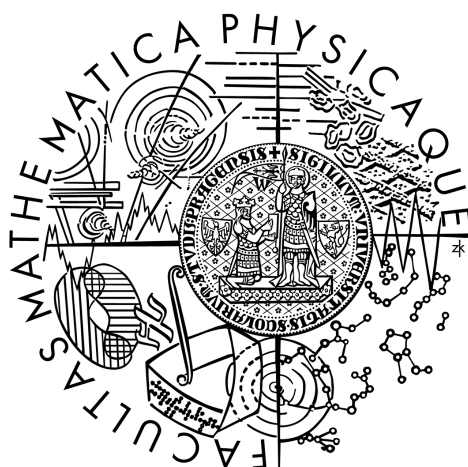
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Michal Hocko

Tuning Virtual Memory for Performance

Department of Software Engineering
Supervisor: RNDr. Tomáš Kalibera, Ph. D.
Study Program: Informatics, Software systems

In the first place, I would like to thank RNDr. Tomáš Kalibera, Ph. D. for the great help, many ideas, and support all the time I was working on this thesis. Many thanks go also to all others who helped me, especially Pavel Šanda for the code reviews. Mgr. Pavel Milar, Pavel Ondroušek and Světlana Ondroušková for text and grammar reviews.

I am also grateful to Itka and my family for big support during my studies.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne ....

Michal Hocko

Table of Contents

1 Context: Impact of Memory Usage on Software Performance.................................9
1.1 CPU Cache Behavior...10
1.2 Physical Memory Layout...11
2 Definition of the Problem 13
3 Goals for the Solution..15
4 Structure of the Thesis...17
5 Problem Analysis...19
Alternative approaches..22
6 Architecture of the Solution...23
6.1 Allocation Layer..24
6.2 Mapping Layers...24
6.3 User Interface Layer..25
7 Solution Overview...27
7.1 Physical Memory Allocator..27
7.2 Memory Layout Manipulation...28
7.2.1 Existing Mapping Manipulation...29
7.2.2 On-Demand Mapping Manipulation...30
7.3 File User Interface...31
8 Implementation..35
8.1 Linux Kernel Internals..35
8.1.1 Memory Management Description..35
Physical Memory...36
Physical Memory Allocator...36
Virtual Memory...38
Process Address Space...39
Page Fault Handling..40
8.1.2 Proc File System..42
Process Specific Proc File Entry...43
seq_file Interface...43
8.1.3 Linux Kernel Modules...44
8.1.4 Linux Kernel Configuration System..45
8.2 Kernel Implementation...46
8.2.1 Strategy Infrastructure..46
Data Structures..46
API for Modules...48
Kernel Usage API 48
8.2.2 Strategy Allocator...49
Generic Strategy Allocation...49
Exact Strategy..51
Modulo Strategy..52
Page Coloring...53
Bin Hopping..55
8.2.3 Hint Page Fault Handling..56
Hint Fault Table...56
Strategy Page Fault Handler..58
8.2.4 Memory Layout Remapping..61
8.2.5 Proc User Interface...62
Read Operation..63
IOCTL Implementation 65
8.3 User Space Library Implementation...67

9 User Guide....69
9.1 Patch Set Installation....69
Configuration....69
Compilation....72
9.2 User Space Library Installation and Usage.... 73
9.3 VMWare Image....73
10 Evaluation....75
10.1 Limitations of the Solution....75
10.2 Related Work....77
10.3 Use Case Application....78
FFT Description....78
Testing Environment....79
Test Cases....79
Test Cases Evaluation....80
11 Conclusion....87
List of Abbreviations....89
Literature....91

Název práce: Tuning Virtual Memory for Performance

Autor: Michal Hocko

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Tomáš Kalibera, Ph. D.

e-mail vedoucího: tomas.kalibera@dsrg.mff.cuni.cz

Abstrakt: Většina dnešních operačních systémů implementuje virtuální paměť a pro aplikační (uživatelskou) vrstvu poskytuje pouze pohled na virtuální vrstvu. Je známo, že výkonnost některých aplikací je výrazně ovlivněna aktuálním použitím fyzické paměti, které je určeno momentálním virtuálním a fyzickým mapováním. Mapování je pod plnou kontrolou operačního systému a není deterministické, což vede k sub-optimalitě a nedeterminismu ve výkonu aplikace.

Cílem práce je umožnit ladění výkonu aplikace daného mapováním, a to jak z režimu jádra, tak i z uživatelského režimu. Modifikovali jsme aktuální jádro operačního systému Linux a připravili tak rozhraní, které aplikaci umožní měnit rozložení paměti na základě strategií implementovaných jako moduly jádra. Naprogramovali jsme strategie optimalizované pro CPU cache a ukázali jsme, že tento přístup může pomoci v řešení jak sub-optimality, tak i nedeterminismu ve výkonu některých aplikací.

Klíčová slova: správa virtuální paměti, nedeterminismus ve výkonnosti, Linux

Title: Tuning Virtual Memory for Performance

Author: Michal Hocko

Department: Department of Software Engineering

Supervisor: RNDr. Tomáš Kalibera, Ph. D.

Supervisor's e-mail address: tomas.kalibera@dsrg.mff.cuni.cz

Abstract: Most of the current operating systems implement virtual memory management and provide only a virtual layer for the user land. It is known that the performance of some applications (especially memory intensive) is influenced by the current use of physical addresses specified by virtual memory mapping performed by the operating system and is not fully deterministic. The problem results in both sub-optimal and non-deterministic performance.

This thesis focuses on the user space approach to virtual memory tuning for an application with special requirements. The Linux kernel was modified to provide a simple interface for the user space, which enables a process specific physical memory layout manipulation on strategies implemented as kernel modules. We have implemented CPU cache sensitive strategies and shown that this can improve both the optimality and determinism of performance for some applications.

Keywords: virtual memory management, non-determinism in performance, Linux

1 Context: Impact of Memory Usage on Software Performance

Hardware performance has increased rapidly in the last decade; CPU speed and operating frequency has grown to GHz, the primary memory has enlarged to GiB (gibibyte) and the secondary storage capacity has reached TiB (tebibyte) values. On the other hand, it is very difficult for the memory to follow the speed-up of processors.

The difference in speed causes that even the performance of a very fast CPU is degraded when it needs to wait for data coming from memory. Taking a look at the standard memory model used for current computers [13], it categorizes the memory according to its speed, distance from CPU and size (see Figure 1.1).

Starting from CPU – at the top level of the hierarchy, there are CPU general *registers*. The *registers,* placed directly on CPU*,* provide the fastest but also the smallest memory[1].

Going down in the hierarchy, on the second stage there are *CPU caches* usually divided into 2 levels. *L1 (Level 1) cache* placed directly on CPU and *L2 (Level 2) cache* placed very near to the CPU. *L1 cache* is as fast as registers and is bigger (typically from 4kiB to 64kiB). *L2 cache* is slower but usually much larger (hundreds of kiB to MiB).

The main memory is situated bellow the cache. This is the general-purpose, relatively low-cost *primary memory* (mostly MiB to GiB). Typically, it is RAM or a similar memory technology. Data are usually not preserved when the computer is turned off.

At the bottom, there may be a *secondary storage* layer. This involves hard drives, CD-ROM, DVD and many others. Common property is relatively low price (compared to the capacity) but also low speed when compared to the primary memory speed. Compared to primary memory data are preserved when machine is turned off.

---

1 They are hardwired directly on the CPU, so it is impossible to change them in the future. Cost per byte is very high. On i386 architecture there are 8 general registers and some specialized (FPU, MMX etc.).

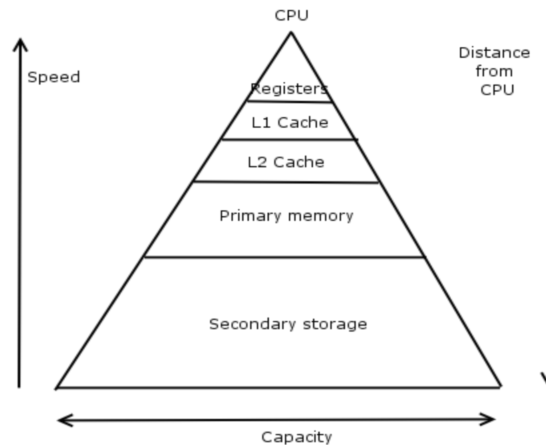


Figure 1.1: Memory hierarchy with speed/capacity relation.

The purpose of the whole memory hierarchy is to allow a reasonably fast access to a large amount of memory at the discretion of the memory management to keep the most often used data as close to CPU as possible. Whenever CPU tries to get data it will request L1 cache. If the data are present, they are immediately returned to CPU. Otherwise it is necessary to get the data to *L1 cache* from *L2 Cache*. Analogously, *L2 cache* gets the data from *Primary storage* and finally *Secondary storage*. The worst case is when the data needs to be read from the secondary storage may take a lot of time. Even fetching data from the primary memory may rapidly slow down the performance. Therefore the CPU cache usage is significant.

1.1 CPU Cache Behavior

CPU cache is a small associative memory. Data are stored in *cache lines* with the usual size of 16B. We distinguish a *full and n-way associative* cache [10].

The full associative can place data to any cache line, hence its effective capacity usage. Although this is highly effective, it is also rather an expensive solution. The cache controller is very complex and it grows as it increases in size.

The n-way associative approach provides a compromise between price and efficiency. It splits cache lines to sets of lines where each *set* is a full associative cache for n lines. CPU selects an appropriate set according to some part of the address.

According to the used address type, a cache is either *real-indexed* (for physical

addresses) or *virtual-indexed* (for virtual addresses). Both have some pros and cons. CPU has to translate the virtual address before storing into or reading data from a *real-indexed* cache; however, the context switch will not cause a problem because the physical address space is not changed. On the other hand, the virtual-indexed cache suffers from context switches when all data in the cache must be flushed even if used by another process (for example, the shared data). Therefore L1 cache is usually virtual-indexed and L2 cache is real-indexed.

When CPU tries to get data which are not present in the cache a *cache miss* occurs. This means that the data must be fetched from a lower memory layer. When the data are ready they need to be stored in the cache. The cache controller determines the line (or set of lines) and if there is no to place, it needs to replace some lines. This may be rather complex, because the cache line may be written (*dirty*) and thus it needs to be flushed to a lower memory layer, which is also time-consuming. The best solution would be to evict non-dirty line which will not be accessed in the near future.

1.2 Physical Memory Layout

Memory, from the user point of view, is represented by the continuous linear virtual memory. The operating system is responsible for virtual to physical address mapping. We will call this mapping *Physical Memory layout*.

As discussed previously in this section, performance is highly dependent on the effective usage of CPU caches. The L2 cache is typically implemented as real-indexed. Thus, the physical memory layout represented by virtual to physical memory mapping/translation, has also a big influence on the effectivity of cache usage and as a result on the whole application performance. Such influence has also been reported reported in [16][22].

2 Definition of the Problem

As described in the previous section, the effective usage of CPU caches is very important for the whole performance.

We are focusing on *real-indexed* caches because they are used for almost all current L2 cache implementations. This means that the effective cache usage is highly dependent on the code and data placement in the physical memory.

The application performance is then subject to the current physical memory layout. This can lead to performance *non-determinism* and *sub-optimality* caused by different virtual to physical memory mappings for separate runs of the same application (new mappings can produce different cache misses, thus different performance).

Most operating systems keep full control over the application physical memory layout and will not expose it out of the kernel. However, the kernel cannot assume the application memory usage and prepare optimal mappings for it. Many operating systems use random mappings: Linux uses the first available physical page frame [11, Chapter 6], some use the cache sensitive allocation heuristics, such as page coloring or bin hopping [16].

