

Technische Universität Berlin
Faculty IV Electrical Engineering and Computer Science

Bachelor's Thesis

Communication between a Plan Based Scheduler and User Space Processes in Linux

Max-Georg Debbert
Matriculation Number: 357514

Berlin, September 29, 2021

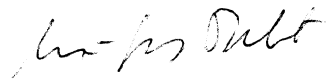
Supervisor: Barry Linnert (FU Berlin)
First Reviewer: Prof. Dr. Hans-Ulrich Hei
Second Reviewer: Prof. Dr.-Ing. Jochen Schiller (FU Berlin)

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, den 29. September 2021

A handwritten signature in black ink, appearing to read 'Max-Georg Debbert', written in a cursive style.

Max-Georg Debbert

Zusammenfassung

High-Performance-Computing-Systeme (HPC-Systeme) bestehen aus einer großen Anzahl kleinerer, vernetzter Rechner (Knoten). Sie werden verwendet, um Anwendungen mit großen Ressourcenanforderungen parallel auszuführen. Um die Einhaltung von Fristen bei der Ausführung zu gewährleisten ist ein plan-basierter Ansatz beim Scheduling nötig. Da auf den Rechenknoten in der Regel Linux als Betriebssystem zum Einsatz kommt, muss der Linux-Scheduler mit Scheduling-Plänen umgehen können. In dieser Arbeit wird untersucht, wie solche Scheduling-Pläne, die ein Benutzerprozess von einem Verwaltungsknoten des HPC-Systems empfängt, dem Linux-Scheduler zur Verfügung gestellt werden können.

Contents

1	Introduction	1
1.1	High Performance Computing	1
1.2	Grid Computing	3
2	Foundations	5
2.1	The Linux Scheduler	5
2.1.1	Scheduling Classes	5
2.2	The PB Scheduler	6
2.3	Requirements	8
3	Implementation	9
3.1	System Calls in Linux	9
3.1.1	Passing Data to the Kernel via System Calls	9
3.2	Memory in Linux	10
3.2.1	Virtual Memory	10
3.2.2	Kernel Space and User Space	10
3.2.3	Shared Memory	11
3.3	Adding a New System Call	12
3.3.1	System Call Interface	12
3.3.2	Shared Scheduling Plan Data	13
3.4	Kernel Memory Allocation	13
3.4.1	The Page Allocator	13
3.4.2	The Slab Allocator	15
3.4.3	Page Allocation for Shared Scheduling Plan Data	15
3.5	Creating a New Mapping in User Space	16
4	Conclusion and Open Questions	18
4.1	Conclusion	18
4.2	Open Questions	18
4.2.1	Dynamic Allocation of Memory	18
4.2.2	Error Handling	18
4.2.3	Synchronization	19
	References	20

List of Figures	21
Listings	22

1 Introduction

1.1 High Performance Computing

In scientific and business contexts, there are applications like weather forecasting and earthquake simulation that require huge resources (computing power, memory, etc.) to generate results in a reasonable amount of time.

Examples of applications with high resource requirements are the Weather Research and Forecasting Model (WRF) and SeisSol. WRF is a numerical weather prediction system that is used for both short-term weather forecasting (Fig. 1.1) and long-term climate modeling. SeisSol is an application for the numerical simulation of seismic wave phenomena and earthquake dynamics (Fig. 1.2).

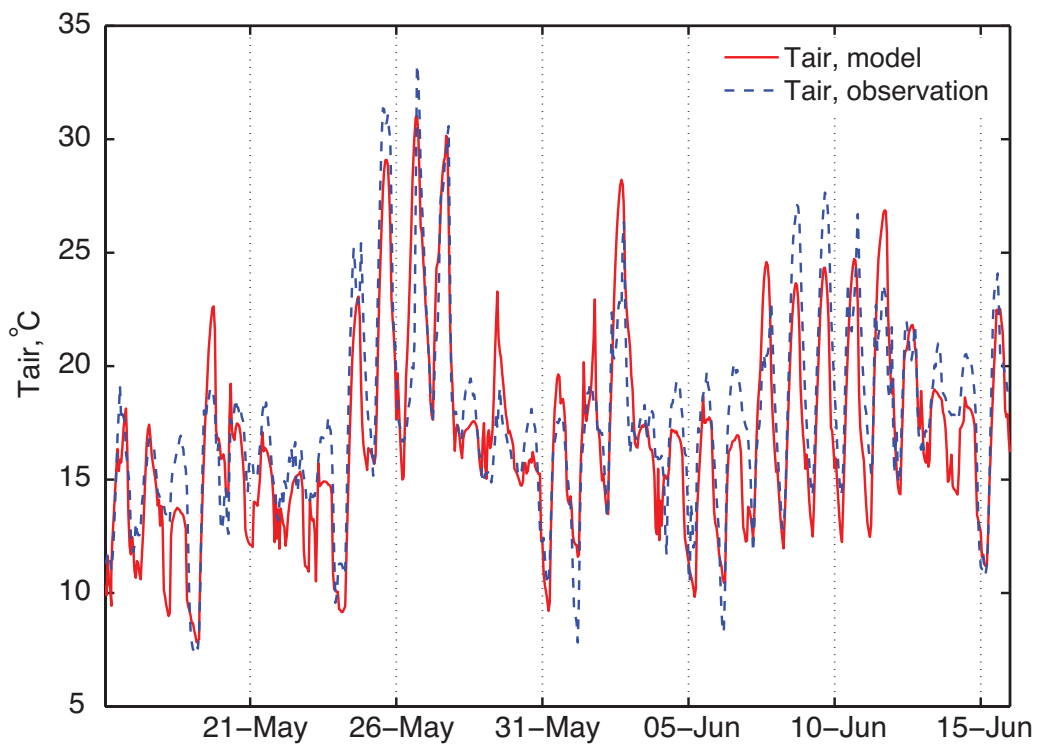


Figure 1.1: Comparison of measured air temperature with WRF simulated temperature (from [1])

