this can be done without recompilation of kernel code and any interference to the services that the system is currently providing.

So, a straightforward thought would be to implement schedulers as independent modules to achieve the flexibility that we want. But in fact, a scheduler is much more complex and important than a hardware driver. As it is extremely widely used in a kernel, other core functionalities depend heavily on the scheduler and can be easily affected by even tiny changes in it. Therefore, a standard interface between schedulers and the rest of a kernel must be defined. The schedulers which conform to this interface can be switched over from one to another without jeopardizing other parts of the kernel. An important supplement to the scheduler interface is the mechanism of handing over tasks between schedulers, as they probably have

their own task holding structures. Furthermore, in some operating systems, (e.g. Linux and Windows) certain scheduling related informations are accessible from user space. This means users and user applications can help making scheduling decisions and query informations from the scheduler. Thus, to avoid misunderstandings, those user interfaces should be kept as stable as possible. Also, as a fundamental element of an operating system, a scheduler gets woken up at the very beginning in the system initialization phase and terminates right before the whole system halts or restarts. During this period it is active almost all the time and the majority of its functions are made use of at an extremely high frequency. These characteristics lead to the fact that efficiency is one of the key points when developing schedulers and it also requires a lot of effort when designing and implementing online exchangeability between schedulers. It is unreasonable when the achieved flexibility would significantly slow the actual scheduling down.

## 1.2 Structure

There is already some related work to make schedulers more flexible and schedulers themselves evolve continually, too, with new thoughts and requirements showing up all the time. In Chapter 2, some background informations and former work will be introduced as well as their advantages and shortcomings.

Our proposed solution will be explained in detail in Chapter 3. As already mentioned in the overview, in the progress of designing and implementing the exchangeability there are some points which should be especially paid attention to. So, based on the general thought of implementing schedulers as modules, those issues and the ways to resolve them will be described in detail.

In Chapter 4, 5, tests and their results will be presented to verify our solution and its usability. Also, after summing up, a perspective of how it might progress further is going to be introduced briefly.

## Chapter 2

# Background and Related Work

Along with the development of operating systems, schedulers have also progressed enormously from batch schedulers at the very beginning to today's complex multitasking schedulers. Meanwhile, a great deal of effort has been invested to make them more flexible and capable to fulfill as many different kinds of needs as possible. The current Linux scheduler is chosen as an example here because Linux is a widespread operating system and every one is allowed to download, study and modify its code. As a matter of fact, our design and implementation also takes the kernel source code of Linux as a basis. Other than the Linux scheduler, a solution of Hewlett-Packard to implement Linux scheduling policies as loadable modules will be introduced as well because it has similar objectives and uses also analogous techniques as we do.

### 2.1 Introduction to the Linux Scheduler

As its core data structure, the 2.6 Linux scheduler [1] has for each CPU in the system a separate runqueue to keep track of all runnable tasks assigned to it and some other scheduling related information that needs to be kept on a per CPU basis. The brilliance of the runqueue is how tasks are contained in it. The following data structure serves this purpose.

```
struct prio_array {
    int nr_active;
    struct list_head queue[MAX_PRIO];
    unsigned long bitmap[MAX_PRIO+1];
}
```

The second member of this structure, named `queue`, is an array of `MAX_PRIO` heads of linked lists. `MAX_PRIO` has usually the value 140, which is the highest value of a priority and contrarily denotes the lowest priority a task can get. It means the Linux scheduler has for each priority level a list and when a task is added to a priority array, it is added to the list which corresponds to its priority level. The structure member *bitmap* of size `MAX_PRIO + 1` has bits set for each priority level

that contains active tasks. In order to find the highest priority task in a priority array, the scheduler only has to find the first bit set in the bitmap. Multiple tasks of the same priority are scheduled round-robin; after running, tasks are put at the end of their priority levels list, so, the first task in a list is always the first one that gets picked to be running within this list. Because finding the first bit in a finite-length bitmap and finding the first element in a list are both operations with a finite upper bound on how long the operation can take, the Linux scheduler realizes an  $O(1)$  scheduling algorithm for selecting the next task.

Linux tasks are classified into two categories by the scheduler according to their priorities. Soft realtime tasks have priorities from 0-99 while non-realtime task priority values map onto the internal priority range 100-140. Because RT tasks have lower priorities than non-RT tasks, they always preempt non-RT tasks. As long as RT tasks are runnable, no other tasks can run because the scheduler gives them different scheduling schemes, namely `SCHED_FIFO` and `SCHED_RR`, while non-RT tasks always operate with `SCHED_NORMAL`, which is default scheduling behavior.

`SCHED_FIFO` tasks schedule in a first-in-first-out manner. If there is a `SCHED_FIFO` task on a system it will preempt any other tasks and runs for as long as it wants to. `SCHED_FIFO` tasks do not have timeslices. Multiple `SCHED_FIFO` tasks are scheduled by priority; higher priority `SCHED_FIFO` tasks will preempt lower priority `SCHED_FIFO` tasks.

`SCHED_RR` tasks are very similar to `SCHED_FIFO` tasks, except that they use timeslices and are always preempted by `SCHED_FIFO` tasks. `SCHED_RR` tasks are scheduled by priority, and with a certain priority they are scheduled in a round-robin fashion. Each `SCHED_RR` task with a certain priority runs for its allotted timeslice, and then returns to the end of the list in its priority array queue.

Priority and scheduling schema of a task can be altered during its life time either via special function calls to the scheduler such as `sys.nice` and `sched_setscheduler` or by the scheduler itself based on interactivity heuristics to give I/O-bound threads prior access to CPUs over CPU-bound threads. How the heuristics really works is beyond the scope of this thesis.

As can be seen from the above short introduction, the current Linux scheduler has already reached certain level of flexibility. Normal and soft-realtime tasks can both be served by it. Besides, scheduling policies do not stay fixed, both users and the nature of the tasks can have impacts on them. As designed for a general-purpose operating system which concerns above all the versatility, the Linux scheduler is totally adequate.