However, most operating systems do not provide an interface for applications to enable them to participate in the allocation process or to give kernel hints for the creation of memory mappings. Some applications, especially those that are memory intensive, could highly benefit from a similar interface.

3 Goals for the Solution

This thesis aims at providing a user space interface for the communication with the operating system kernel to enable a safe physical memory layout manipulation in a virtual memory based environment. It includes the possibility to change the existing mappings as well as to provide information on how the mapping should be created on a per process base.

According to the conclusions drawn from the standard memory model behavior and its influence on performance *non-determinism* and *sub-optimality*, we assume that a more versatile interface with a kernel implementation can lead to both better optimality and determinism.

We modify the operating system kernel to support extensible strategies for memory layout creation as well as enable remapping of the current layout for a specific process. This functionality is exposed to the user space by a well defined interface.

An application can use this interface directly or it can modify the layout of different processes[2]. Therefore the target application does neither need to be aware of, nor changed for, a better or more deterministic performance.

As the target operating system, we have chosen open-source and freely distributable *Linux kernel [27]*.

Some basic strategies will be implemented to show that their usage helps a use case application for both a better performance optimality and determinism when compared to the unchanged kernel. Fast Fourier transformation benchmark [14] is used for the use case application, because it is known to have a highly non-deterministic performance behavior on Linux systems caused by a non-deterministic memory layout [14].

As a result, an application can use the extended memory management functionality to examine the performance improvements in a concrete physical memory layout with different mapping strategies. An application can also implement its own strategy optimized for its particular requirements.

2 With the emphasis on security. For example, just a privileged process is allowed to manipulate a different process.

4 Structure of the Thesis

The following text is structured into several parts. To begin with, the *5 Problem analysis* is explored and possible ways how to extend the Linux kernel memory manager to support virtual memory tuning are discussed. Next, the *6 Architecture of the Solution* chapter describes high-level description of the used multi-layer architecture.

The *7 Solution Overview* chapter covers the design and interface of each layer separately as well as the basic data structures used for the implementation and integration to the Linux kernel.

The *8 Implementation* chapter follows and is split into three parts where the first one, *8.1 Linux Kernel Internals,* provides information required for understanding the kernel part. The second part, *8.2. Kernel Implementation*, focuses on the kernel modifications required for the physical memory layout manipulation. This includes the implementation details about the internal and external API as well as the provided strategies. The last part, *8.3. User Space Library Implementation*, briefly discusses an additional user space library provided for an easy usage of the kernel API from the user space.

9 The User Guide chapter provides information on the prototype implementation, its configuration, installation and demo application.

10 The Evaluation chapter presents results from the use case application which uses the created advanced memory layout manipulation. Finally, a comparison between the chosen approach and the existing solutions is discussed.

5 Problem Analysis

The physical as well as virtual memory layouts are maintained by the kernel of the operating system. The virtual memory is created when an application requires the memory by memory allocation or mapping backing storage to the memory (e. g. file mapping, dynamic library loading, etc.). While the physical mappings are created with the *on-demand* approach where the physical memory is assigned to the virtual when it is really used (read or written).

The general purpose operating system cannot assume, as already stated, the pattern of concrete application memory usage, therefore cannot optimize mappings for it. However, it can use a *strategy* and decide which physical address to use for the given virtual one.

Some operating systems use a random strategy where the first available free physical memory is used to satisfy mapping creation (e. g. Linux). This approach is very fast because no special logic is required; however, it leads to the most non-deterministic mappings.

On the other hand, cache sensitive strategies were also developed [16]. *Page coloring* and *bin hopping* are the most used ones. Both exploit the locality of reference [9] memory access patterns and provide physical memory mappings which do not produce cache misses, either consecutive in memory location or time of reference. As a result, cache misses occur less frequently, thus the performance is better.

Certain applications, especially those intensively using memory (databases or scientific calculations), could highly benefit from the interface which enables them to change or control their memory layout. Note that this interface does not change the virtual layer and enables just a safe manipulation with mappings. An application must not have full access to the memory because of security. The only concern of such an interface would be to inform kernel which mapping is the best and the rest is done by the kernel for the application transparently.

The memory layout manipulation can lead to more optimal performance for a concrete application as well as the stabilization of the random initial state caused by different layouts between the separate runs.

There are several ways how to deal with the memory layout manipulation from the user space point of view. Let us focus on 3 basic approaches:

- Enable to modify the virtual layout
- Enable to control the physical memory allocation
- Enable to give hints for the physical memory allocation and change the current virtual to physical memory mapping.

The first, in fact, is already supported to a certain extent by the most modern operating systems. UNIX-like systems support the `mmap` system call which maps a range of file backed storage to the virtual address space and process may indicate where to put the mapped area. However, this is only a hint (do not confuse it with hints used for the strategy based physical memory allocation discussed later) and the operating system can choose a different location (see the `mmap` manual pages).

There is also a possibility to advise about how this mapped area will be accessed using the `madvise` system call (defined by POSIX). If an area is to be accessed sequentially, mappings can be created with read-ahead because they will be used shortly after. Note that the file backed mapped memory can prevent from multiple separate reads from the storage and read one big block for the area. On the contrary, the area with random access would spend a lot of time by read-ahead for mappings which would not be used at all.

Although this can be very helpful, especially for storage access time optimization, it still hides details about memory mappings thus gives only a limited number of possibilities for the cache sensitive memory usage. E. g. the preallocated physical memory could produce a lot of cache misses caused by mappings and the advantage of block read from storage optimization would be suppressed by the poor CPU performance.

The second approach requires the application to have access to the low level physical memory allocator. This can be seen in some *micro-kernel* architectures [25], but it is very hard to achieve in the *monolithic-kernel* architectures where the memory management is usually a hardwired sub-system of the kernel space. There are also some non trivial security issues which need to be considered for the user space manipulation with the memory [12][19].

The last one makes a compromise between the previous two approaches. The

physical memory allocation is kept in the kernel space with no direct access from the user land. However, it needs to enable hints the for memory management from the user space and support different allocation strategies which work with the data from hints.

The *physical memory layout* is therefore exposed to the user space in a restricted way so that an application can observe the current layout or change it by a (trusted) kernel code based on strategies using hints provided by the application. Applications with no special requirements can use the standard allocation strategy and memory sensitive one can give hints for a special strategy which works well for it. A hint is a simple piece of information from an application intended for a strategy to understand the requirement.

Strategies implemented in the kernel need to be extensible and support (adding) new algorithms without extensive modifications of the kernel. E. g. the Solaris kernel support multiple algorithms [20] which can be changed in runtime with a user space tool. However, these algorithms are hard-wired and new one would require to touch the kernel.

Many operating systems support loadable modules which provide a simple way to add new functionality to the kernel without any modification to the system core. Strategy interface should cooperate with modules subsystem therefore enable convenient way to support new algorithms.

It seems that the last approach with modular strategies could provide the most scalable and extensible way to deal with the special memory requirements driven from the user space.

Strategies and memory layout manipulation can be intended not only for performance optimizations but also for performance reproducibility. This can be used especially for regression benchmarking when the memory state of the measured application can be stored and a test can be rerun with the same memory conditions in the future. This situation requires a strategy which provides exact physical memory allocation for the mapping. An application layout can be stored during run and later used for a new run.

Alternative approaches

Besides the memory layout manipulation, there are also other approaches to improve performance by optimizing cache use. Let us mention at least the compiler-based techniques and data structures fills and paddings very briefly.

The first mentioned falls broadly into two categories [4]: those that attempt to *reduce* and *tolerate* cache misses. The reduce techniques include loop transformations to enhance data locality [24]. To tolerate memory latencies caused by cache misses, the compiler can insert prefetch instructions to move data into the cache before it is needed, thus overlapping memory access with computation [5].

Data structures *fills* and *paddings* are the commonly used techniques for cache conflicts elimination. Some parts of the data structures are used almost all the time together and therefore they should fit different cache lines. E. g. multiple locks used for fields serialization should not be in one cache line when used together because both locking and unlocking would cause unwanted misses. This is usually solved by fills, with cache aligned size, added to a structure among the together used data so that they start on different cache lines.

With padding, compiler, loader or memory allocator offsets data starting location to avoid cache misses. As an example, Linux kernel Slab allocator [11, Chapter 8], which provides memory for the kernel data structures, uses cache coloring for the allocated objects. The memory pool (continuous physical memory area) for object allocations is not used the whole but objects start from the cache size block aligned addresses.

6 Architecture of the Solution

In this thesis, we have addressed the defined problem by providing a new file based user space interface for the process based memory layout manipulation and a new concept of strategy based physical memory allocation for the *Linux* kernel. This approach requires a change of several subsystems, hence the use of *4-layer architecture*.

- Allocator layer
- On demand mapping layer
- Process layout manipulation layer
- File user interface layer

Each of them focuses on a certain functionality with a well defined interface.

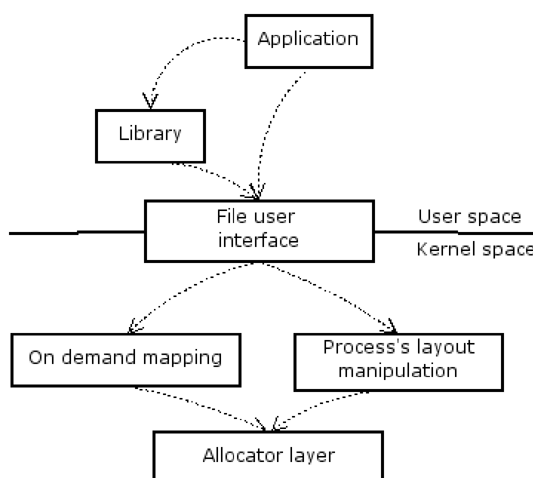


Figure 6.1: 4-Layer architecture of the kernel modification. Arrows denote dependencies.

In short, the allocator layer is responsible for providing physical memory according to the given requirements. The on-demand page mapping layer enables to control on-demand layout creation when mappings are created only if really needed. On the other hand, the direct manipulation of the existing mappings is provided by the process layout manipulation layer. Finally, the user space interface is represented by a process specialized file with a standard file system operation interface. All layers are implemented as kernel parts.

For an enhanced usage of user land applications, a user space library was also created, which wraps the technical details about the interface provided by the kernel and defines the abstraction for all supported features. This is not included in the

architecture as a separate layer because it is not necessary for the functionality.

The chosen architecture was used in consideration to both an independent maintenance of separate functionality and a good granularity for the configuration.

6.1 Allocation Layer

Physical memory allocation is a critical kernel component, because it provides physical memory to higher kernel subsystems, basically the memory layout management.

The design of the implemented allocator is based on the currently used *buddy allocator* [11, Chapter 6]. Moreover, it was extended with the support for *strategies.* A s*trategy* is an abstraction for selecting physical memory (from the currently not used) according to the requirement called a *strategy hint*. A *hint* contains strategy specific data which are used during memory allocation.

The allocator gets a *hint* from a higher layer and identifies the strategy to be used. Then the *strategy* specific code is used for the final allocation. Note that this cooperation model enables modular implementation of strategies; moreover, it enables their adding and removing without modification of the allocator.

6.2 Mapping Layers

Memory layout manipulation and control require handling of two basic situations:

- *On-demand* mapping creation
- *Existing* mapping modification.

The first one implies that the standard memory *on-demand* policy is taken over. Hints for the physical memory allocator need to be stored and prepared for the time when a particular memory should be used. Thus we have used per process *hint tables* and provided an interface for table manipulation. A table entry is identified by a virtual address and keeps the associated *hint*.

When an on-demand mapping request occurs, the table is searched and the hint is used for the low level strategy allocator. No hint means standard handling (the original on-demand mapping).