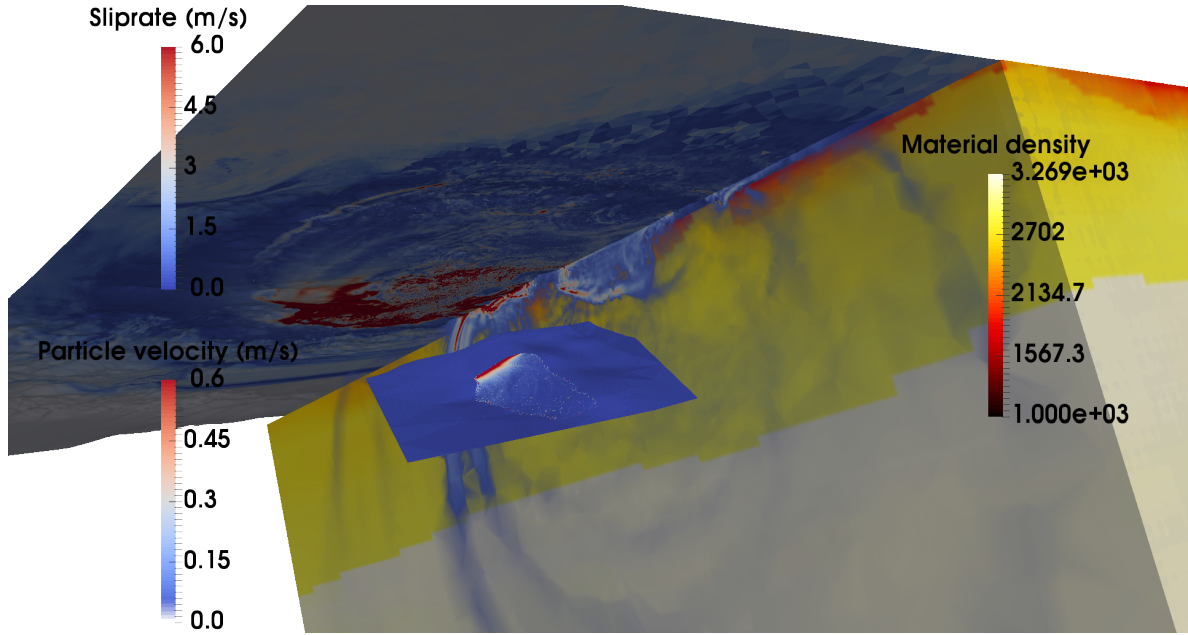


Figure 1.2: Visualization of the 1994 Northridge earthquake simulated with the CVM-H (from <https://www.seissol.org/node/31>, visited on 09/28/2021)

As shown in Fig. 1.3 the run time of WRF jobs can be substantially decreased if more CPU cores are used. Systems with an enormous amount of resources are called *high performance computing* systems (HPC systems). These supercomputers are typically composed of multiple independent computers called *nodes* running their own operating system. Nodes are interconnected via high bandwidth network technologies. The resources of an HPC system are assigned to user submitted jobs by the *resource manager*.

For example, the Cheyenne supercomputer (Fig. 1.4) used at NCAR-Wyoming Supercomputing Center (NWSC) by the National Center for Atmospheric Research (NCAR) has 4032 dual-socket compute nodes running 18-core 2.3-GHz Intel Xeon E5-2697v4 processors. Of these nodes, 3168 have 64 GB RAM each, while 864 are high memory nodes with 128 GB RAM making more than 313 TB RAM total. The nodes are connected via EDR Infiniband with a hypercube topology with 100-Gbps link bandwidth and 0.5 μ s latency. The nodes run SUSE Linux Enterprise Server 12 SP1, and jobs are scheduled with Altair PBS Pro [2].

Like most resource managers, PBS is a queuing-based batch system [3]. Users submit jobs specifying resource requirements (number of nodes, memory, etc.). When sufficient resources are available, jobs are scheduled. A maximum execution time is set for each job to prevent it from running indefinitely.

More resources may decrease the execution time of a job but increase the waiting time. For example, high memory nodes have to be used if a job requests nodes with more memory than

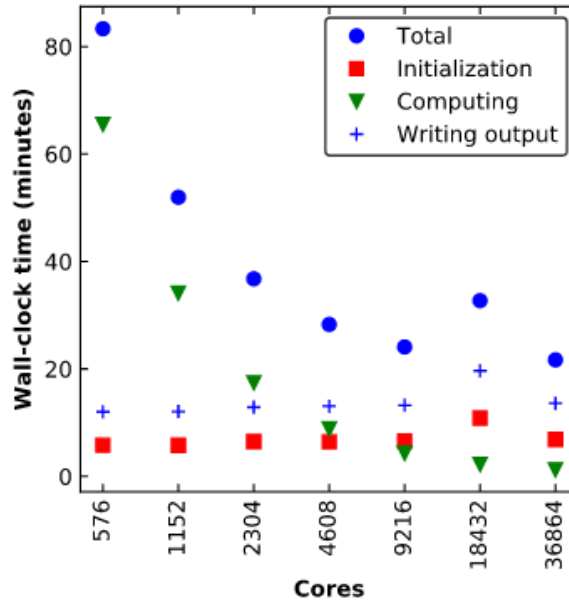


Figure 1.3: Total run time for WRF jobs with increasing numbers of cores (from <https://www2.cisl.ucar.edu/resources/wrf-scaling-and-timing>, visited on 09/28/2021)

the ordinary nodes have. Because there are fewer high memory nodes, the waiting time may be longer.