Even though, it is still not omnipotent. There are situations which it is not able to handle. For example, in an energy aware system, a scheduler is supposed to make

scheduling decisions according to the energy consumption of the whole system and each task. The only way to adapt the current Linux scheduler is to rework its code and then recompile the kernel and that brings back the problems which have already been mentioned in the first chapter, namely system restart and service black out. Thus a clear conclusion is, despite the flexibility the current Linux scheduler has already achieved, a mechanism to realize online and seamless exchangeability between schedulers can still be beneficial and makes Linux even more versatile.

## 2.2 A Solution by Hewlett-Packard

The Management Solutions Lab of Hewlett-Packard has published a scheme to implement exchangeable Linux scheduler policies in 2000 [2]. Although this scheme is based on the 2.2 and 2.4 Linux kernel, which are quite old today, it is still worth mentioning here because it also takes advantage of loadable modules to avoid kernel recompilation and system restart when switching scheduler policies.

The most important data structure that this solution brings into the Linux kernel is the scheduling policy structure:

```
struct sched_policy {
    int sp_runnable;
    struct task_struct *(*sp_choose_task)
        (struct task_struct
         *current, int cpu);
    void (*sp_preemptability) (struct task_struct *current,
                              struct task_struct *thief,
                              int cpu);
    void (*sp_handle_ticks) (struct task_struct *current,
                             int ticks);
    char sp_private[0];
}
```

When a scheduler module is loaded into a system, a new *sched\_policy* structure must be created by its *init\_module* function and the three function pointers, namely *sp\_choose\_task*, *sp\_preemptability* and *sp\_handle\_ticks* are to be linked to the actual functions which are implemented in the module. The last element, *sp\_private* is preserved for schedulers to save their own private scheduling related information.

In the original Linux task structure, a new field is added to associate a certain *sched\_policy* to processes and for CPUs, a new global per-CPU array is declared:

```
struct sched_policy ** policy_of_cpu;
```

If a policy has been defined for particular CPUs or processes, the desired functions are invoked through the corresponding function pointers in its *sched\_policy* structure, otherwise, the logic falls through to the default Linux scheduling behavior, as

illustrated in the following code fragment in the core scheduler function *schedule*, and that is how an exchangeable scheduler substitute the default Linux scheduler to perform the actual scheduling according to its own policy.

```

if (policy_of_cpu && policy_of_cpu[this_cpu]sp_choose_task){
    next = (*policy_of_cpu)[this_cpu]\
           sp_choose_task)(prev, this_cpu);
    goto choice_made;
}

P = init_task.next_run;
/* Default process to select.. */
Next = idle_task(this_cpu);

```

This implementation is actually quite simple. It does not involve many modifications in the existing kernel functions and data structures. A standard interface is defined for scheduler developers and according to the performance measurement that the author provides; negative impacts on system performance are kept at a minimum level. These are also essential objectives which should be reached in our solution.

However, what is loadable and exchangeable here are actually scheduling policies not schedulers themselves. The default Linux scheduler is never replaced in this case, its behaviors are merely overridden on occasions when a new policy is defined for CPUs and processes. This means as well that all data structures which are used by the default scheduler have to stay untouched. For example, if we transplant this solution onto the 2.6 Linux kernel, the *runqueue* and the *prio\_array* structures which have already been mentioned in Section 2.1, can not be altered. It is just not possible if one wants to improve *prio\_array* to make the organization of runnable tasks more efficient and applicable for his own scheduling algorithms.

One other inconvenient point with this implementation is the global per-CPU array. When a scheduler module loads itself, it has to check this array to make sure there is not already a global policy defined, and stop proceeding and rolls back all work it has accomplished if there is. Therefore, at most one scheduler module can be in the system at any time. If an user needs to switch from one policy to another, he has to firstly remove the old module and than load the new one, and meanwhile the system has to suffer a scheduling policy switchover from the old one to the default Linux policy and than again from the default to the new one.

## Chapter 3

# Proposed Solution

### 3.1 OS Independent Design

Already in the introduction, we have outlined the general idea of implementing schedulers as modules. Behind this straightforward idea, there are several problems to be taken care of. The first thing is, the scheduler is not a standalone part of an operating system. Many other kernel functions rely on the scheduler to work properly. Accordingly, exchangeable schedulers and the action of switching between them should by no means interfere the other functionalities of a system. A standard interface, or more accurately, a group of functions has to be defined for this purpose. By calling these functions, expected scheduling services will be provided. How these jobs are actually done, stays opaque to the world out side of a scheduler. Any schedulers, which conform to this interface, can be considered online exchangeable.

The second point is the actual switching from one scheduler to another. With help of the standard interface, the cut-over consists of two parts, namely switching code of the methods which are declared in the interface and the hand over of currently active tasks. For the latter one, the current scheduler could connect all the active tasks together and pass them on to the next scheduler. To transfer tasks as a standard linked list solves the problem that different schedulers might organize tasks differently. The code switching of functions, on the other hand, is much more complex and operating system dependent. An example implementation under Linux will be presented later in this chapter.

The last thing is how loaded schedulers are held in the system and how users can actually switch them. Again, we can use a list structure here. A special registration block which contains basic information of a scheduler(e.g. name) will be created for each scheduler upon loading and hooked up to the list of the already installed schedulers. Users have the possibility to list all loaded schedulers and pick one of them to replace the current running one by reading and writing to a special system file. The realization of this system file is usually OS dependent.

## 3.2 Implementation for Linux

Since Linux is an open source operating system, our example implementation takes the code of its scheduler as a basis. The version of the kernel that has been used is 2.6.18. It was the latest released stable kernel when this thesis started.