The existing mapping modification requires changing the physical memory currently assigned to the virtual transparently for a process[3]. The *Process layout manipulation* layer defines a hint based interface. A modification request holds the virtual memory and the hint to be used for the new physical memory for mapping. Remapping implementation uses a hint for the strategy allocator and sets new mapping for the returned memory.

6.3 User Interface Layer

At the top of the architecture hierarchy, there is a user space interface. Normally, user space applications are not allowed to enter the kernel. There is only a strictly defined system call interface. Whenever an application needs to operate on a resource protected by the kernel, it needs to use the dedicated predefined system calls. The kernel performs the system call operation and returns the result to the user space.

Nevertheless, a new system call implementation is generally not recommended because of the involved technical difficulties and also possible security issues for the whole system. Moreover, system call parameters are very low level and it is hard to design them to be well scalable and extensible. Therefore a new system call is usually added the to mainline kernel after very long discussion, when no other reasonable variants exists.

On the other hand, Linux (and some other UNIX-like systems) provides the *proc* file system [21], which is the standard way to communicate with the kernel through normal file system operations.

Thus, in the implementation, the *proc* file system approach with a special file for each running process is used. The main responsibility of the layer is to translate file operations to concrete requests for lower layers as well as to fill the required data and return the value of the operation back to the user space.

3 The process accesses only to the virtual layer of the memory and thus there is no problem to change the underlaying physical memory. However, implementation must guarantee that the process can use the memory during remapping.

7 Solution Overview

This chapter covers the detailed design and interface of each layer specified by the architecture described above. Implementation details are covered in the next chapter. Some Linux specific parts and data structures will be discussed also in this chapter. They are briefly described in this part, and further information can be found in *8.1. Linux Kernel Internals part*.

7.1 Physical Memory Allocator

Linux splits the physical memory to chunks of the same size called *page frames* (one page has typically the size of 4 kiB). The kernel keeps track of each available page frame (see *8.1.1 Memory Management Description*), especially of those which are not currently used (*free page frame*).

The physical memory allocator (called *Buddy* allocator [11, Chapter 6]) is responsible for free pages maintenance and for providing them to other kernel subsystems. The current implementation does not enable the hints usage and is based on the hardwired *first fit* allocation strategy. Nevertheless, it behaves well for the common case.

In this thesis, the allocator data structures are preserved and implementation is unchanged; the non-intrusive *strategy allocator* with a similar interface working upon the same data structures is provided. Therefore both allocators can be used without any problems or regressions, at least as long as the data structures do not change.

A strategy is represented by a set of callbacks which are used during the allocation process and they are associated with an identifier called the *strategy id*. Callbacks define the behavior and logic of a strategy.

All supported strategies are stored in a global list which is maintained by the strategies infrastructure API. Therefore each strategy needs to be registered before it is used. This approach with the Linux kernel modules support enables modular implementation of strategies and thus runtime modules with strategies.

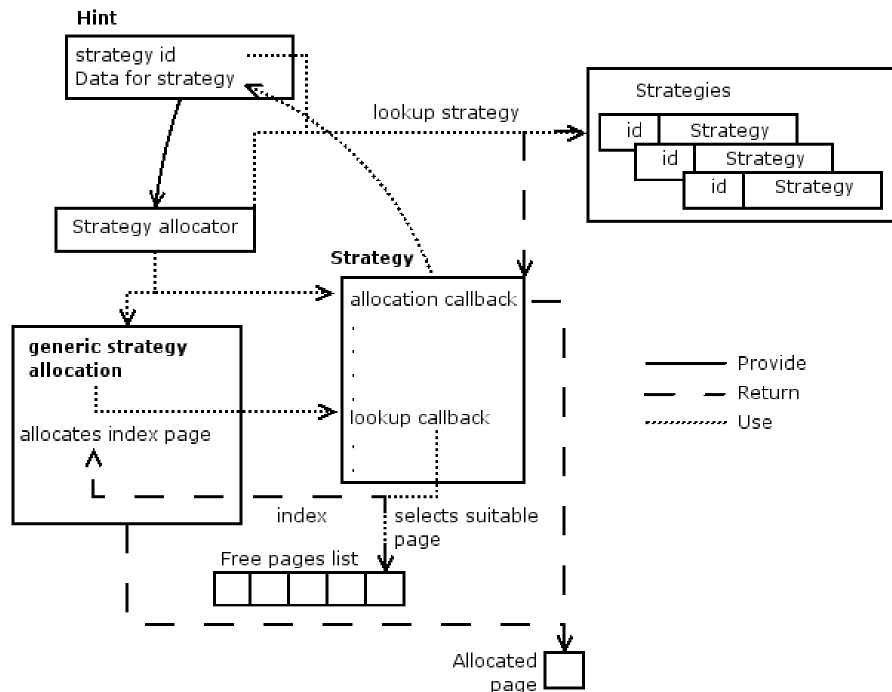


Figure 7.1: Strategy allocator infrastructure.

Strategies implementation should be as simple as possible, therefore the *generic strategy allocation* was implemented. It gets a hint and handles the free pages list in the same way as the original allocator but it uses the callbacks defined by the hint strategy when selecting a page frame from the free list. Consequently, the strategy implementation can focus just on the higher level logic and do not need to work with the low level data structures and use the low level generic allocation for that purpose.

The strategy allocator, when compared to the original one, introduces an additional hint and strategy based allocation concept. A hint is a simple data structure with a strategy id and data for the strategy. An allocation request contains all parameters given to the standard allocator, but also an additional hint structure. A particular strategy for allocation is searched according to the id stored in the hint. If there is a strategy which can handle the given hint, its allocation callback (if it is provided) or generic allocator code is used. An allocated page frame is returned (see Figure 7.4).

7.2 Memory Layout Manipulation

The process memory layout is described by page tables in the Linux kernel. The virtual address space consists of pages of the same size, which are mapped to the

physical page frames. The mapping is described by a *page table entry*. A new entry is created when a new virtual page is allocated, mapped (with the `mmap` system call), the stack is enlarged or COW (Copy On Write) page allocated.

A page table entry keeps the current state of mapping, a state specific value and other information. The new entry is usually created as a *not present* without an associated physical page frame and an association is created with the *on-demand* policy.

7.2.1 Existing Mapping Manipulation

The existing mappings (with a table entry) can be changed by modification of the page table entry to refer to a strategy allocated page frame. However, this is not enough, because there are several types of pages which need some additional steps:

- *Not present* pages do not have a physical frame associated yet. A new page frame according the given hint is associated.
- *Shared* pages where one physical page frame is mapped to multiple pages (from one and/or multiple processes). All such page table entries refer to the same page frame. The remapping code needs to find and remap all of them.
- *Cached* pages, which are used for better low level data sharing, are stored in the cache related data structures. A newly allocated (hint) page must replace the current.
- *Swapped* out pages need to swap-in data from backing storage before remapping.
- *I/O* pages are currently under data transfer either from or to a backing storage. Remapping must wait until the transfer finishes.
- *Locked* pages are under mutually exclusive process and waiting is necessary.
- *Kernel* pages represent the memory held by the kernel for its data structures. These pages cannot be remapped at all because the kernel uses direct mapping to the physical memory (see *8.1.1 Memory Management Description*).

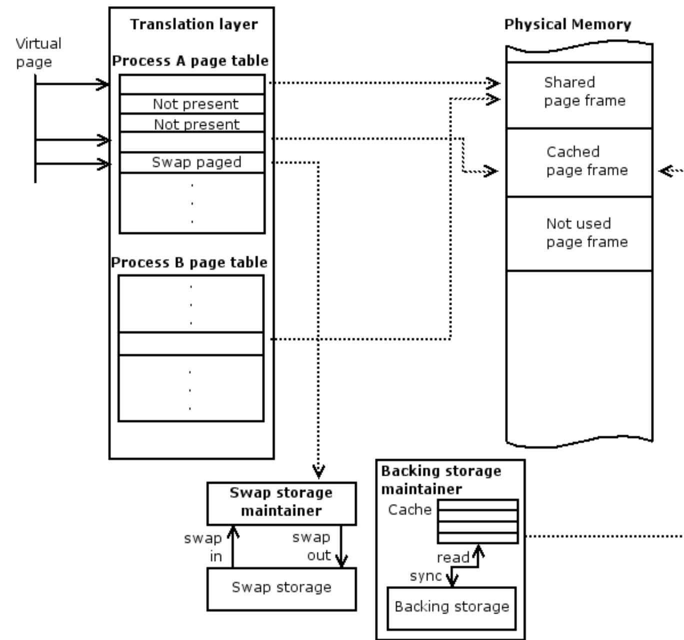


Figure 7.2: Cases for existing page table entries.

7.2.2 On-Demand Mapping Manipulation

As already mentioned, *on-demand* mappings are handled at the moment memory is needed. We have addressed that by a per process specific *hint table*. The table is represented by a *red-black tree* data structure [23] which offers the best possible worst-case guarantees for the insertion time, deletion time, and search time. Hints are stored in the tree nodes according virtual addresses for them. The table is stored in the process memory descriptor structure (*8.1.1 Memory Management Description*).

Hints are grouped in *hint clusters* with a configurable number of *slots* (*9.1. Patch Set Installation*). Each cluster is associated with a cluster size aligned page address[4] and it keeps track of the slot usage. Nodes are sorted according the cluster address.

4 If each cluster starts with the cluster size aligned virtual page address (`cluster_vaddr%cluster_size=0`), we will get nodes which do not overlap their clusters in the tree. Slot searching for a specific virtual page (`vaddr`) address within the cluster is a constant time operation and `slot_idx = (vaddr - cluster_vaddr)/PAGE_SIZE`

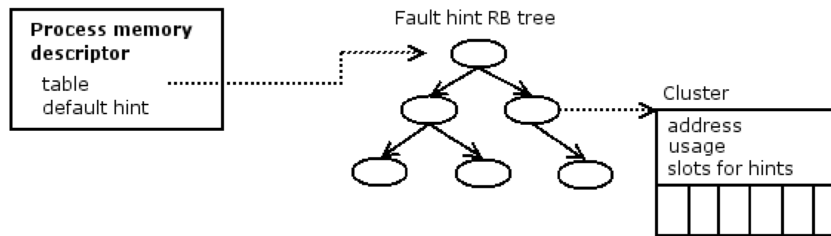


Figure 7.3: Fault hint table and hint cluster nodes.

The cluster usage decreases the number of nodes stored in a tree, hence the speed up search operation. On the other hand, it requires more memory for tree nodes (each node also includes memory for hints stored in slots) and just a few slots used from each node, which would be extremely inefficient. However, programs tend to show the locality of reference behavior for memory access [9]. This means that hints will be also provided for close virtual pages and thus it may fit to the same cluster.

A default hint was implemented to enable default behavior for the process page faulting. If no hint for the address is stored in the tree, the default is used (if it is specified). A process can use a certain strategy for all its mappings, simply by setting the default hint.

The page fault handler tries to search for a hint in the first place and use it if found. Otherwise, it falls back to the standard implementation.

The hint page fault handler needs to follow the standard handler behavior (*8.1.1 Memory Management Description*) but will rather use the strategy (with a hint) as a normal allocator. When a page is provided by the storage specific code hidden by a callback (e. g. the mapped memory), it is checked whether it fits the strategy. If not, it is changed by remapping the layer. In the end, a hint page frame is used for faulted mapping, otherwise the hint fault handler fails and falls back to the normal handler.

7.3 File User Interface

UNIX-like systems, and Linux as well, expose the versatile virtual file system (*VFS*) layer [3]. VFS defines interface for file and directory manipulation in an implementation independent way. This enables the file approach for manipulation and maintenance for many resources on different levels of abstraction (regular files for their content manipulation, special files to communicate with devices etc.).

The proc file system does not use a secondary storage and all requests to files and directories are transformed to operations upon in-kernel data structures. Device drivers use this interface very often when they need to get some input from, resp. provide output to user land [21].