1.2 Grid Computing

Grid computing bundles the resources of various HPC systems and provides a common interface for clients. At a global level, these resources are managed by grid middleware such as the *Virtual Resource Manager* (VRM). In contrast to other approaches to grid computing, the VRM uses *service-level agreements* (SLAs) to guarantee *quality of service* (QoS), i.e. that jobs can be completed by a specified deadline [4]. Particularly in commercial deployments, this is a significant concern as different parts of a development chain depend on each other.

Keeping deadlines cannot be achieved using queuing-based resource management but requires a planning-based approach to scheduling. An existing resource management system that works based on planning is *OpenCCS* [5]. Instead of allocating resources ad hoc as jobs are executed, a plan is created beforehand that assigns resources to jobs for fixed time slots, taking into account the approximate run time of the jobs and the deadline as submitted by the job owners.

However, in this approach, the inaccuracy of the run time estimates leads to inefficient use of resources: job creators often submit estimates that are multiples of the actual run time because



Figure 1.4: Cheyenne supercomputer at the NCAR-Wyoming Supercomputing Center (from <https://www.wired.com/2017/03/put-supercomputer-wyoming/>, visited on 09/28/2021)

jobs are canceled when their assigned time slot has ended. On the other hand, requests with longer run times are more expensive.

Better utilization of the resources of HPC systems could be achieved by estimating the resource consumption of jobs on a more fundamental level, as modeled by a program graph. Splitting jobs into smaller units with known run times also allows for the shared use of nodes by different jobs.

In this approach, for each job, an execution plan is created. These plans are integrated into a global scheduling plan for an HPC system. The HPC-level scheduling plans consist of local scheduling plans for the compute nodes. These plans are not static but need to be adjusted when new jobs are submitted. In addition, planning deficiencies must be accounted for.

Because Linux, the operating system most often used on HPC nodes (see [6]), is geared towards interactive use to allow for plan-based scheduling, it is necessary to equip Linux with plan-based scheduling capabilities.

2 Foundations

In order to integrate the plan-based scheduling approach into the Linux kernel, the Linux scheduler has to be modified. The scheduler is the component of the operating system that decides which process gets to use the CPU. The scheduling mechanisms used by current Linux versions¹ do not make use of plans. This chapter explains how the Linux scheduler works and then describes how plan-based scheduling functionality was integrated into the Linux kernel in [7].

2.1 The Linux Scheduler

The Linux scheduler implements different mechanisms for scheduling through *scheduling classes*.

2.1.1 Scheduling Classes



Figure 2.1: The Linux scheduling class hierarchy (adapted from [7])

Scheduling classes are organized hierarchically (see Fig. 2.1). The main scheduling routine iterates through the scheduling classes and calls the `pick_next_task` function of the scheduling class. Tasks with a lower-ranked scheduling class are only scheduled when there are no tasks with a higher-ranked scheduling class to be run. Tasks with a higher scheduling class always preempt other tasks. The scheduling policies implemented by the scheduling classes only apply to the tasks in the scheduling class’s run queue.

Current Linux kernel versions include five scheduling classes.

Special Scheduling Classes

The highest and lowest-ranked scheduling classes are special classes that do not implement scheduling policies or schedule user tasks. The stop class preempts any tasks currently run

¹Because this thesis builds on the work done in [7], the Linux kernel version used throughout is v4.13.

and cannot be preempted by any other task. The idle class runs the idle thread if no other task is to be run.

Real-Time Scheduling Classes

Linux has two real-time scheduling classes. The deadline class schedules tasks according to a modified version of the earliest deadline first (EDF) algorithm. Each deadline task is assigned a run time (how long does the task take to run?), a period (how frequently does the task run?), and a deadline (when should the task be finished?) when it is created. The scheduler then checks whether the task requirements can be fulfilled. If not, the task is rejected. The deadline scheduler always picks the task with the earliest deadline to run.

The POSIX real-time scheduling class implements two scheduling algorithms, FIFO (first in, first out) and round-robin.

The CFS Scheduler

The default scheduling class used for non-real-time tasks is the CFS scheduling class (CFS stands for Completely Fair Scheduler). The CFS scheduling algorithm works as follows: Each task has a `vruntime` property that increases the more CPU time the task gets. The task with the lowest `vruntime` is chosen for execution.

2.2 The PB Scheduler

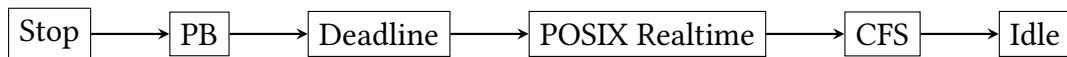


Figure 2.2: The scheduling class hierarchy with the PB class (adapted and modified from [7])

None of the existing scheduling classes offer support for plan-based scheduling. While the deadline scheduling class assigns deadlines to the tasks scheduled, because of the periodic nature of the tasks, it is unsuitable for plan-based scheduling. Therefore, a new scheduling class for this purpose was proposed in [7]. The *PB class* schedules tasks according to a scheduling plan. Because tasks scheduled by other scheduling classes should not preempt the tasks scheduled by the PB class, the PB class is placed at the top of the scheduling class hierarchy (see Fig. 2.2).

In the prototype, plan entries are represented by instances of `struct plan_entry` (see Listing 2.1). The expected execution time of a task is stored in `exec_time`. Tasks that are not part of the scheduling plan still need to be executed, for example, the process that communicates with the management nodes of the HPC system. Therefore, after executing a planned task,

```

struct plan_entry {
    u64 exec_time;
    u64 uall_time;
};

```

Listing 2.1: `struct plan_entry`

these tasks are scheduled by the CFS scheduler in the *unallocated time* specified in the field `uall_time`.