### 3.2.1 Interface Abstraction - Data Structures

The effort to abstract a scheduler interface consists of two parts. The first part is to draw scheduler dependent information out of the data structure which is used by the Linux default scheduler, namely the runqueue structure that contains informations like active threads on the local CPU and the amount of them. This structure as well as per CPU static variables of this type are all defined in sched.c and not referenced anywhere else. So, it is sure that this structure is not directly accessible to the rest of the kernel and it can be safely tailored as needed. Our approach is to leave scheduler independent fields still in the runqueue structure and provide a new pointer to reference the private data block that schedulers can use to store their own information. When a new scheduler is selected to be running, it just has to initialize its own data block and redirect this pointer to it. The rest of the runqueue can stay untouched. Other than this pointer, we also add some new fields, but they are irrelevant here and will be introduced later in Section 3.4.

The reason why we split the runqueue into scheduler dependent and independent part and do not scheduler developers decide the whole structure is, some univocal members of the structure could be read indirectly from outside of the a scheduler by means of calling special facility functions exported in sched.c and are therefore supposed to be available permanently, for example, the counter which tells how many active threads are there in the runqueue. Furthermore, in order to avoid any misunderstanding, the interpretation of these fields should not vary from scheduler to scheduler. Under this circumstance, the splitting of runqueue structure brings the advantages that the per CPU runqueues will still be allocated statically by the kernel, schedulers just need to take care of the memory space for their private data, and the existing facility functions mentioned above could be reused without any modifications.

Some representative members left in the runqueue are listed as following:

```
spinlock_t lock;                /* the lock to protect
                                this structure */
unsigned long nr_running;        /* current number of
                                runnable threads */
unsigned long nr_uninterruptible; /* current number of
                                uninterruptible
                                threads*/
unsigned long long nr_switches; /* total number of
```

```

atomic_t nr_iowait;          context switches */
                             /* current number of
                             threads waiting
                             for IO */
struct task_struct *curr, *idle; /* the current and
                             idle process of
                             this CPU */
void *scheduler_private;     /* private data of
                             the of the current
                             scheduler */

```

The private data block of a scheduler comprises first of all the organization of active processes on the local CPU, for example the `prio_array` structure introduced in Section 2.1. What else falls into this category depends totally on its developers. For the default Linux scheduler, the private block is also used to save information about load balancing between processors.

### 3.2.2 Interface Abstraction - Functions

After separating the runqueue structure, the next step is to decide which functions should appear in the scheduler interface. This is the most significant job of the whole design because these functions essentially represent a scheduler. The rest of the kernel calls these functions to obtain the whole scheduling service while schedulers differ from each other by implementing these functions differently. We make the decision based on the splitted data structure and how functions defined in `sched.c` are employed.

Firstly we find out the current boundary between the default Linux scheduler and the rest of the Linux kernel. It can be done conveniently with the help of cross-references of Linux kernel code. With cross-references, we can browse the whole kernel code and click an identifier, such as a function name or a variable name, to demand all information about in which file it is declared, defined and referenced. This makes it obvious if the identifier is accessed somewhere else in the kernel. So, we have to check all non-static functions in `sched.c`, since the static functions are only reachable from inside of the file. However, there is a shortcoming with this method. The functions which do not have any external reference listed must be rechecked because the cross-reference tool does not include the files that are written in assembly language and they might use some of the scheduling functions too. One example found here is `preempt_schedule_irq`. We take the Linux pattern searching tool, `grep` as a supplement here.

After all the external referenced functions are found, we have to decide which of them are really necessary to be included in the scheduler interface. Firstly, the functions which operates on the scheduler private data (refer to Section 3.2.1) must



The *switch\_on* member of this data structure points to a function that actually makes the named scheduler to be the active scheduler in a system. The *switch\_off* member, on the other hand, turns the named scheduler to be inactive if it is the current running scheduler. Scheduler modules must provide codes for both functions because they depend exceedingly on the realization of schedulers. How these two functions are called to carry out the scheduler switchover will be explained later in this chapter.

Such a registration block can be treated as an identity of a scheduler within a system. When one scheduler is being loaded, it is responsible for creating and initializing this identity and linking it to the identity list by calling a registration function which is also provided by us. If a scheduler is not currently in use, which can be seen from the member *usage*, it can then call the deregistration function to withdraw its identity and finally release it if users request the scheduler module to be uninstalled.

### 3.2.4 User Interface

Providing an user interface to enable system administrators to access loaded schedulers and switch between them is an important aspect of the design. Under Linux, we decided to realize such an interface by taking advantage of a pseudo file system, the so called *Sysfs* [3], which is a new feature of the 2.6 Linux kernel. *Sysfs* is an in-memory filesystem which exports information about devices and drivers from the kernel device model to userspace, and is also used for online kernel configuration. This means *Sysfs* provides a way to query and influence a running kernel from userspace, which is totally suitable for our needs.

So, we set up in the virtual filesystem under the object *kernel* a new attribute *scheduler*, which is an ordinary file */sys/kernel/scheduler* if seen from the userspace. This attribute can be accessed both for reading and writing. If an user issues a command on this file to view its content, it eventually triggers to call the *show* function of the corresponding kernel attribute and this function traverses all the registration blocks linked to the scheduler list which was introduced in the last section and delivers all scheduler names with the active scheduler having a *current* tag on the side back to the userspace. This means, by reading this file a list of all loaded schedulers is printed with the active one especially accented. Based on the listed scheduler names, an authorized user can find out which scheduler is currently the active one and which others are available in the system and can therefore make an informed decision of choosing an appropriate scheduler. Choosing a new scheduler can only be done by writing its name into this special kernel attribute file, the */sys/kernel/scheduler*. This triggers the *store* function of this attribute, which seeks out the registration block of the named scheduler and carries out a scheduler switch by invoking the *switch\_off* function of the active scheduler and the *switch\_on* function of one which is about to become the active. After the *store* function returns,

the user can read again from the attribute file and this time the the *current* flag ought to appear on the side of the new scheduler.