Although the proc file system could exhibit the whole functionality declared by VFS to the user land, just a small subset of operations is usually implemented. Directories usually enable files (files may be regular, devices or symbolic links) listing and lookup but not adding or removing entries. Files support reading and writing in most cases.

Our approach is based on a state-full process specific file (`physmaps`) with read, seek and IOCTL (Input-Output ConTroL) operations. Therefore file `open`, `close`, `read`, `lseek` and `ioctl` callbacks are implemented for the kernel proc file system layer. A file is present for each existing process in the `/proc/<pid>/` directory, where all process specific files and directories are located.

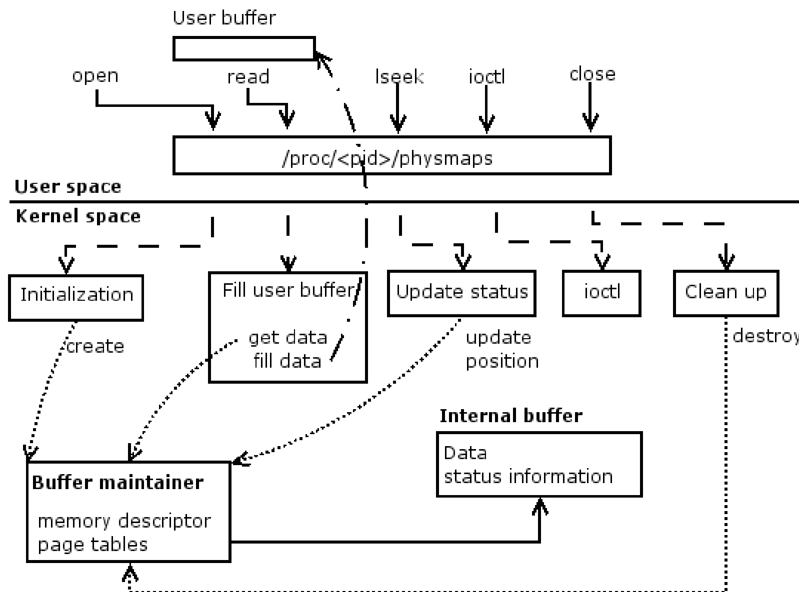


Figure 7.4: Proc file communication.

The proc file manipulation starts with the standard *open* function on the `/proc/<pid>/physmaps` file which is translated into the internal data structures initialization. As a result, *VFS* provides file handle to the user space.

The standard *read* function returns with process's current memory layout in the provided buffer which have the following format:

```
vma:vm_start vm_end:count
"(pfn|N|F|S) "{count}
```

where

`vma` stands for a virtual memory area (*8.1.1 Memory Management Description*), which represents a continuous virtual memory area with the same purpose and permissions. `vm_start` stands for a hexadecimal virtual address of the first page of the VMA. On the other hand, `vm_end` represents the first page immediately after the VMA. Finally, `count` keeps the number of pages inside the area.

The following line lists comma separated mappings for all pages from the area. Four values are possible:

- `pfn` – a hexadecimal value with a page frame number means that mapping exists and the page is associated with the page frame with such an address[5].
- `N` – no physical memory is associated with the virtual page.
- `F` – represents not present non-linear file mapping (see the *man remap_file_pages* for more information).
- `S` – stands for mapping with the swapped out content.

These 2 lines are returned for all memory areas.

The kernel uses an internal buffer maintainer for each opened file (see Figure 7.4). The read operation is transformed into data copying from the internal to caller buffer (given as a read parameter). If the current content of the internal buffer is not sufficient for the user space request, it is filled from kernel data structures (for VMA information) and page tables (for mappings) by the buffer maintainer.

The seek operation moves the internal position of a file and the internal buffer so that the next read request starts from a different location.

The read operation does not provide any possibility to change mappings or any other layout manipulation operations. The write operation could enable that, but the semantics for the user data format can be very complex. To enable a more precise interface, we have used the *ioctl* file operation [8, Chapter 6]. It is intended for special controlling of files outside the usual read/write concept. The ioctl call consist of a file identifier, command and specific data. The command identifies the operation and it is

5 More precisely, page frames are addressed with physical addresses. However, the kernel does not need to store the whole address to keep track of page frames, because they have the same size and starts from the well defined physical space start (almost always from 0 but note that some architectures enable holes in the physical memory). Therefore the order of a page frame is enough for its constant time location. This order is called the *pfn* – page frame number.

represented by a numeric value. The ioctl callback calls the appropriate handler according to the command value.

We have implemented commands for

- Memory mapping – `IOCTL_GET_PFN`, `IOCTL_SET_PFN`
- Memory layout – `IOCTL_FILL_AREA`, `IOCTL_GET_VMA`, `IOCTL_GET_VMA_COUNT`
- Fault hints – `IOCTL_GET_HINT`, `IOCTL_SET_HINT`, `IOCTL_REMOVE_HINT`, `IOCTL_GET_DEFAULT_HINT`, `IOCTL_SET_DEFAULT_HINT`

The first set of commands handles both getting current mapping for virtual page and setting its mapping (according to the hint given in the data parameter).

The memory layout commands family provides an API for getting (not changing) the layout information. It defines *vma* as a memory region mapped to the process address space and *area* as its subset.

Finally, the last group of commands controls on-demand page fault hints. It enables to add, remove and check the current hints as well as to control the default fault hint.

Although ioctl is a very extensible concept for file controlling, it does not provide a consistent API for different types of operations (each can have a different data parameter). Its usage may be very complex and error prone. Therefore the `libvmtune` user space library was implemented to hide the ioctl usage and will rather provide a high level interface for applications (see *8.3. User Space Library Implementation*).

8 Implementation

The previous chapters have discussed the thesis approach from a more general point of view with the emphasis on the architecture and the used concepts. This chapter, on the other hand, focuses on the concrete implementation with layers API and data structures. The first part describes The Linux kernel memory, proc and module subsystems used for the implementation. The second part shows kernel part implementation and the last one the documents provided by the user space library.

8.1 Linux Kernel Internals

Linux is a rapidly evolving operating system kernel and we have focused on the current 2.6.18 version series. The current implementation does not allow any allocation strategies and also no possibility for memory layout manipulation for the user space is available.

This chapter discusses the basic internal information about the Linux kernel implementation, especially those parts used by the thesis. We will focus just on the concept and data structure relations description and do not go deep into the technical details.

8.1.1 Memory Management Description

Linux, like most modern operating systems, provides the virtual memory management. In this chapter, we will provide the basic information about the Linux memory management architecture[6], basic data structures and behavior. See [11] or [8, Chapter 15] for more information.

Linux kernel keeps full control over the physical memory and exhibits only a virtual view. Thus each application can have a linear point of view on memory and theoretically use up to the whole addressable memory[7]. In the standard configuration (32b x86 architecture), the virtual address space is split into the *user* and *kernel* portion where the kernel is mapped to the user's top memory area). The kernel part

6 Note that the Linux kernel is being in rapid development at the moment and the structures may change in future. We are discussing the current 2.6.18 version.

7 4GiB on 32b architectures are addressable. However, more memory can be available with certain extensions like PAE (physical address extension) on the Intel architecture, where up to 64GiB can be addressed.

uses direct mapping to physical memory, where the physical address can be calculated directly from the virtual (the first kernel virtual page is mapped to the first physical page frame). All kernel related data structures must fit this memory.

The virtual memory is composed of fixed sized chunks called pages (4kiB usually), while the physical memory consists of the same sized page frames (usually 4kiB). The main responsibility of the memory management is to provide mappings from virtual pages to physical page frames.

Physical Memory

When talking about the physical memory layout[8], we distinguish *dma*, *low* and *high* memory zones. Each of them keeps pages for a certain purpose:

- *dma* – the first 16MiB required by some ISA (Industry Standard Architecture) devices for DMA (Direct Memory Access) transfer
- *low* (also called normal) – 16MiB up to 896MiB – kernel memory (See [11, Section 4.1] for more information)
- *high* – 896MiB up to RAM available (note that this zone may be empty if not enough RAM is available). The kernel can access this memory only by temporary mapping its portions to the kernel space.

A zone is described by the `struct zone` structure (see `include/linux/mmzone.h` for more information) which maintains free page frames, page usage statistics, locks, data for swap daemon and others.

Each page frame is described by the kernel *`struct page`* (see *`include/linux/mm.h`* for more information) structure which holds the frame status information like flags, number of mapping for frame, reference count, address space, lock, etc. (see the current sources because this structure is under permanent development). This structure helps the kernel to keep track of the whole memory, even for that which it cannot access directly (high memory).

Physical Memory Allocator

Physical page frames are provided to the kernel by a *binary Buddy allocator.* This is an allocation scheme that combines a normal power-of-two allocator with free

8 Note that we are only considering uniform memory architectures (UMA) which have just one memory layout. Non-uniform architectures (NUMA) have multiple memory nodes which have their zones. The Linux code is well designed to support both by emulating UMA to have just one statically allocated node, therefore much code can be shared between them. The NUMA specific code needs to consider only the inter node logic.

buffer coalescing. The physical memory is broken up into continuous blocks with the power of 2 number of pages. The base of the power is called the *order*. Blocks with the same order are linked together. A zone, as a free memory information keeper, holds an array of order lists (see Figure 8.1a).

The page allocator provides an interface for order sized allocations and uses the *first fit strategy* (If no block with the same size is available, it uses the first larger block). If the found block is larger, it is split to two halves (*buddies*), the first is used and the other is inserted into the list with smaller order (see Figure 8.1b). This is repeated until the chosen block reaches the required order. When the page frame is freed, it is returned back to the order list and all its members are checked whether they form a buddy with it. If yes, it coalesces them to the block with a higher order and puts them to the appropriate list. This is repeated while there is something to

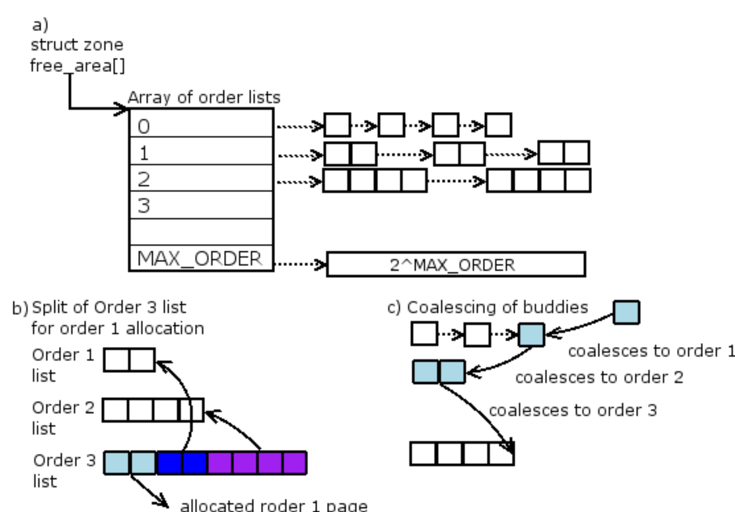


Figure 8.1: Buddy allocator
a) Scheme of Buddy allocator pages organization
b) order 1 allocation with split of higher order page
c) coalescing of order 1 page up to order 3 list

coalesce (see Figure 8.1c).

A buddy allocator is very fast and well organized also for higher order allocations[9]. However, the *first fit strategy* results in quite random memory mappings after the system has been running longer. The performance for a separate application run can be almost random (see *10.3. Use Case Application*) as a result.