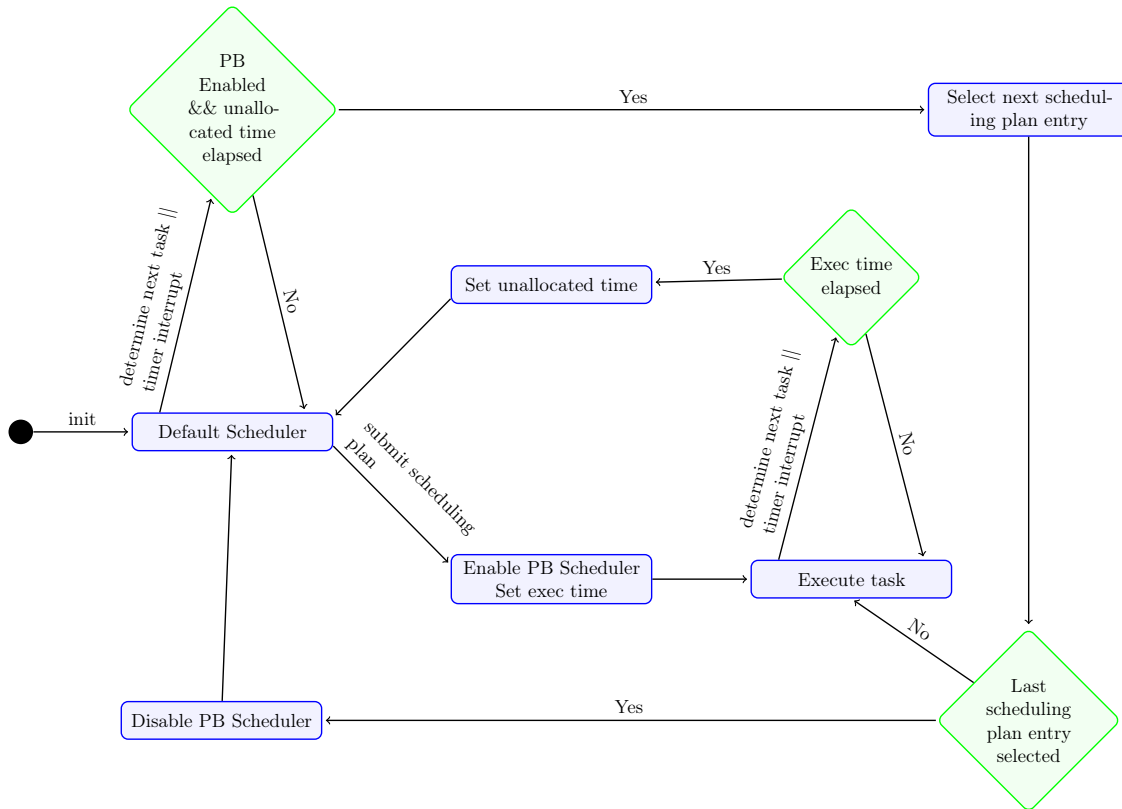


Figure 2.3: State transition diagram of the PB scheduler (from [7])

The main scheduling routine checks if there are tasks to be scheduled by the PB class by comparing the `size` and `c_entry` fields of the PB run queue structure (`struct pb_rq`, see Listing 2.2). The index of the scheduling plan entry last executed is stored in `c_entry`, while `size` represents the total number of entries in the scheduling plan.

The interaction of the PB class and the other scheduling classes (see Fig. 2.3) is controlled by the modes of the PB scheduler: disabled, execution, and unallocated. The PB scheduler is

```
struct pb_rq {
    struct plan_entry *plan;
    unsigned int size;
    unsigned int c_entry;
    ...
    int mode;
};
```

Listing 2.2: `struct pb_rq`

disabled if there are no tasks in the current scheduling plan or all tasks of the scheduling plan have already finished executing (i.e. `c_entry == size`).

If there are unfinished tasks in the scheduling plan, the PB scheduler switches to execution mode. The next task in the plan array is executed without interruption. When the task is finished, the PB scheduler switches to unallocated mode, and the default scheduler is allowed to schedule tasks for the amount of time specified in the `uall_time` field of the plan entry. When that time has elapsed, the PB class switches from unallocated mode to execution mode, and the next task in the scheduling plan is picked for execution.

This thesis investigates how scheduling plans can be made available to the PB scheduler. While the PB scheduler is part of the kernel, the scheduling plans are received via the network by a user process. Therefore, for the scheduler to access scheduling plans, data must be shared between the user process and the kernel.

2.3 Requirements

The scheduling plan is a dynamic entity that changes when new jobs are submitted to the HPC system or if the existing plan cannot be complied with due to resource outages or deficiencies in the resource consumption estimates of tasks. Changes to the node scheduling plans may happen frequently. In order to prevent the node from working with outdated scheduling plans for too long, changes to the node scheduling plan should be made available to the Linux scheduler as fast as possible. Therefore, the interface between the user process that receives the node scheduling plan from the HPC system and the Linux kernel should have minimal overhead, i.e. use few system call invocations and avoid memory copy operations.

Kernel data is isolated from user processes for security reasons. When sharing data between a user process and the Linux kernel, it must be ensured that no other kernel data is made accessible to user processes.

3 Implementation

In order to make scheduling plans available to the Linux scheduler, the user process has to communicate with the kernel. This is typically done using *system calls*.

3.1 System Calls in Linux

System calls are interfaces that allow user processes to interact with the kernel. When a user process issues a system call, the operating system executes the requested actions in kernel mode.

3.1.1 Passing Data to the Kernel via System Calls

User processes can pass data to the kernel via system call parameters. System call parameters are passed in registers. Which registers are used and how many parameters can be passed is architecture-dependent. For example, in x86 systems, system calls can take up to six arguments.