### 3.2.5 Invoking Scheduler Functions

As already introduced in Section 3.2.2, a scheduler provides its own implementation of scheduling algorithm in a module and this algorithm itself is opaque to the outside of the module. The only way to make use of a scheduler is to invoke its functions, which are declared in the standard scheduler interface. Since these functions are mostly referenced by other parts of the kernel, which are not aware of the existence of this new interface, we are in need of a bridge to accomplish a seamless function call from the current Linux kernel to the interface functions realized by scheduler modules. To solve this problem, our design is that we keep such interface functions still in the old *sched.c* file as an entrance so that the other parts of the kernel will not have any problem finding them. But most of them are so simplified that they do not do anything else than invoking again their counterparts in the active scheduler module. By this means, the scheduling requests of the Linux kernel can be served dynamically by the current scheduler without the knowledge of where the exact functions are.

Another issue of the design is how the kernel can find the current scheduler and its functions. Since we already have an abstracted data structure *sched\_structure*, which is introduced in Section 3.2.2, to represent the scheduler interface, we assign one instance of it to each CPU of the system. As opposed to the default Linux kernel, which uses always the same scheduler on all processors, we think, it might be a beneficial feature if different CPUs can have their own schedulers. This is the reason why we make instances of *sched\_structure* CPU-local instead of using one global instance. For the purpose of creating per-CPU data instances, we use the kernel macro *DEFINE\_PER\_CPU*. As its name says, this macro defines variables on a per CPU base and these variables are allocated at kernel compilation. The function pointers of this per CPU variable are linked to the implementation of the active scheduler. Therefore, the problem of finding the current scheduler and its functions is turned into finding the data structure variable of the local CPU and this step can be easily done by using another macro *per\_cpu*. So, the whole process of calling a scheduler function consists of the following steps; the entrance defined in *sched.c* gets invoked, it does certain scheduler independent work, it retrieves the local variable of the scheduler structure and finally runs the code which has been linked to the function pointer in this variable.

### 3.2.6 Switchover between Schedulers

Based on the idea of the last section, a switchover between two schedulers seems quite intuitive. One just needs to redirect the function pointers in the local variable

of the scheduler structure to the function implementations in the new scheduler module. This way, the new scheduler will be in use and the old one will be of no effect. However, it is quite difficult to realize.

The reason is switching scheduling functions is a critical step. Some threads might run in the old functions when the switchover takes place. Suppose we have a thread that is creating a new child thread and it ends up running the *sched\_fork* function, which is also one of the scheduler interface functions, when it is being taken away from the CPU. Regardless of whether it has no more time slices left or a higher priority thread becomes active and preempts it. Coincidentally, a scheduler switchover begins while our thread is waiting to continue which means the old *sched\_fork* function that our thread is executing becomes invalid and the value of the EIP register, which points at the next instruction to be executed, of the saved context of our thread now points to something whose content is unpredictable if the old scheduler module has been removed and the memory space used by that module has been overwritten. This will most likely result in a system crash and leave behind some even more serious damages. So, this kind of situations have to be avoided in any case.

The easiest way to resolve this is that all the scheduler interface functions run in an atomic manner and to hold a spinlock and keep local interrupts disabled when the interface functions are running to assure the atomicity. It means, as long as the spinlock is free, the pointers of interface functions can be innocuously linked to new code because it is for sure no thread running the old one.

The advantage of this solution is its simplicity. The work of holding as well as releasing spinlocks and disabling as well as enabling interrupts will be done in the entrance functions defined in *sched.c* and the scheduler developers only need to make sure that their functions are unblocking which by the way is obligate anyway since these interface functions are not allowed to perform any action that blocks the current thread and moves it out of the runqueue to some waitqueue. If they do block, threads that run these functions will be taken out of a runqueue and their states will be changed to sleep either interruptibly or uninterruptibly, which leads to the fact that when a scheduler switchover takes place, all threads in the system instead of only the active ones in the runqueues have to be scanned and if some of them are running scheduler functions which block these threads to wait for some resources to be available or some events to happen we have to postpone the switching and wait until all of them finish the blocking operations and return from the scheduler functions. It not only tremendously complicates the implementation but also results in an unpredictably long time in the worst case for a scheduler exchanging to be done.

Although the solution of atomic functions is simple to realize, it also has a weakness, as it does not take into account that the 2.6 Linux kernel brings a new feature

named Kernel Preemption. Without Kernel Preemption, a thread can only be preempted, which means forced to relinquish CPU and replaced by another thread, when it is running in user mode. Once it executes kernel functions and enters kernel mode, it can not be preempted anymore until it either gets blocked and moved out of the runqueue of active threads or finishes its kernel operations and return back to user mode. The disadvantage of it is threads which are doing more critical jobs have to wait for a lower priority thread to switch back into user mode even when they are ready to run. Such situations of priority inversion are unwanted. To reduce the dispatch latency of the user mode processes and improve system interactivity, kernel preemption is implemented in Linux kernel. With this new feature, a thread can be replaced by other threads even when it is currently executing kernel code. This is relevant to our design because not all scheduler functions have to be executed without interruption. If we hold a spinlock and disable local interrupts while running them, we can assure their atomicity and have no problem substituting them with new functions. But the convenience comes at the price that it is not possible any more to pause a thread whose executing path is happen to be in one those functions and put the CPU to use for something more important, even when a short interruption is definitively harmless to the thread.