A free list is a shared structure and therefore it has to be protected from simultaneous access. Page allocation is often an operation, thus contention of lock

9 This is rare situation required by kernel. In almost all cases 0 order page are allocated.

can be bottle neck (bottle-necked) for the performance on SMP (Symmetric Multi-Processor) systems. Therefore the Linux kernel optimizes most often the 0 order page frame allocation case. Rather than allocating from a buddy allocator, it uses the CPU local list of already allocated pages. When the list is emptied (or reaches a low water mark), a bunch of 0 order pages is allocated at once. Also when a page is freed, it is returned to the CPU list and if the high water mark is reached, pages are released to the buddy allocator. Allocation from the CPU set do not need any locking therefore the allocation fits better a SMP.

Virtual Memory

The whole system (the user and kernel space) uses virtual addresses. When a CPU has to read real data it needs a translation from a virtual to physical address.

Linux maintains a generic platform independent code for page table handling. It uses 4 level tables and if the architecture uses a different model, it emulates a generic interface. For example, i386s use 2 level page tables where the 2 layers have just the size of 1 and point back to a higher level. This is quite tricky because the code will only be optimized during compilation and does not exist in the runtime mode.

The first layer is called the PGD (Page Global Directory). Its entry points to a page frame with an array of PMD (Page Middle Directory) entries. In the same way, a PMD entry points to the PUD (Page Upper Directory used only 64b architectures). Finally the lowest level PTE (Page Table Entry) keeps flags for a page frame and its physical address in the *PFN* (Page Frame Number – physical address/page size) form (see *Figure 8.2*). The flags are used to track the frame state (present, shared, write protected etc. See `include/linux/page-flags.h` for more information). The virtual address is split to 4 (3 if the PUD is not used) parts and those are used for the physical address location.

The kernel keeps and maintains page tables for itself and all processes (each process has a pointer to its PGD page frame).

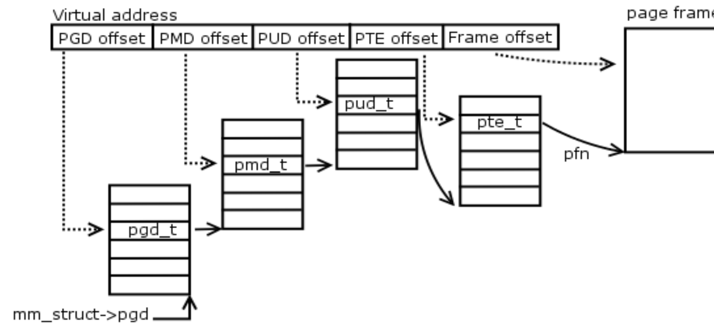


Figure 8.2: Virtual address translation from 4 level page tables.

Initially, when the processor needs to map a virtual address to physical, it must traverse a page table. To avoid this considerable overhead, architectures provide the TLB (Translation Look aside Buffer). Linux assumes that most architectures provide the TLB therefore provides a fine grained generic (platform independent) code for the manipulation with it (the architecture only provides empty implementation of architecture specific hooks). The L1 Cache manipulation API is available too and works very similarly.

Process Address Space

From the user perspective, the address space is flat and linear. As mentioned above, it is split into the user and kernel portion. The process address space is described by the high level `struct mm_struct` structure (see `include/linux/sched.h` for more information). One process has exactly one instance of this structure stored in the `struct task_struct` structure (see `include/linux/sched.h` for more information) which describes the whole process. Address spaces are not usually used whole but rather consist of several sparse memory regions which are called the VMA.

The VMA is described by the `struct vm_area_struct` structure (see `include/linux/mm.h` for more information) and represents a continuous virtual memory area with the same protection and purpose (let us say a stack, heap or `mmap`-ed memory). All regions are linked together, ordered by address and can be reached from `mm_struct`.

Very roughly, we could categorize the VMA into two big groups. The *anonymous* and *file/device backed* area. The first stands for the memory which does not use a

secondary storage (heap, stack, anonymous `mmap` etc). The other is a family of areas which maps the secondary storage or virtual file in general to the memory (memory mapped files by `mmap`, shared libraries, System V shared memory). For that purpose, the `vm_operations_struct` (see `include/linux/mm.h` for more information) structure is used which contains callbacks to handle the concrete situation as a `open`, `close` area or `nopage` callback used by the page fault handler. To enable access to the backing storage, an area uses a pointer to the file (see `struct file` in `include/linux/fs.h`) with its address space (see `struct address_space` in `include/linux/fs.h`).

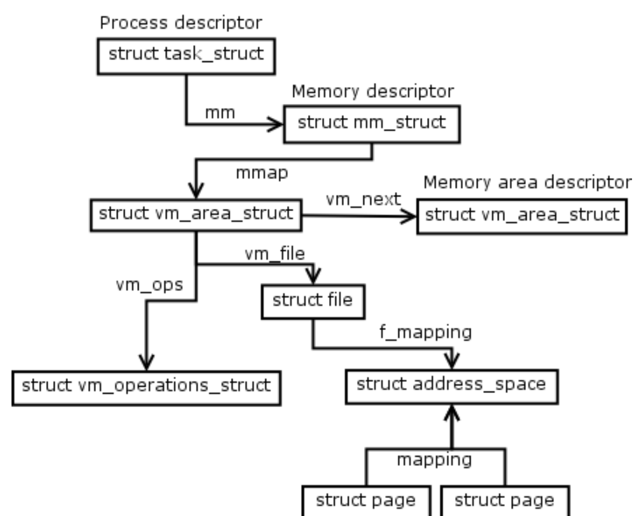


Figure 8.3: Memory related data structures.

This provides a good enough abstraction for all basic use cases as shared, private or COW mappings without any concern of how a concrete implementation works.

Page Fault Handling

Pages in the process linear address space are not necessarily resident in the memory (no mapping to page frame exists for them). Linux like most of the other modern operating system use *Demand fetching*. This means that a page frame is assigned to the virtual page only when hardware raises a page fault exception (no mapping exists for the required virtual address, or access is not allowed by the current mapping). The exception is trapped by the operating system which uses a page fault handler function to deal with it.

A low level of the page fault handler is closely architecture dependent, therefore each architecture registers its own function (usually do_page_fault - see arch/i386/mm/fault.c for more information about the i386 architecture), which performs the basic sanity checks (e. g. a fault is from a known virtual area with correct permissions, the access is under stack which is the reason why it has to be enlarged etc.) and prepares all data needed for handling (mm_struct, vm_area_struct and the access type for the faulted virtual address). Then it can call the generic handle_mm_fault function (see include/linux/mm.h for more information) which handles all fault cases.

In fact, handle_mm_fault is just an alias for __handle_mm_fault (see mm/memory.c) for the MMU (Memory Management Unit) architecture and it results in BUG[10] for the noMMU architecture. This function prepares a page table entry for the faulted virtual address and calls the PTE fault handler handle_pte_fault. This function is responsible for mapping the creation of PTE. If no problem occurs, the page table entry is set to point to the frame. There are several situations which may occur:

- No mapping for an anonymous address – a new page is allocated from the page frame allocator (see do_anonymous_page in mm/memory.c)
- No mapping for a backed address – uses the VMA nopage callback to get the page with the required data from the backing storage and correctly handles the shared or private cases (see do_no_page in mm/memory.c)
- No mapping for a non-linear file backed area – (see do_file_page in mm/memory.c)
- Page is swapped out – retrieves a page from the swap
- Page is mapped but the page table entry is set to the read only while the write access was required. This is either a bad access or COW case. The bad access is handled by a low level handler, thus only the second situation can be considered here. The Linux kernel implements the COW concept by sharing pages even for writable private mappings (e. g. when mmap is called with MAP_PRIVATE flags and PROT_WRITE|PROT_READ prot parameters, or the child process is created by *fork* and its address space is created from the parent). There is no problem when the data are read because the current table entry enables it and the data are effectively shared. When writing is required, a page fault occurs which is legal because the VMA allows writing. Nevertheless, the fault handler needs to change the table entry to be writable, too. Thus it has to make a copy to a new page frame and change mapping with regard to the write access

10 BUG is a special way how to stop kernel with some useful data printed before fall down. It is used in situations when unexpected situation has occurred.

enabled (see `do_wp_page` in `mm/memory.c`).

All these functions are a little more complicated, because they need to be reentrant and ready for simultaneous page faults (even on the same mappings). Performance and elimination of future faults are also considered.

The source of exception does not register the page fault process at all unless an error occurs. If the user accesses a bad virtual memory (not mapped or mapped with different permissions than required) or no physical memory is available for mapping, or the required mapping cannot be created because of permissions, the application is terminated.

8.1.2 Proc File System

Linux like all UNIX-like systems implements the *proc* file system which provides a simple file interface for the communication with the kernel. Reading from *proc* files is translated to a kernel request for writing data (it depends on the *proc* file what kind of data it uses) to the user space memory. Writing to the *proc* file is by contrast translated to a read request from the user space memory. It uses the data for a file specific purpose (e. g. to set a property, turn something on/off, etc.).

The Linux kernel also provides a process specific *proc* infrastructure which is located in the `/proc` directory and assigns one subdirectory to each existing process. A *pid* (we will use the `/proc/<pid>/` notation in the following text) directory contains files/directories with the process specific data.

See the example for running the `init` process:

```
$ ls /proc/1/
attr     cpuset   exe   mem         oom_adj    smaps  status
auxv     cwd      fd    mounts      oom_score  stat   task
cmdline  environ  maps  mountstats  root       statm  wchan
```

where

`cmdline` – contains command line parameters of the process
`environ` – contains all environment variables
`exe` – is a symbolic link to the process executable
`fd` – is a directory with open file descriptors
etc.

Inside the kernel, there is an easy to use interface for the *proc* manipulation for

both the process specific (see `fs/proc/base.c` for more information) and general proc file [21].

Process Specific Proc File Entry

Although the general and process specific proc file background is principally the same, there are some differences. They have a common concept of VFS file operations. A significant difference is the file entry lifetime. Whereas the general *proc* file entry exists all the time when it is registered, the process specific file entry exists only when a process exists. It has to be created and destroyed transparently, and needs to have access to the process specific data (`struct task_struct`).

All process specific parts of implementation are handled by the generic code. A new process specific file (or directory) requires a compile time registration to the `tid_base_stuff` array (see `fs/proc/base.c`) where the file name and access permissions are specified. Each file is identified by the `pid_directory_inos` enumerator. Finally, the generic `proc_pid_ent_lookup` (called when a file is resolved from the `/proc/<pid>/` directory) function needs to be modified (in the big case statement) to supply a pointer to `struct file_operations` with VFS callbacks for file (directory) operations.

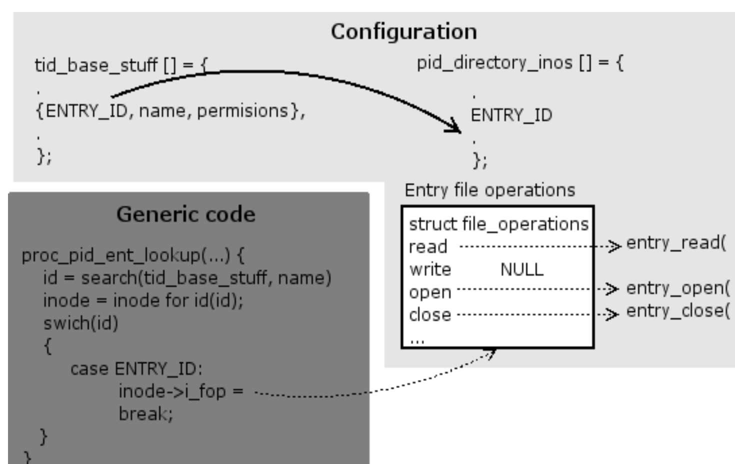


Figure 8.4: Process specific proc infrastructure.

seq_file Interface

As already mentioned, the standard manipulation with a proc file, and the process specific as well, means copying data between the kernel data structures and user space buffer. The Linux kernel provides a generic concept of `seq_file` [6]

which abstracts the memory data structures usage with a simplified file interface for the proc read-only manipulation.