Because system call parameters are typically passed in registers, it is not possible to directly transfer a large amount of data (e.g. a C struct). In order to accomplish this, a user process can pass a pointer to data in user memory as a system call parameter. The system call can then copy the data from user memory to kernel memory via kernel internal functions like `copy_from_user`.

To transfer scheduling plan data from a user process to the kernel in this fashion would require a large amount of overhead, i.e. one system call invocation and one memory copy operation for each change to the scheduling plan. For this reason, a shared memory approach is more suitable. To understand how to implement this, we first look at how the Linux memory management system works.

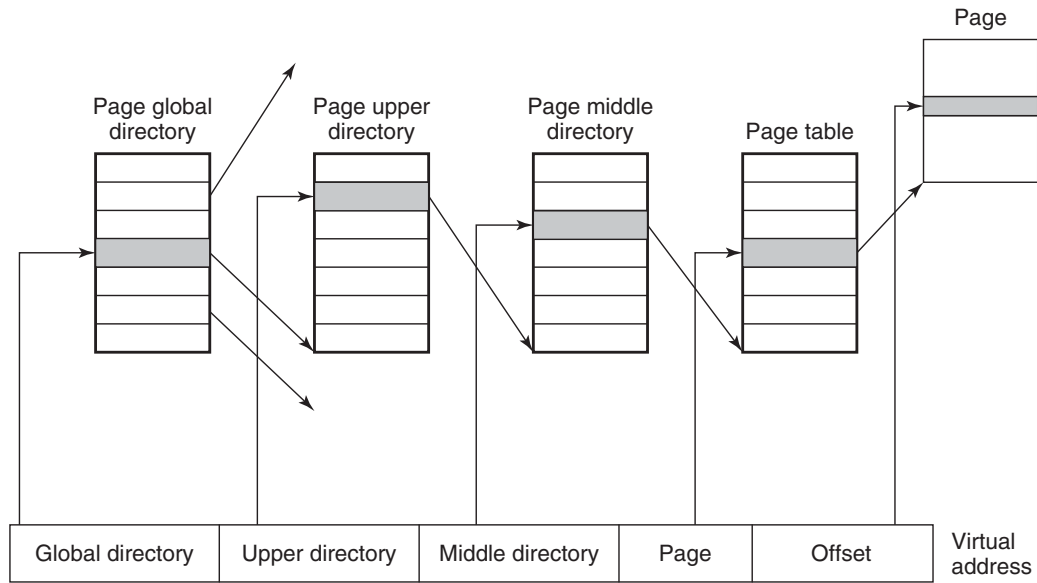


Figure 3.1: Four-level paging in Linux (from [8, p. 761])

3.2 Memory in Linux

3.2.1 Virtual Memory

Modern operating systems like Linux use *virtual memory* instead of directly working with *physical memory* addresses. Each process has its own *virtual address space*. The addresses used by a process are virtual addresses that are translated into physical addresses on access. Memory is divided into chunks of a fixed size called *pages*. When an address is accessed, the *memory management unit* (MMU) translates the virtual address to a physical address by looking up the corresponding page number in the *page table*. The page table holds references to the mapped physical *page frames*. To avoid wasting memory for unused page table entries *multi-level paging* is used. Page tables are broken into multiple levels (for example, current Linux versions support up to five levels). Different parts of the virtual address are used to index the different levels of the page table (see Fig. 3.1).

It is also possible that a virtual page is not resident in RAM, for example, when pages are swapped to backing storage to free memory. In this case, a page fault is raised, and the fault handler of the operating system tries to load the page into memory.

3.2.2 Kernel Space and User Space

In Linux, the address space is divided between kernel space and user space (see Fig. 3.2). While user space virtual memory is different for each process, kernel space virtual memory is

shared between all processes (i.e. there is no change to the kernel space part of the virtual address space). This reduces the performance cost of system calls compared to approaches that isolate user virtual memory and kernel virtual memory more thoroughly, for example, by using different page tables for user processes and the kernel. In any case, user processes can be denied access to kernel virtual addresses by setting the protection bit of the page table entries.

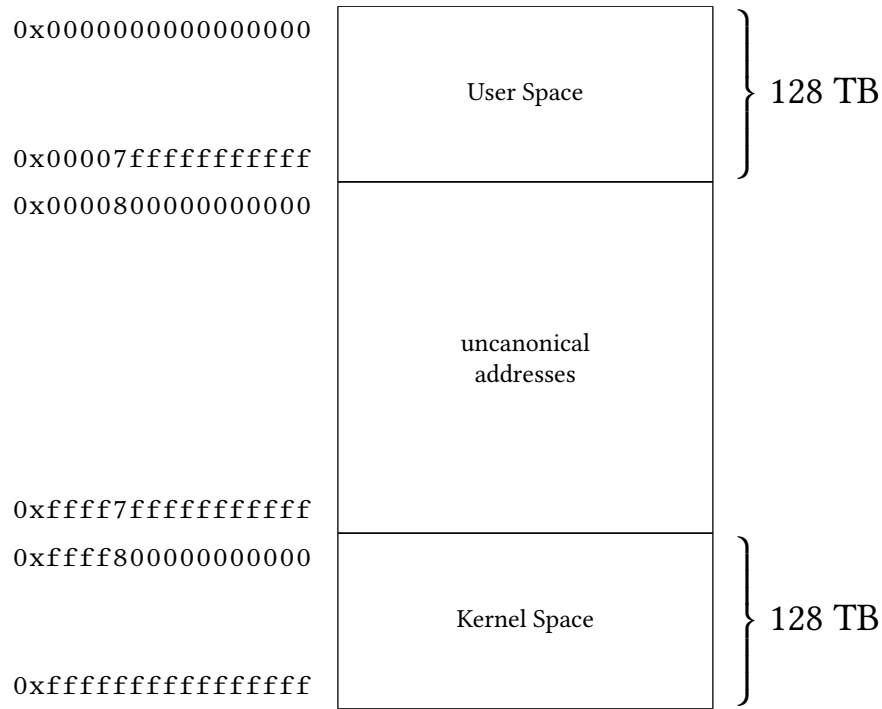


Figure 3.2: Kernel space and user space in x86-64 with four-level page tables (see https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt)