So, to be able to take advantage of Kernel Preemption we choose to abandon the simple solution with atomic functions and design a more elaborate switching mechanism. We set up a new in-scheduler flag for every thread in its task structure. This flag is actually an atomic integer and its value gets increased by one if the thread goes into a scheduler function and decreased by one when it backs out. With this flag we can easily find out whether a thread is using scheduler functions, in such cases its flag must have a positive value. The reason of taking an integer instead of a bit is because scheduler functions can be called in a nested manner. For example, while a thread is running *sched\_fork* to create a child thread, a timer interrupt occurs and calls *scheduler\_tick*, which is also a scheduler function, in its interrupt handler. Since interrupt handler works in the context of the thread that gets interrupted, it looks like the thread in our example calls *scheduler\_tick* in *sched\_fork*. In such cases, we can use the atomic integer to keep track of how deep a thread is in nested scheduler function calls. With the help of this flag and based on the fact that scheduler functions are not allowed to block, which excludes the possibility that any threads which do not appear in the runqueues are using scheduler functions, we made in the following an demonstration design of the *switch\_off* and *switch\_on* functions in the schedulers registration block which is mentioned in Section 3.2.2.

When a scheduler switchover takes place, the *switch\_off* function of the current scheduler gets called. As this function is supposed to hand over a list of all active threads to the next scheduler, it scans its private runqueue which holds all active threads and checks the in-scheduler flag of all task structures. If the flag of a thread does not have a positive value, it will just be removed from the runqueue and linked to a thread list which will later be passed on. If the value of the in-scheduler flag

is positive, the thread that is actually responsible for switching scheduler, in other words, the thread that is currently running the *switch\_off* function, puts itself on hold by calling the function *schedule* and let the in-scheduler thread run one more time. The CPU time that the in-scheduler thread gains is only for it to finish all its scheduling related operations. Once it accomplishes them, which means the value of its in-scheduler flag turns eventually zero, it loses the CPU again. This is checked every time a thread exits a scheduler function. After that, the thread that is running the *switch\_off* function gets scheduled again and goes on with the scanning. This procedure repeats again and again until there is nothing left in the runqueue and all active threads are moved to the thread list. At this point, the job of the *switch\_off* function is in fact done because the thread list to be passed to the next scheduler is ready and the functions of the current scheduler are no longer in use which means they are safe to be replaced. Now the *switch\_on* function of the new scheduler is invoked to generate its own private runqueue on the basis of the thread list and redirect pointers of the scheduler functions to the codes provided in the new scheduler module. Upon return of this function, we consider a scheduler switchover finished. Although we provide an example design of the *switch\_off* function here, it is not supposed to a standard one. Scheduler developers are free to implement it on their own will.

As opposed to the solution of atomic functions, the advantages of this design is that it allows kernel preemption in scheduler functions which gives the developers the possibility to design their schedulers in a way that follows the Linux guideline of assuring system interactivity and favoring user interactive tasks. Although this will not help improving the overall performance, it makes the system react more timely to user requests and thus make the users feel like the system has better performance. The disadvantages of this design are on one hand that it is more complex and on the other hand that it relies more on the scheduler developers to realize the scheduler exchanging mechanism correctly.

### 3.2.7 Implementation

We realized the design mentioned above both in the default Linux scheduler and the test scheduler which will be introduced in the next chapter. The implementation of the key points of this design are explained here in detail. This will make the whole idea more understandable.

The first point is the thread that performs the switch from one scheduler to another. We identify it by calling the Kernel Macro *current*, which returns the pointer of the task structure of the currently running thread, in the *store* function of the kernel attribute file */sys/kernel/scheduler* because the whole switching process takes place in this function. What it exactly does, can be found in Section 3.2.3. Once we find this thread, we give it virtually the highest priority. This is achieved by add the

following code block in the scheduling algorithm for selecting a thread to run.

```
if (unlikely(is_exch_scheduler(rq) && rq->exc)) {
    if (prev != rq->exc)
        next = rq->exc;
}
```

The variable *rq* is a pointer to the private runqueue structure, the *exc* member of this structure points to the task structure of the thread that do the switchover job and *is\_exch\_scheduler* is a macro that we provide to tell if a scheduler switchover is currently active. The code inside the brace pair has the effect that if the *exc* thread exists, it always gets picked to run next and if it is the previous thread that needs to be replaced, the next one will be picked through the normal thread selecting logic. The latter case happens when the *exc* encounters a thread which has a positive in-scheduler flag while scanning the runqueue. So, it gives up CPU by calling the function *schedule* and that in-scheduler thread will definitively be picked to run because all threads prior to it have already been removed from the runqueue to a thread list. Once this in-scheduler thread no longer uses any scheduler functions, it relinquishes the CPU by calling *schedule*. After that, the *exc* gets the CPU and proceeds with the scanning.

Secondly, we bring in a pair of new macros, namely ENTER\_SCHEDULER and EXIT\_SCHEDULER, to enwrap a call to scheduler specific code in the entrance functions in sched.c. The definition of ENTER\_SCHEDULER looks as following:

```
if (unlikely(is_exch_scheduler(rq)) {
    if (atomic_read(&x->in_scheduler) <= 0)
        schedule();
}
atomic_inc(&x->in_scheduler);
```

The *x* here represents the current thread. This macro normally increases the value of the in-scheduler flag to record that the thread just enters a scheduler function. It also prevents a thread from proceeding if a scheduler switch is taking place and this thread is not yet in any scheduler code. We do this to speed up the switch process by avoiding the case in which too many threads are still in scheduler code and we have to switch too many times between the *exc* thread and them to let them finish their scheduling work. However, if a thread is already running scheduler functions, we should not make them to relinquish its CPU because this brings two more unnecessary thread switches. One from the current one to the *exc* thread because it calls the function *schedule* and one switch from the *exc* thread back to it because it is still in scheduler functions and we have to let it run again after the if condition until it finishes all its scheduling operations. The definition of EXIT\_SCHEDULER is:

```
if (atomic_dec_and_test(&x->in_scheduler)) {  
    if (is_exch_scheduler(rq))  
        schedule();  
}
```

This macro decreases and checks the in-scheduler flag of a thread when it exits a scheduler function. If a scheduler switch is happening and the current thread is not any more in any scheduler functions, it gives up the CPU and let the *exc* continue.