`seq_file` API (see `include/linux/seq_file.h` for more information) supplies a safe initialization, connection to the proc `file_operation` structure and functions for the internal buffer content manipulation (in `printf` like manner).

The user (a kernel programmer) provides the `struct seq_operations` structure with callbacks which define the iteration and data logic. The initialized operation structure is assigned to the proc file structure (with the `seq_open` function) and the generic seq_file API can be then used to connect with the `file_operations` for proc entry (`seq_read`, `seq_lseek` etc.). User defined callbacks are called on demand to satisfy a specific file operation request.

8.1.3 Linux Kernel Modules

A kernel module represents a compiled code which can be added to and removed from the kernel in the runtime without the need to reboot. It provides an additional functionality which can be used immediately after the module is loaded. Modules are (un)loaded by a special kernel subsystem and the user space tools are provided for a simple manipulation (the `modutils` package with the `modprobe` tool for example)

The kernel module operates in the kernel space[11] and is restricted to the functionality exposed by the kernel (only exported symbols can be used). Unlike normal programs, it cannot link to the user space libraries. Deeper information about modules implementation can be found in [8, Chapter 2].

Basically, the module creator must provide the initialization (constructor) and exit (destructor) functions which are called when the module is loaded, resp. unloaded. These functions are responsible for the functionality registration and clean up.

Many kernel subsystems (file system, networking, device manipulation etc.)[12] define a kind of interface which is exported for modules. A module usually defines

11 This can be very dangerous because the kernel space is implicitly trusted and any error can lead to the crash of the whole system.

12 A typical example can be the file system implementation which registers callbacks for the VFS and thus a new file system can be used from the user space.

implementations for callbacks defined by the subsystem with a registration interface during initialization. On the other hand, the clean up function removes (unregisters) callbacks from the subsystem and deallocates all internal data structures. The rest is strongly dependent on the used subsystem and module implementation.

8.1.4 Linux Kernel Configuration System

The Linux build process consists of 2 stages. The first is responsible for configuration and the second for compilation. The configuration part determines which parts of the kernel should be compiled directly to a kernel image or as modules and which should not be used at all because the target computer does not support (or contain) a piece of hardware, or service is not required. The later uses configuration results and compiles specific parts to the result kernel image and its modules.

The configuration part can be performed by the `make config`, `make menuconfig` or `make xconfig` command from the kernel source tree (`make config` provides a simple console question and answer interface. `make menuconfig` and `xconfig` provide a menu based interface in a text resp. graphical environment). During configuration, special `Kconfig` files are processed which contain the configuration logic. As a result, lists of object files for direct compilation and modules are prepared and a standard GNU `make` tool can be used for the compilation (said rather roughly).

The `Kconfig` syntax is very simple and defines entities representing menus, menu items with a value (`bool` for used not used; tristate for not used, build-in or module; an integral value or string for a constant) or a list of values. Each entity can define its dependencies (on other options), help information default values and more (conditional definitions etc.). All configured items are then exported to the `include/linux/autoconf.h` as macro definitions with the `CONFIG_` prefix, and therefore can be used by all kernel codes for a further usage (like `ifdef` etc.).

The complete description and manual for the `Kconfig` file creators can be found in the `Documentation/kbuild` directory in the kernel source tree.

8.2 Kernel Implementation

The Linux kernel is an open source project with a big community of contributing developers. It is based on the “patch” driven development model. This means that the source code is freely available and developers send patches[13] to sub-system and release maintainers. Successful patches are then merged to the mainline kernel.

Therefore the kernel part of this work is also distributed as a set of patches for the 2.6.18 version series - ore precisely called *vmtune patch set* currently in the 1.0 version. The patches modify several parts of the source tree

- `include/linux` – new header files with the `physmpas_` prefix were added and the `memory.h` file was updated to contain the strategy allocator definitions
- `mm` – new files with the `physmaps_` prefix, modules files for strategies were added and the `migrate.c`, `page_alloc.c`, `swap_state.c` and `memory.c` files were changed
- `fs/proc` – changed to support the *vmtune* proc interface implementation
- `vmtune` – a new directory for the patch set configuration
- `kernel` – the `fork.c` file changed to support the page fault hint table

See *9.1 Patch Set Installation* for more information how to apply the patch set and configure it for a concrete usage.

The added functionality is documented with the kernel standard documentation format which is very similar to *doxygen* and relative (see `Documentation/kernel-doc-nano-HOWTO.txt` for more information).

8.2.1 Strategy Infrastructure

The strategy API is declared in the `include/linux/physmaps_hint.h` file and implemented in the `mm/physmaps_strategy.c` file. The strategy infrastructure is described in 3 parts; the data structures in the first, then the API for modules in the second, and for the kernel use in the last part.

All available strategies are stored in a global array. Note that this array shouldn't be used directly. The API for modules and kernel usage should be used instead.

```
static const struct pfn_strategy * strategies[MAX_PFN_STRATEGIES]
```

Data Structures

Two basic data structures, defined in the `include/linux/physmaps_hint.h` file, are

13 Differences to released sources.

used.

- A hint structure which identifies a strategy to by used and data for it.

```
struct pfn_hint {
    pfn_strategy_t strategy;
    unsigned char flags;
    unsigned long value;
};
```

where:

`strategy` - *Identifier of a strategy to be used for this hint.*

`flags`, `value` - *Strategy specific fields.*

- A strategy structure holds callbacks and internal data for a strategy implementation.

```
struct pfn_strategy {
    const char * owner;
    const char * name;
    struct page * (*alloc_pages)(gfp_t gfp_mask, unsigned int
    order,
                struct zonelist *zonelist,
                unsigned long vaddr, struct pfn_hint * hint);
    void (*shrink)(void);
    int (*check_hint)(struct pfn_hint * hint);
    void (*init_hint)(unsigned long vaddr, struct pfn_hint * hint);
    unsigned long (*get_index)(struct page * page, int page_order,
                struct pfn_hint * hint);
    void (*get_strategy)(void);
    void (*put_strategy)(void);
    int (*in_use)(void);
};
```

where

`owner`, `name` - *Both fields are not mandatory and they are used just for messages printed to logs.*

`alloc_pages` – *The non-mandatory callback for the order (2^{order} continuous pages) strategy allocation. Strategies are expected to implement the high level logic upon the generic strategy implementation is discussed in the next chapter. However, it can get free page frames from a given zones list and allocate a page directly from it.*

`shrink` - *If this function is specified, it is called when a strategy is unregistered and it is responsible for clean up after* `alloc_pages` *– e. g. deallocate cached pages etc.*

`check_hint` – *The mandatory callback for the hint validity checking. Checks whether all hint fields have correct values.*

`init_hint` – *The non-mandatory function for the hint initialization. It is called before a hint is used and strategies which requires some additional logic or have requirements for hint fields should do initialization here.*

`get_index` – *The mandatory lookup callback which returns* `index` *(in the continuous physical memory area represented as an array) of a first page complying the given hint. The $\geq 2^{order}$ value is returned if not found.*

`get_strategy`, `put_strategy`, `in_use` – *The non-mandatory strategy usage callbacks - if defined,* `get_strategy` *is used before and* `put_strategy` *after the strategy*

allocation is used. The `in_use` function returns true if a strategy is in use and it cannot be unregistered.

API for Modules

The API for modules is defined (modules have to include) in the `include/linux/physmaps_hint.h` file and implemented in the `mm/physmaps_strategy.c` file.

```
int register_pfn_strategy(pfn_strategy_t strategy, struct
    pfn_strategy *descriptor)
```

If a module wants to provide a new strategy implementation it needs to register initialized `struct pfn_strategy` and an identifier for it.

```
int unregister_pfn_strategy(pfn_strategy_t strategy)
```

When the module is about to be unloaded, the exit function must do the clean up and unregister the strategy.

Kernel Usage API

The kernel code should not use a strategy structure or a strategy array directly, and should rather use wrapper macros (defined in the `include/linux/physmaps_hint.h` file) and functions (in the `mm/physmaps_strategy.c` file).

```
alloc_pfn_hint_pages(gfp_mask, order, zonelist, vaddr, hint)
```

Gets a strategy for a hint. No strategy results in the `NULL` return value. If also its `alloc_pages` is specified it is called. Otherwise the generic strategy is used.

```
shrink_pfn_hint_pages(hint)
check_pfn_hint(hint)
init_pfn_hint(vaddr, hint)
get_pfn_hint_index(page, page_order, hint)
get_pfn_strategy(hint)
put_pfn_strategy(hint)
in_use_pfn_strategy(hint)
```

Get strategy for given hint. If no strategy is found they will fail. Otherwise check the callback and call it if present.

```
void __init init_pfn_strategies(void)
```

Function used for the build-in strategies registration. It is called very early during the boot process when the kernel page tables and memory structures are initialized. It initializes the global `strategies` array and registers all build-in strategies. The `__init` modifier tells that function should be thrown away after the kernel is initialized and thus does not occupy the memory.

```
int register_buildin_pfn_strategy(void)
```

All build-in strategies should be registered in this function. Only the exact strategy is allocated at this moment. Note that the build-in strategy cannot be compiled as a module.

8.2.2 Strategy Allocator

The strategy allocator is represented by `alloc_pfn_pages` similarly like the standard `alloc_pages` with an additional hint parameter. This interface hides the NUMA and UMA specific configuration[14] which provides a `zone` lists for an allocation. Finally, the `alloc_pfn_hint_pages` macro is used(see Figure 8.5) which determines whether to use strategy specific or generic allocation.

There are also helper functions and macros for a simpler usage of concrete situations (the same like the standard allocator).

`alloc_pfn_page_vma(gfp_mask, vma, addr)`

Alias for `alloc_pfn_pages` with the 0 order (just one page is allocated).

`alloc_zeroed_user_pfn_highpage(vma, vaddr, pfn)`

Allocates a 0 order high memory page (preferably) which is zeroed. If architecture doesn't support automatic clearing (zeroing) of the memory, it is done manually.

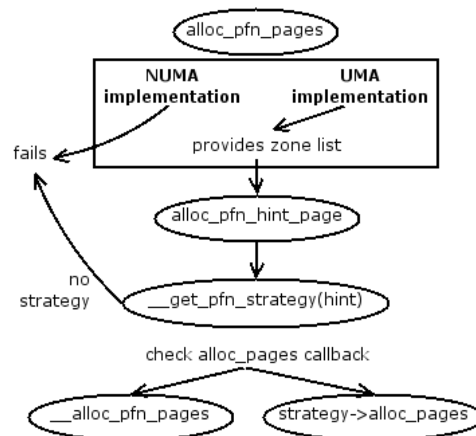


Figure 8.5: `alloc_pfn_pages` call graph.

Generic Strategy Allocation

Free page frames, which can be used for allocation, are stored either in the per CPU list or zone's free list (see *8.1.1 Memory Management Description*). We have implemented the generic strategy allocator represented by `__alloc_pfn_pages` which works similarly like the standard low level page frame allocator (`__alloc_pages`). It uses the same parameters with an additional `hint`.

14 Although the strategy allocation implementation can be used for both the UMA and NUMA architectures, a different preparation for it is required for them. The current strategy allocation API doesn't implement the NUMA configuration and the allocation always fail at the moment (it is planned to be supported in future).

Both the allocators share the common allocation parameters semantic and the cooperation with the swap daemon. To prevent from code duplications, the original code from `__alloc_pages` has been abstracted into the `__alloc_hint_pages` function with the additional hint parameter which is `NULL` for the standard allocator. The allocators are just aliases to this function, each with the correct hint parameter (see Figure 8.6).