3.2.3 Shared Memory

The overhead involved in the copy approach explained above can be avoided by mapping the memory used for the scheduling plan data into both the address space of the user process and the kernel virtual address space. This requires only one system call invocation to set up the shared mapping. Subsequently, both the user process and the scheduler can operate on the same memory copy of the scheduling plan data. In the following sections, we describe how a new system call for this purpose can be added to the kernel.


```
SYSCALL_DEFINE3(sched_plan, unsigned long, length, struct plan_data __user **, plan_data,
                size_t, size)
{
    ...
}
```

Listing 3.1: System call definition

```
asmlinkage long sys_sched_plan(unsigned long length, struct plan_data __user **plan_data,
                               size_t, size);
```

Listing 3.2: System call interface

3.3 Adding a New System Call

How a system call is invoked is architecture-dependent. On x86-64 systems, this is typically done by using the `syscall` assembly instruction. Before invoking a system call, the system call arguments are loaded into the appropriate registers. For x86-64, this means that the system call number uses the `rax` register while the other up to six arguments use the `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` registers. Linux system calls and system call numbers vary across architectures. For example, the `read` system call has the system call number 0 in x86-64 systems, while 63 is used in arm64 systems.

When a system call is invoked, the appropriate handler function is called. This is done by looking up the handler function in the system call table using the system call number as an index. The system call table contains function pointers for all available system calls.

System call handler functions in Linux are defined using a `SYSCALL_DEFINE` macro (see Listing 3.1). In addition, a function prototype is added to `include/linux/syscalls.h` (see Listing 3.2).

3.3.1 System Call Interface

The system call makes the scheduling plan data available to the calling process by mapping the memory pages allocated by the kernel to user space. The user address of the shared scheduling plan data is written to the address of a pointer passed as the second parameter to the system call (`struct plan_data **plan_data`). The additional system call parameters are the minimum size of the shared scheduling plan data `unsigned long length` and the size of the scheduling plan data structure `size_t size`. The return value of the system call is the size of the mapped memory in bytes or a negative value in the case of an error. Because there is no dedicated C library wrapper, the system call is invoked by calling the `syscall` function with the appropriate system call number (see Listing 3.3).

```
long syscall(__NR_sched_plan, unsigned long length,
            struct plan_data **plan_data, size_t size);
```

Listing 3.3: User interface of the system call

```
struct plan_data {
    __u64 c_entry;
    __u64 l_entry;
    struct plan_entry entries[];
};
```

Listing 3.4: `struct plan_data`

3.3.2 Shared Scheduling Plan Data

The shared scheduling plan data is represented by an instance of `struct plan_data` (see Listing 3.4). This includes the plan entries represented by a flexible array of `struct plan_entry`, the number of total plan entries (`l_entry`) and the number of finished plan entries (`c_entry`).

The user process can change the entries of the scheduling plan by writing to the `struct plan_entry` array, add entries to the scheduling plan by changing the number of total elements in the array, and reset the plan by setting `l_entry` and `c_entry` to zero.

3.4 Kernel Memory Allocation

In the kernel C library functions like `malloc` cannot be used. Instead, kernel code that allocates memory uses functions like `kmalloc` and `__get_free_pages` provided by the memory management subsystem. At the most basic level these functions always use the *page allocator* to allocate physical memory pages (for the following see [8, ch. 10]).

3.4.1 The Page Allocator

The page allocator can only allocate a number of pages that is an order of two. This is done using the *buddy algorithm*: For example, to fulfill an allocation for eight pages in a system with 64 pages (Fig. 3.4a), the memory is first split into two partitions of 32 pages (Fig. 3.4b). Next, one of these pieces is further split (Fig. 3.4c) and so on until the size of the allocation request is reached (Fig. 3.4d).

While this allows for fast memory allocations, the restriction to 2^{order} pages can lead to a considerable amount of *internal fragmentation*, i.e. memory is allocated but not used. In order to avoid this, the kernel uses the *slab allocator* on top of the page allocation system (see Fig. 3.3).

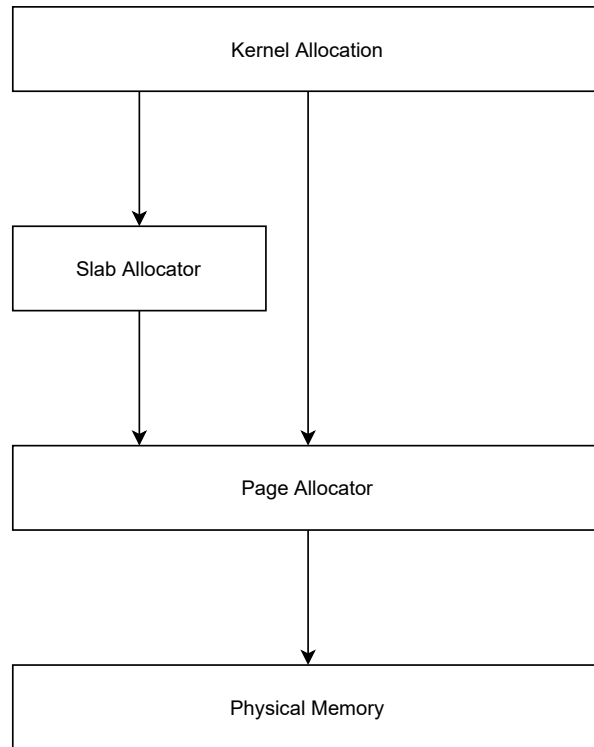


Figure 3.3: Memory allocation in the Linux kernel

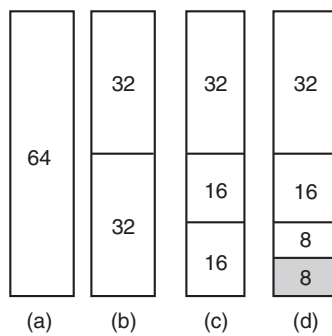


Figure 3.4: Operation of the buddy algorithm (modified from [8, p. 762])