There is an exception that we do not use these macros when calling a scheduler function. It is the function *schedule*. This function consists basically two parts, selecting the next thread to run and switching context of the previous thread to the one of the next thread. The second part is completely scheduler independent but finding the next thread comprises the core algorithm of a scheduler and this functionality is also provided by one of the scheduler interface function. The reason why we do not need those macros is that this interface function executes while the calling thread is holding a spinlock with interrupts disabled which is also implemented like this in the default Linux scheduler. This guarantees the atomicity of this scheduler function. So we do not have to worry about threads getting preempted when they are executing this function, hence this function can always be safely replaced while we are holding the spinlock.

### 3.2.8 Boot Scheduler

The basic idea of our solution is to implement exchangeable schedulers as independent kernel modules, which means schedulers can be added and unloaded at run time. This is, however, not always applicable. We need a built-in scheduler, which the kernel can use for the boot process. The reason is that the scheduler is one of the parts that get initialized and start operating at the very beginning in the starting phase of a Linux kernel and unlike disk driver modules, which are dispensable at first and can be loaded later from the initial ramdisk during the boot process when they are actually needed, at the time when the scheduler starts, there is not even support for modules yet. Without a correctly functioning scheduler, a system not only fails to start, it even does not provide any boot messages on the console. In this phase, users still do not have access to the system and hence have no chance to install a valid scheduler. So, users have to wait for the boot process to end to be able to load a scheduler module and the system can boot only with a valid scheduler. This is a chicken and egg problem.

We solve this problem by putting a restriction that at least one scheduler has to be built into the kernel, which therefore can not be unloaded later, and one of these built-in schedulers has to be defined for the boot purpose. This so-called boot scheduler is initialized and used when the kernel starts. After users gain access to

the system, they can choose to replace the boot scheduler with their own or keep using it because from the point of view of scheduling functionality, the boot scheduler is not different from the ones provided by other scheduler developers at all.

In our implementation, we provide two schedulers, the default Linux scheduler and another test scheduler. In the kernel configuration menu, one can choose only one or both of them to be compiled into the kernel and one of them must be selected to be the boot scheduler. The selected boot scheduler sets up its private runqueues, assigns its functions to the pointers included in the scheduler interface when the kernel initializes it and becomes available for later use. One special restriction for the boot scheduler is that at the time when it starts, the system is still in its very initial status, which means a lot of system functionalities are not ready yet, especially the memory management. This leads to the fact that functions for memory allocation, for example *kmalloc* and *\_get\_free\_page*, are not available. So, the memory space for its private runqueues and other purposes, which normally should be allocated and released dynamically when a scheduler becomes active and inactive for the sake of saving system resources, must be reserved at the compilation time of the kernel and can never be released later.

## Chapter 4

# Test and Verification

### 4.1 Test Scheduler

For testing purpose, we need to have two schedulers to carry out switchovers between them. We tailored the default Linux scheduler according to the new scheduler structure to get one candidate. For the other one, we built a simple round-robin scheduler out of the default Linux scheduler. A round-robin scheduling algorithm is quite primitive. With such algorithm, all active threads get an equal portion of CPU time, the so-called time slice, in one scheduling round and run one after another. Once a thread starts working on the CPU, it becomes unpreemptable to other threads until it runs out of its time slice or it voluntarily gives up the CPU. After all of them use up their time slice, each obtains again one slice and a new round begins.

The default Linux scheduler can be adapted to such strategy without major modifications because itself can also be seen as a round-robin scheduler which uses time slices in inconstant lengths with extra features of priority levels and thread preemption. So, what we do is always assigning the same priority to threads and disabling all the priority recalculation logic in the former Linux scheduler. After that, thread preemption is automatically stopped because preemption means that threads with higher priority can rob the CPU from threads with lower priority for themselves to proceed. Since all threads are at the same priority level now, none of them is able to preempt others any more and hence we achieve a nonpriority and nonpreemption scheduler. For the size of a time slice, we take the default length defined in the default Linux scheduler, that is 100 milliseconds, as a constant value and unlike in the default Linux scheduling algorithm, this value is completely independent from the priority and the interactive nature of a thread. We also use a single linked list to hold all active threads on the local CPU. The scheduler always picks the first thread in this list to run and after this thread finishes its timeslice, it will be given another 100 milliseconds and moved back to the last position of the list, which means it has to wait for the threads that located in front of it to finish their slice and get positioned behind it, to run again.

## 4.2 Function Test

The whole test consists of two parts. One for demonstrating the functionality of our scheduler-exchanging mechanism and the other one for verifying the usability of our design and implementation by evaluating the extra performance cost which the exchangeability brings into a system.

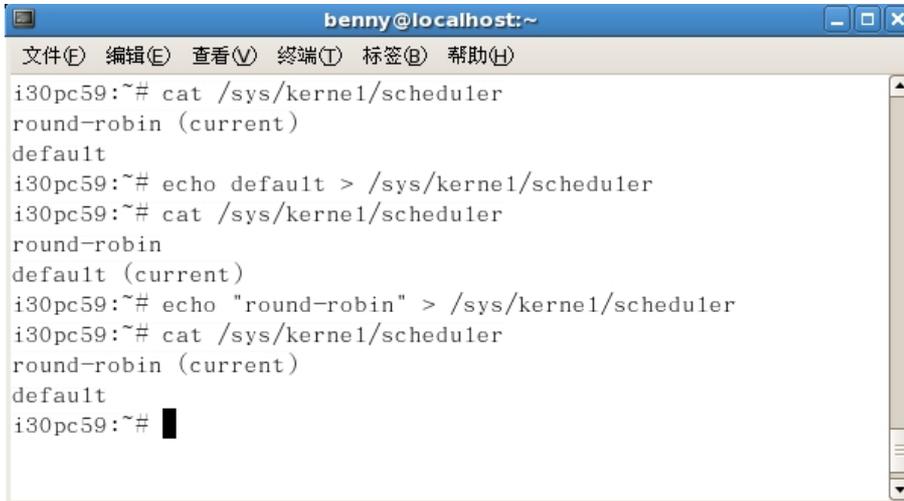
Since schedulers work in the background and users normally do not see that they are running, we print out a message which says how many active threads are currently in the system followed by a scheduler name in the function *scheduler\_tick* in both of our test candidates in order to make an exchange of schedulers visible. We choose the function *scheduler\_tick* because it will definitely be invoked every time a timer interrupt comes and since it runs in interrupt context it can tell which thread is currently running being interrupted. Also we put *printk* statements in the *switch\_on* and *switch\_off* functions to show how many threads are to be handed over to the new scheduler and how many are actually accepted by it. The numbers should be equal all the time.