This function is responsible for the `gfp_mask` (get a free page mask – see [11, Chapter 6.4]) semantic and the memory tight situation handling (see `mm/page_alloc.c` for more information). It calls the `get_page_from_free_list` function, which chooses a zone from the given zone list to be used for allocation. The chosen zone is then used either for the strategy generic (`buffered_pfn_rmqueue`) or standard (`buffered_rmqueue`) list allocator.

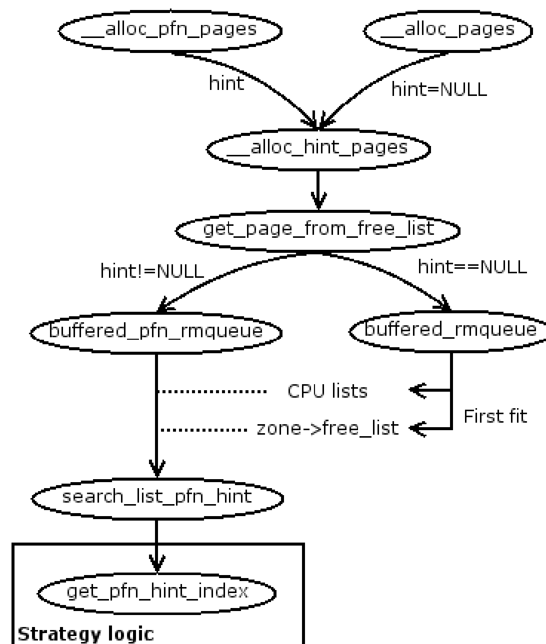


Figure 8.6: Generic strategy and standard allocation call graphs.

The generic list allocator searches the per CPU lists at the first place (if the 0 order allocation is required because the per CPU lists contain pre-allocated 0 order pages – see *8.1.1 Memory Management Description*) and if not successful, traverses and searches the zone's order free list.

The `search_list_pfn_hint` function is used for order list traversing. It uses

get_pfn_hint_index for each entry until the strategy appropriate page frame is found. The per CPU list case simply returns the found page frame, because it is already prepared. The zone's list needs additional steps to consolidate the list and the entry after the page frame was found. The entry is removed from the list and remaining page frames from the entry's continuous area are split to correct order zone's lists (see Figure 8.7).

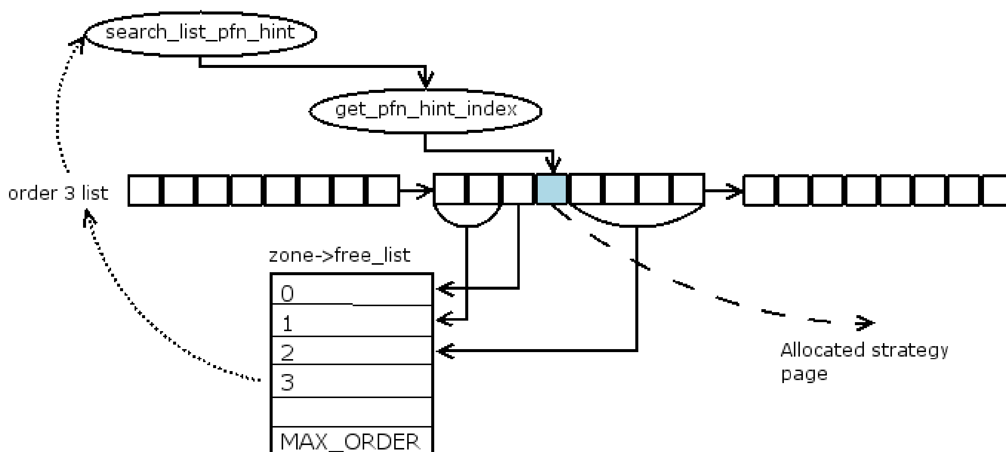


Figure 8.7: Generic strategy zone's order list allocation.

If a higher order strategy allocation is required, the per CPU lists are ignored and the zone's list with higher orders are searched. search_list_pfn_hint must additionally check whether the order page frame starting at the index (returned from get_pfn_hint_index) fits within the entry and the rest is the same.

Exact Strategy

An application which requires the same layout for separate runs may use the exact strategy which provides only a page frame with an exact address (see the implementation in the mm/physmaps_strate.c file). The physical address (in PFN format) is given in the hint's value field. The flags field is ignored and and the strategy has to contain the EXACT_STRATEGY constant defined in the include/linux/physmaps_ioctl.h file.

```
static struct pfn_strategy exact_strategy = {
     .owner = "BUILD_IN",
     .name = "EXACT_MATCH",
     .alloc_pages = NULL,
     .shrink = NULL,
```

```
    .check_hint = exact_check_hint,
    .init_hint = NULL,
    .get_index = exact_get_index,
    .get_strategy = NULL,
    .put_strategy = NULL,
    .in_use = NULL
};
```

where:

`exact_check_hint` - *Checking function only checks the given hint for* `NULL`.

`exact_get_index` - *Simply calculates the difference between the required pfn and the given page's pfn. If the result fit in the* 2^{page_order} *range, the generic allocator will use the* `idx` *page.*

Note that this strategy does not guarantee exactly same mappings, because the required page frame may be used at the moment

- Another process has mapped the page frame
- Kernel caches use the page for buffered data (e. g. for a block device) – this can be very often case because the kernel uses the cache especially for data from the secondary storage. The pages are reclaimed (returned back to free pages) when the memory is tight.
- Belongs to the kernel mapped memory – kernel data structures are mostly allocated by the `kmalloc` function [11, Section 8.4] and it uses the physical page allocator for low level allocations from a zone normal. If a computer doesn't have the high memory (more than 896 MiB RAM) or there are no more free pages in it, the zone normal is shared, therefore the user space fights with the kernel for the physical memory.

The first two cases could be solved by page stealing when a page would be unmapped in a process and reclaimed in a cache case. However, this implementation does not use this technique because it is rather intrusive and may produce some side effects on the process or the whole system account.

Modulo Strategy

Exactly the same layout for separate runs is rather a strict requirement as shown above. This strategy provides equivalent mappings where each equivalence class contains many pages thus there is a higher probability of an available complying page frame.

Both virtual pages and page frames are split to *equivalence classes*. Each class groups all pages and page frames with the same *address modulo base* results (with the base number of classes ranging from 0 to the base - 1).

The strategy is implemented as a kernel module in the `mm/modulo_strategy.c` file. It uses the hint with the `MODULO_STRATEGY` (defined in the `include/linux/modulo_pfn_strategy.h` file) `strategy` value. The `flags` field is ignored and finally the `value` encodes modulo base and required result.

```
struct pfn_strategy modulo_strategy = {
     .owner = MODULE_NAME,
     .name = "MODULO",
     .check_hint = modulo_check_hint,
     .get_index = modulo_get_index,
     .get_strategy = modulo_get_strategy,
     .put_strategy = modulo_put_strategy,
     .in_use = modulo_in_use
};
```

where

`modulo_check_hint` - Checks the hint for `NULL` and whether the hint value contains correct values of base and result (`base≠0` and `result<base`)

`get_index` - Calculates the index of the first page frame with `pfn % base = result` within the given page order range (`idx = (result - (page_pfn%base)) + (result < (page%base))?base:0`).

Where `result` and `base` is obtained from `hint->value` and `page_pfn` is the given page PFN.).

`get_strategy`, `put_strategy`, `in_use` - Controls the current usage of a strategy and if it is non-zero, it locks the module from being unloaded.

Following functions are exported by a module, so that other modules can use them. They need just to include the `include/linux/modulo_pfn_strategy.h` header file.

```
unsigned long create_modulo_value(unsigned base, unsigned result);
unsigned get_modulo_base(unsigned long value);
unsigned get_modulo_result(unsigned long value);
```

Provides simple manipulation with encoded values in the hint value.

```
struct page * alloc_modulo_page(gfp_t gfp_mask, unsigned int order,
     struct zonelist *zonelist, unsigned long vaddr, unsigned base,
     unsigned result);
```

Prepares a hint from given parameters for the modulo allocation and calls the generic strategy allocator.

Page Coloring

This strategy is implemented as a kernel module (`mm/page_coloring.c`) and it is based on the *page coloring* approach [16], which tries to make the virtual to physical memory mapping more optimal for the real indexed cache usage.

The CPU cache is split to page sized *bins,* each one associated with a *color* (number from 0 up to `bins_count-1`). Both virtual pages and physical page frames are colored according to addresses `color = address % bins_count`.

The module has a optional parameter with `bins_count` called `colors_count` (e. g. `modprobe` *colors_count=512*). If no parameter is given, a configured value is used (see *9.1. Patch Set Installation*). It uses a hint with the `COLORING_STRATEGY` (defined in the `include/linux/page_coloring.h` file) `strategy` value. The `flags` field is used to indicate whether the hint holds already the initialized value. The `value` field holds the color number which should be used. The caller does not need to fill this value because it is initialized by the `coloring_init_hint` callback before it is used. The module exports

```
struct page * alloc_color_page(gfp_t gfp_mask, unsigned int order,
    struct zonelist *zonelist, unsigned long vaddr, unsigned long
    color);
```

Calls `alloc_modulo_page` with `colors_count` base and `color` result. Note that the modulo_strategy module has to be loaded too because of the symbol dependency.

```
unsigned long coloring_get_index(struct page * page, int
    page_order, struct pfn_hint * hint);
```

Calculates the first index of the page with the same `color` as `hint->value`.

The strategy itself is defined by the following structure with callbacks

```
struct pfn_strategy coloring_strategy = {
    .owner = MODULE_NAME,
    .name = "PAGE_COLORING",
    .alloc_pages = coloring_alloc_page,
    .init_hint = coloring_init_hint,
...
};
```

where

`coloring_alloc_page` - *Calls `alloc_color_page` function with color from `hint->value`.*

`coloring_init_hint` - *Checks whether flags contains the `F_INIT` value and if so, initialization has already been done and there is nothing to do. Otherwise it calculates a color from a given virtual page address and stores it to the value field. Flag is marked by the `F_INIT` value.*

This approach exploits the application memory access with the spatial locality behavior (the concept that the likelihood of referencing a memory location is higher if a location near it has been referenced [24]. Virtual pages which are close together are mapped to different bins thus cache misses occur less often when data on pages

are accessed.

Note that this strategy can rapidly reduce the performance non-determinism, because mappings are equivalent between separate runs from the cache point of view.

Bin Hopping

Bin hopping represents another cache sensitive mapping strategy [16]. It is very similar to *page coloring*; it also splits the cache into *bins* with colors, physical page frames have colors according their addresses. The only difference is that virtual pages do not have colors calculated from their addresses but rather according the time when they are used. The allocator always uses the next color which was used for the last allocation.

The strategy is implemented as a kernel module (see the `mm/bin_hopping.c` file) which has up to 2 parameters. The same as *page_coloring* module, the `colors_count` parameter specifies the number of bins. The second `color` parameter represents the color for the first allocation. The default value defined in the configuration is used for non-specified parameters (*9.1. Patch Set Installation*).

Hint with the `BINHOPPING_STRATEGY` (see the `include/linux/bin_hopping.h` file) strategy value is required. The `flags` field is not used. Although the `value` field is used for the color for allocation, it is not required from the user and it is initialized before allocation. The module keeps the color for immediate usage and increases this value whenever a hint is initialized by the `init_hint` callback. The strategy is defined as follows:

```
struct pfn_strategy binhop_strategy = {
     .owner = MODULE_NAME,
     .name = "BIN_HOPPING",
     .alloc_pages = binhop_alloc_page,
     .init_hint = binhop_init_hint,
     .get_index = binhop_get_index,
};
```

where

`binhop_alloc_page` - *Calls* `alloc_color_page` *with a color defined in* `hint->value`*. Note that the page_coloring module has to be loaded too because of symbol dependency.*

`binhop_init_hint` - *Initializes the given hint->value with the current color and posts the increment color value for the next allocation (*`color = (color + 1) %`

`colors_count`).