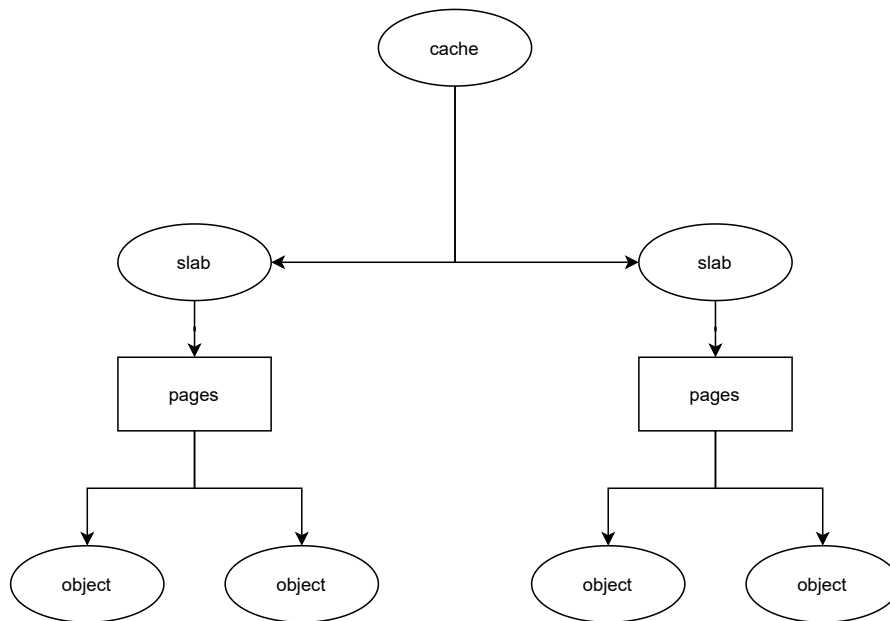


Figure 3.5: Caches, slabs and objects (modified from [9, ch. 8])

3.4.2 The Slab Allocator

The slab allocator allocates pages for frequently used data objects. For example, the virtual memory system uses the `struct vm_area_struct` structure to manage virtual address space ranges of tasks. Instead of directly allocating pages each time a new instance of `struct vm_area_struct` is needed, they are taken out of an *object cache*. Each cache consists of multiple *slabs* (see Fig. 3.5), i.e. memory pages that are used exclusively to hold objects of one type. When the kernel now allocates a new `struct vm_area_struct`, the slab allocator looks through the slabs of the `struct vm_area_struct` cache (`vm_area_cachep`) and returns an unused slot. If all slabs are full, the slab allocator allocates a new slab by allocating new pages using the page allocator. When an object allocated via the slab allocator is no longer needed, the slot in the slab is marked as free.

For generic allocations the `kmalloc` function is used. `kmalloc` is built on top of the slab allocator by using caches for generic objects of a fixed size. For example, when 14 bytes are allocated, `kmalloc` looks through the `kmalloc` caches for the cache with the next bigger object size (e.g. 16 bytes). The allocation is then treated like other slab allocations.

3.4.3 Page Allocation for Shared Scheduling Plan Data

Because the pages allocated for the scheduling plan data are shared between the kernel and a user process, they must not hold any other kernel data. This can best be achieved by allocating pages directly.

```
void init_pb_rq(struct pb_rq *pb_rq)
{
    ...
    pb_rq->plan = (struct plan_data *) get_zeroed_page(GFP_KERNEL);
    ...
}
```

Listing 3.5: `init_pb_rq`

```
struct mm_struct {
    struct vm_area_struct *mmap;           /* list of VMAs */
    ...
    unsigned long (*get_unmapped_area) (struct file *filp, unsigned long addr,
                                         unsigned long len, unsigned long pgoff, unsigned long flags);
    ...
};
```

Listing 3.6: `struct mm_struct`

The prototype allocates a single page in `init_pb_rq` (see Listing 3.5). This function is called when the scheduler is initialized by `sched_init`¹.

3.5 Creating a New Mapping in User Space

In Linux the virtual memory of a user process is represented by an instance of `struct mm_struct` (see Listing 3.6). The address space is split into *virtual memory areas* (*vma*) represented by instances of `struct vm_area_struct` (see Listing 3.7).

The address range of a *vma* is defined by the `vm_start` and `vm_end` fields. The properties of a virtual address area are defined by the `vm_flags` field. For example, if the `VM_WRITE` flag is not set the *vma* is write protected.

```
struct vm_area_struct {
    unsigned long vm_start;
    unsigned long vm_end;
    ...
    struct mm_struct *vm_mm;
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    ...
};
```

Listing 3.7: `struct vm_area_struct`

¹`kernel/sched/core.c`

3 Implementation

```
down_write(&mm->mmap_sem);
...
u_addr = get_unmapped_area(NULL, 0, npages * PAGE_SIZE, 0, vm_flags);
...
vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
...
vma->vm_start = u_addr;
vma->vm_end = u_addr + npages * PAGE_SIZE;
...
vm_iomap_memory(vma, virt_to_phys(pb_rq->plan), npages * PAGE_SIZE);
...
insert_vm_struct(mm, vma);
...
up_write(&mm->mmap_sem);
```

Listing 3.8: Mapping kernel memory to user space

The system call creates a new `vma` for an unused virtual address range and a new mapping to scheduling plan data (for the following, see Listing 3.8). When working with virtual memory areas, the `mmap_sem` semaphore of the current process' `mm_struct` has to be held to protect against concurrent changes to the address space. To get an unused virtual address range, the `get_unmapped_area` function of the `mm_struct` is called.