In the command window (Figure 4.1), we firstly read the kernel attribute file and see there are two scheduler in our test system while *round-robin* is the active one. Then, we write "default" into this file to switch from *round-robin* to the default Linux scheduler and after that we read the file again. This time *default* becomes the active one. At last we switch again from "default" back to *round-robin* and again we can see the flag of active scheduler, the (*current*), goes back to *round-robin*.

In the console window (Figure 4.2), we can see the corresponding kernel messages from the schedulers. At first, it is the round-robin scheduler which reports the total number of threads in its runqueues and which task is running currently. During a switchover, one task is handed over from *round-robin* to *default*. After that, all the kernel messages come from the default Linux scheduler. At last, another switchover from *default* back to *round-robin* takes place. This time, three threads are successfully transferred and *round-robin* become again the active scheduler. These messages completely coincide with the command sequence that we input in the command window. It means that our scheduler exchanging mechanism delivers the expected result and therefore works correctly.

## 4.3 Performance Test

To verify the usability of our design and implementation we evaluate the performance of a kernel with scheduler exchanging ability and using the adapted default Linux scheduler against an original Linux kernel in two test scenarios. This way,

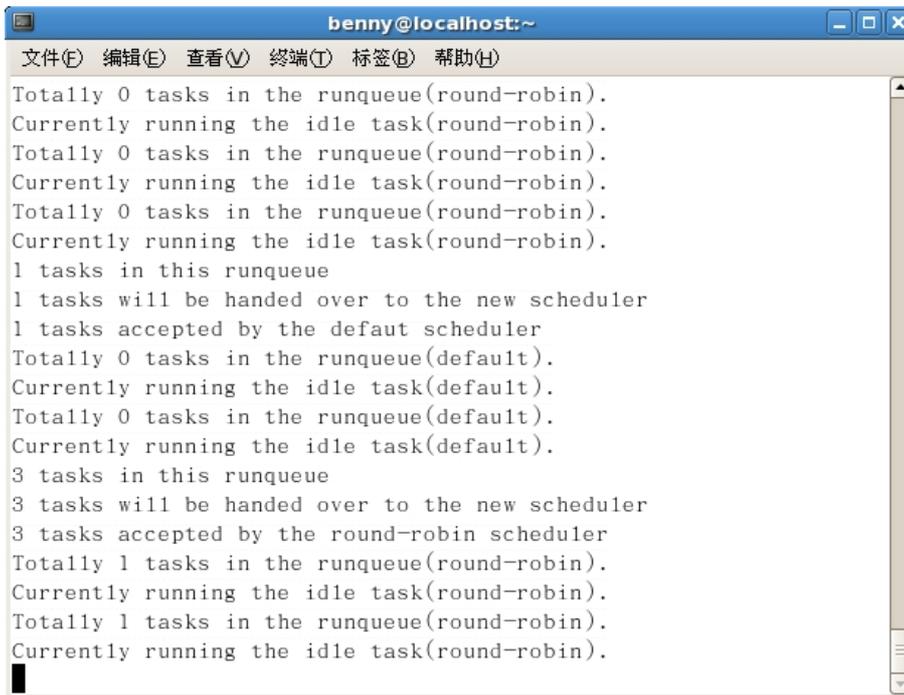


```

benny@localhost:~
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
i30pc59:~# cat /sys/kernel/scheduler
round-robin (current)
default
i30pc59:~# echo default > /sys/kernel/scheduler
i30pc59:~# cat /sys/kernel/scheduler
round-robin
default (current)
i30pc59:~# echo "round-robin" > /sys/kernel/scheduler
i30pc59:~# cat /sys/kernel/scheduler
round-robin (current)
default
i30pc59:~# █

```

Figure 4.1: Command Window of the Test Computer



```

benny@localhost:~
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
Totally 0 tasks in the runqueue(round-robin).
Currently running the idle task(round-robin).
Totally 0 tasks in the runqueue(round-robin).
Currently running the idle task(round-robin).
Totally 0 tasks in the runqueue(round-robin).
Currently running the idle task(round-robin).
1 tasks in this runqueue
1 tasks will be handed over to the new scheduler
1 tasks accepted by the default scheduler
Totally 0 tasks in the runqueue(default).
Currently running the idle task(default).
Totally 0 tasks in the runqueue(default).
Currently running the idle task(default).
3 tasks in this runqueue
3 tasks will be handed over to the new scheduler
3 tasks accepted by the round-robin scheduler
Totally 1 tasks in the runqueue(round-robin).
Currently running the idle task(round-robin).
Totally 1 tasks in the runqueue(round-robin).
Currently running the idle task(round-robin).
█

```

Figure 4.2: Command Window of the Test Computer

the performance difference between them is completely caused by the extra work of employing schedulers through the new scheduler interface.

### 4.3.1 Kernel Compilation

In the first scenario, we make standard Linux kernel compilation with the two different kernels. The reason why we choose kernel compilation is because in this process many child threads are created and terminated and switching between threads happens at a high frequency. It means accordingly that, in this test scenario, scheduler functions will be run very often and thus produce much more extra performance overhead than in a normal situation. If the extra overhead falls in an acceptable range, we can say with certain confidence that our implementation will not cause remarkable performance degradation under most of the circumstances.

Test	Total Time	User Time	System Time
1	14m6.867s	12m18.306s	1m23.021s
2	13m56.317s	12m15.386s	1m17.521s
3	13m57.029s	12m15.346s	1m17.769s
4	13m56.941s	12m15.674s	1m17.621s
5	13m56.599s	12m14.486s	1m17.889s
Average	13m58.7506s	12m15.8396s	1m18.7642s
Test	Total Time	User Time	System Time
1	14m2.384s	12m17.026s	1m20.505s
2	13m59.826s	12m16.198s	1m20.041s
3	13m59.729s	12m16.222s	1m19.689s
4	13m58.008s	12m16.090s	1m19.409s
5	14m0.329s	12m16.042s	1m19.509s
Average	14m0.052s	12m16.3156s	1m19.8306s
Overhead	0.155%	0.065%	1.354%

Table 4.1: Kernel Compilation Test

– Standard Linux Kernel

– Our Test Kernel

In the above table, we can see that the extra performance overhead caused by the online exchangeability of schedulers almost does not impact the total time and the user mode time of a kernel compilation process as they are only 0.155 and 0.065 percent longer than those on a standard Linux kernel. A noticeable time expense is in system mode which makes sense because the extra work that comes from our implementation locates only in kernel and the time it takes is calculated into

the total time when the threads of the compilation process run in system mode. The extra time expense which we have here is mainly caused by the two macros ENTER\_SCHEDULER and EXIT\_SCHEDULER which we described in Section 3.2.4.2 and the standard Linux procedure of entering and exiting function calls on scheduler interface functions. Although the time in system mode on our test kernel is noticeable longer, 1.354 percent is still a tolerable number especially considering our test scenario of calling scheduler functions at an extremely high frequency.

### 4.3.2 SPEC Benchmark

In the second test scenario we employ a standard and popular benchmark suite for evaluating system performance from the organization Standard Performance Evaluation Corporation (SPEC) which aims to produce "fair, impartial and meaningful benchmarks for computers". The benchmark which we use is CPU2006. This is a newly developed combined performance test for CPUs, memories and compilers and it covers most of the performance aspects of a computer system from program compilation, data compression to popular XML Processing and so on. This means, if our test kernel delivers a satisfying result under this test, in comparison with a standard Linux kernel, we can be fairly sure that the users will not be aware of the extra burden which the scheduler exchangeability puts on their computers.

Test Item	Standard Kernel	Test Kernel	Overhead
400.perlbench	7.09s	7.15s	0.008
401.bzip2	68.9s	69.6s	0.010
403.gcc	7.87s	7.89s	0.003
429.mcf	30.2s	30.3s	0.003
445.gobmk	94.0s	94.3s	0.003
456.hmmmer	15.7s	15.7s	0.000
458.sjeng	20.3s	20.3s	0.000
462.libquantum	0.467s	0.468s	0.002
464.h264ref	73.8s	74.2s	0.005
471.omnetpp	2.80s	2.82s	0.007
473.astar	56.0s	56.1s	0.002
483.xalancbmk	0.525s	0.524s	-0.002

Table 4.2: Benchmark Test

In the above table we can see the performance overhead of the scheduler exchangeability stays at a very low level as its value varies between -0.2 to 1 percent for different applications. The negative value here is very interesting. It is caused by the performance jitter of our test computer, otherwise this can never happen because the computer is actually doing more work with our test kernel than with a standard

kernel. But it also argues from another angle that our design and implementation do not have performance issues since the ignorable overhead can easily be covered up by the tiny uncertainty of computer performance.

## Chapter 5

# Conclusion

With the goal of realizing online exchangeability for Linux schedulers, we defined a standard interface between a scheduler and the rest of the Linux kernel, designed and implemented the mechanism of switching between schedulers on a running kernel and completed the whole solution by supplying the functionality of saving, retrieving and removing schedulers as well as the means of user interactivity. For our implementation, we took the 2.6.18 Linux kernel code as the basic.

With the results of the function and performance tests which we described in the last chapter, we can see that the primary objective of this thesis about designing and realizing scheduler online exchangeability is achieved and it is also verified that our implementation is usable from the performance point of view. However, 1.354 percent more system time can still be reduced. One possibility of improvement is to figure out a new mechanism to protect the scheduler interface structure since our current mechanism seems to be the main source of the extra performance cost.

A new feature might be worth to be developed as a supplement to our design, namely the ability to support more than one scheduler simultaneously on different CPUs. With our implementation, once a scheduler becomes active, it becomes active on all CPUs. As the current Linux kernel is able to divide available physical and logical CPUs into CPU domains to carry out load balancing within them, it might as well be a good idea to assign different scheduling policies to them. With this feature, tasks can be bound depending on their nature to a CPU domain that is running an appropriate scheduler, either with or without user interference. This will make computers more flexible and more capable of handling tasks with completely different requirements.

Another meaningful job would be to design an universal standard priority system. Although itself has nothing to do with scheduler exchanging, it will greatly help unloading the burden of scheduler developers. As we already mentioned in previous chapters, in Linux, priority information of threads are accessible from user

space. To assure coherence, scheduler developers have to map their own priority system to the existing Linux one if it does not fit. However, with a standard priority system which is ideally suitable for most of the scenarios, we can free those developers from the whole mapping job.

# Bibliography

- [1] Daniel P.Bovet and Macro Cesati: *Understanding the Linux Kernel*, O'REILLY, Edition. 3 (2005)
- [2] Scott Rhine: *Loadable Scheduler Modules on Linux(beta)*, Hewlett-Packard Management Solutions Lab(MSL) (2000)
- [3] Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman: *Linux Device Drivers*, O'REILLY, Edition. 3 (2005)