To create a new virtual memory are, the system call allocates memory by using `kmem_cache_zalloc`. The `vma` is then initialized with the address range returned by `get_unmapped_area` and inserted it into the `mm_struct` via `insert_vm_struct`. The scheduling plan data is mapped to the allocated virtual address area by the function `vm_iomap_memory`. This creates the required page table entries in the page table of the `mm_struct`. In order to make sure that the mapping is not treated as a copy-on-write mapping, the `VM_SHARED` flag is set.

4 Conclusion and Open Questions

4.1 Conclusion

The objective of this thesis was to make scheduling plans in user space available to the Linux scheduler. The shared memory approach achieves this with minimal overhead by reducing the amount of system call invocations, resulting in fewer switches between user space and kernel space, as well as avoiding memory copy operations. By allocating exclusive memory pages for shared scheduling plan data, it is ensured that no kernel data is exposed to the user process accidentally.

4.2 Open Questions

4.2.1 Dynamic Allocation of Memory

Currently, only one page is allocated for the scheduling plan. In practice, scheduling plans will be much bigger, and their size may change with time so that more pages will have to be allocated dynamically. This could be done by invoking the system call with the current user address of the scheduling plan data as the second parameter. The system call then allocates a number of pages required to fulfill the request and maps them to user space.

4.2.2 Error Handling

As noted in [7], the prototype PB scheduler does not support error handling. Because the tasks executed by the PB scheduler are simulated using a proxy kernel thread, the execution times specified in the scheduling plan are always accurate. In practice, the execution times of scheduling plan entries are based on predictions and will often prove to be flawed. The PB scheduler will have to decide what to do when a task does not meet its deadline.

In some cases, the PB scheduler will want to report deviations from the scheduling plan to the HPC system. Currently, the communication works only one way: the user tells the scheduler that the scheduling plan has changed by adding scheduling plan entries and changing the scheduling plan size. To report deviations from the scheduling plan, the scheduler could write the actual execution time of a scheduling plan entry to the shared memory, for example, using an extra field in the scheduling plan entry structure (`struct plan_entry`) for this purpose.

4.2.3 Synchronization

The scheduler and the user process work on the same data, so synchronization may be necessary. Because the scheduler is not preempted, write operations by the scheduler are always atomic. In contrast, the user may begin writing and be preempted before finishing. For example, when the user begins changing a scheduling plan entry by changing the `exec_time` but is preempted before the `uall_time` is changed, the plan entry data is incongruous. If the scheduler then executes this plan entry before the user process has finished the write process, the scheduler works with invalid data. This could be remedied by locking the shared scheduling plan data at the cost of suspending the execution of the planned tasks.

References

- [1] R. Ahmadov, C. Gerbig, R. Kretschmer, *et al.*, “Comparing high resolution WRF-VPRM simulations and two global CO₂ transport models with coastal tower measurements of CO₂”, *Biogeosciences*, vol. 6, no. 5, pp. 807–817, 2009. DOI: 10.5194/bg-6-807-2009.
- [2] National Center for Atmospheric Research, *Cheyenne*, <https://www2.cisl.ucar.edu/resources/computational-systems/cheyenne>. (visited on 09/28/2021).
- [3] M. Hovestadt, O. Kao, A. Keller, *et al.*, “Scheduling in HPC resource management systems: Queuing vs. planning”, in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Springer, 2003, pp. 1–20. DOI: 10.1007/10968987_1.
- [4] L.-O. Burchard, M. Hovestadt, O. Kao, *et al.*, “The virtual resource manager: An architecture for SLA-aware resource management”, in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, IEEE, 2004, pp. 126–133. DOI: 10.1109/CCGrid.2004.1336558.
- [5] Paderborn Center for Parallel Computing, *OpenCCS*, <https://www.openccs.eu/core/>. (visited on 09/28/2021).
- [6] E. Strohmaier, J. Dongarra, H. Simon, *et al.*, *Top500*, <https://www.top500.org/>. (visited on 09/28/2021).
- [7] K. Glaß, “Plan based thread scheduling on HPC nodes”, M.S. thesis, Freie Universität Berlin, 2018. [Online]. Available: https://www.inf.fu-berlin.de/inst/ag-se/theses/Glass18_PBScheduling.pdf.
- [8] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015, ISBN: 978-0-13-359162-0.
- [9] M. Gorman, *Understanding the Linux Virtual Memory Manager*. 2007. [Online]. Available: <https://www.kernel.org/doc/gorman/html/understand/>.

List of Figures

1.1	WRF simulation	1
1.2	SeisSol simulation	2
1.3	WRF scaling	3
1.4	Cheyenne supercomputer	4
2.1	Scheduling class hierarchy without the PB class	5
2.2	Scheduling class hierarchy with the PB class	6
2.3	State transition diagram of the PB scheduler	7
3.1	Four-level paging in Linux	10
3.2	Kernel space and user space	11
3.3	Kernel memory allocation	14
3.4	Buddy algorithm	14
3.5	Caches, slabs and objects	15

Listings

2.1	struct plan_entry	7
2.2	struct pb_rq	8
3.1	System call definition	12
3.2	System call interface	12
3.3	User interface of the system call	13
3.4	struct plan_data	13
3.5	init_pb_rq	16
3.6	struct mm_struct	16
3.7	struct vm_area_struct	16
3.8	Mapping kernel memory to user space	17