`binhop_get_index` - Simple alias for the `coloring_get_index` function exported by the page_coloring module.

This approach exploits the application memory access with the temporal locality behavior (the concept that a memory that is referenced at one point in time will be referenced again sometime in the near future [24]. Virtual pages which are accessed in a short time are spread to different bins thus cache misses occur less often for them.

8.2.3 Hint Page Fault Handling

The strategy on-demand mapping creation requires storage for hints and modified page fault handler, as stated in the Implementation Overview chapter. This chapter discusses hint the table data structures and API in the first part and the page fault handler in the second.

Hint Fault Table

Hint fault table (*HFT*) is storage which associates hints for page fault handling with virtual page address. It contains also a lock for the parallel access serialization and the default hint. The table is represented by

```
struct fault_hint_table {
     struct rw_semaphore   lock;
     struct rb_root        table_root;
     struct pfn_hint *default_hint;
};
```

where

`lock` - Read-Write semaphore for the table access serialization. Note that semaphores enable sleeping during the lock state [8, Chapter 5].

`table_root` - Root node of the table red black tree. This field contains a node field which is `NULL` if the tree is empty. The Linux kernel provides the generic implementation of the most of the red black tree behavior located in the `include/linux/rbtree.h` file (the header file also provides documentation and how to for usage). The user of the tree structure is responsible only for the insert and lookup functions implementation which are dependent on data stored in the tree. All the work with tree balancing is implemented by the generic code.

`default_hint` - Default hint for virtual pages which do not have any entry in the table. The `NULL` value is used for no default hint value.

HFT is process specific and it is stored in the `mm_struct`[15] process memory descriptor structure. It is initialized by the `init_fault_hint_table` function which is called during the process creation when the memory descriptor is initialized (see `mm_init` defined in `kernel/fork.c`).

The tree node value is called *hint cluster* and represented by

```
struct hint_struct {
     unsigned long   vaddr;
     struct rb_node  table;
     unsigned    usage;
     struct pfn_hint pfns[PFNS_PER_HINT];
};
```

where

`vaddr` - Virtual address for the first hint slot. The value is aligned to the `PFNS_PER_HINT` value (`vaddr % PFNS_PER_HINT = 0`).

`table` - Field used for the red black tree node. A pointer to it is stored in the tree and the tree generic API provides translation back to the whole structure.

`usage` - Number of used slots. When this number is lowered down (during hint removing from the table) to the 0, whole node is removed from the tree.

`pfns` - Slots for hints. Each one keeps a hint structure without the need of allocation. The number of slots is compile-time constant defined by the kernel configuration.

Clusters are organized according to the `vaddr` value and they do not overlap (see *7.2.2 On-Demand Mapping Manipulation*). The documented internal API for the direct table manipulation can be found in the `mm/physmaps_fault_table.c` file. In fact, there are two external API. The first is for the in-kernel usage and the second for the user space data usage (now used by the proc interface – see *8.2.5 Proc User Interface*). They are split because of a better configuration, because fault tables can be provided just for the in kernel usage without any user interface.

- In kernel usage API declared and documented in `include/linux/physmaps_hint.h`:

`init_fault_hint_table(struct mm_struct * mm)`

Initializes FHT for given descriptor. This function is called from the `mm_init` function which is responsible for the memory descriptor allocation and initialization.

`cleanup_fault_hint_table(struct mm_struct * mm)`

Destroys FHT and deallocates all clusters stored inside. The function is called from the `__mmdrop` function which destroys memory descriptor (defined in `kernel/fork.c`).

```
get_fault_hint(struct mm_struct * mm, unsigned long vaddr, struct
    pfn_hint *pfn, struct rw_semaphore ** lck)
```

Tries to find a hint for the given `vaddr` value and if it is present, makes copy to the given `pfn`

15 With exception of kernel threads which don't have any descriptor.

value and calls `init_pfn_hint` to do the special strategy initialization of the hint. If no hint is present in the table, uses the defined default hint. To prevent from race conditions, the pointer to the locked (for reading) semaphore is returned by the `lck` parameter. When the user does not need the hint, it has to release the lock for reading (by `up_read` function) as soon as possible.

```
update_fault_hint(struct mm_struct * mm, unsigned long vaddr,
    struct pfn_hint *pfn_hint)
```

Updates or inserts the given hint to the table. If no cluster for given `vaddr` exists at the moment, one is created. Moreover, calls `get_pfn_strategy` which is used for the hint which tells the strategy implemented as a module, that it should not enable to unload.

```
remove_fault_hint(struct mm_struct * mm, unsigned long vaddr, int
    count)
```

Removes hints for the given `count` of virtual pages starting from `vaddr`. If it results in an empty cluster, it is removed from the tree. Similarly to the update function, calls `put_pfn_strategy` on removed hints.

```
set_default_hint(struct mm_struct * mm, struct pfn_hint *pfn_hint)
```

Allocates a new default fault hint, if no is present, or updates current one with `pfn_hint`.

```
get_default_fault_hint(struct mm_struct * mm, struct rw_semaphore
    **lck)
```

Returns the default fault hint. The lock is treated in the same way as the `get_fault_hint` function.

- User space data usage API declared and documented in `include/linux/physmaps.h`. All functions are wrappers for their in-kernel counterparts with additional handling of given user space data which can't be access directly by the kernel and needs a special treatment. Note that the `__user` modifier tells a compiler to check and disable dereferencing (they are reported as an error). User data structures are defined in the `include/linux/physmaps_ioctl.h` file and will be described in *8.2.5 Proc User Interface*.

```
user_get_fault_hints(struct mm_struct *mm, struct
    physmaps_fault_hints __user *hints)
user_update_fault_hints(struct mm_struct *mm, struct
    physmaps_fault_hints __user  *hints)
user_remove_fault_hints(struct mm_struct *mm, struct
    physmaps_fault_hints __user  *hints)
user_set_default_fault_hint(struct mm_struct *mm, struct
    physmaps_fault_hint __user  *hint)
user_get_default_fault_hint(struct mm_struct *mm, struct
    physmaps_fault_hint __user  *hint)
```

Strategy Page Fault Handler

As already mentioned in *8.1.1 Memory Management Description*, Linux uses low level architecture specific (`do_no_page`) and architecture independent handlers (`__handle_mm_fault`). The low-level handler is not interesting for strategy mappings because it does not manipulate with them. Therefore we have focused on the high-level handler.

Whereas other parts of the patch set are not intrusive and do not change the standard kernel implementation, page fault handling requires direct intervention to the handler's code. The first is the FHT usage and the second is modification of the PTE fault functions.

The standard handler prepares a page table entry for a faulted address and calls the `handle_pte_fault` PTE fault handler. We have added the `get_fault_hint` call before the entry handler and if a hint is found it is used for the `handle_pfn_pte_fault` function. If no hint for address exists or the pfn PTE handler is not able to handle the fault because of no available page frame for a strategy (`VM_PFN_FAULT`) or fault case is not supported (`-ENOTSUPP`), the standard handling path is used without any hint.

`handle_pte_fault` was slightly changed to prevent a lot of code duplications between a hint and no hint cases. The new `__handle_pte_fault` function, which takes the original implementation with an additional hint parameter, was added. If this parameter is `NULL`, the original code path is used. Otherwise the hint specific decisions are used. `handle_pte_fault` is just alias for the low level handler with the `NULL` hint (see Figure 8.8). Therefore a code which is not aware of hint handling can use `handle_pte_fault` without any changes.

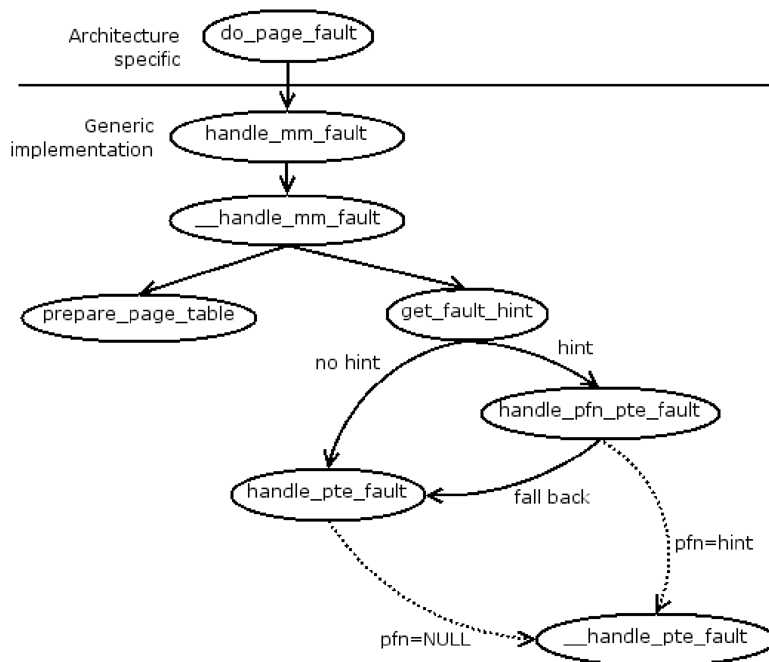


Figure 8.8: Page fault handler call graph with fault hint specific changes.

`__handle_pte_fault` does not handle a fault directly, but rather finds out a correct handler according the fault and memory type, and uses the specific handler for the independent cases discussed in *8.1.1 Memory Management Description*.

All case specific handlers were changed to use the additional hint parameter. The following text discusses their implementation modifications:

- anonymous memory is handled by

```
do_anonymouse_page(struct mm_struct *mm, struct vm_area_struct
    *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        int write_access, struct pfn_hint * pfn)
```

No hint (standard) implementation allocates and maps new zeroed page frame if write access is required or creates COW mapping to `ZERO_PAGE`[16]. If `pfn` is specified, it has to do allocation for both cases. If allocation fails returns with `VM_PFN_FAULT`.

- memory with provided `nopage` callback for VMA is handled by

```
int do_no_page(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd, int
    write_access, struct pfn_hint * pfn)
```

Function is called for not existing mappings from VMA which provides its own `nopage` callback. That is the reason why this function is called in the first place. However, the callback does not get the hint parameter. Therefore it cannot use the strategy allocation. If the returned page frame does not comply the given hint, it needs to be remapped by the `migrate_pfn_page`.

16 ZERO_PAGE is preallocated global shared read only page frame. It contains only zeros.

This handler also optimizes the early COW case which occurs when the write access is required for not shared VMA. In such a case, the current page frame needs to be copied to the newly allocated page frame which is mapped in the end. The strategy allocation is used, instead of the standard one, if the given hint is not `NULL`.

- non linear mapped memory is handled by

```
do_file_page(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd, int
    write_access, pte_t orig_pte, struct pfn_hint * pfn)
```

Hint case is not supported by the current implementation. If given `pfn` is not `NULL`, returns with `-ENOTSUPP` error code.

- swapped out memory is handled by

```
int do_swap_page(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd, int
    write_access, pte_t orig_pte, struct pfn_hint * pfn)
```

This function does not need any direct change, because it does not do any direct allocation (where given hint could be used). When it needs to get a page frame for swapped out data, it uses the `swapin_readahead` function (defined in `mm/swap_state.c`) which implements the read ahead logic for a swap in process implemented in the `read_swap_cache_async` function. Therefore the `read_swap_cache_async` function needs to be changed to use strategy allocation rather than normal one if possible. Nevertheless this function does not get the hint parameter, therefore it needs to get the hint from the FHT. Hint parameter adding would be very intrusive, because this function is used in many places in the kernel.

- COW memory is handled by

```
do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd,
    spinlock_t *ptl, pte_t orig_pte, struct pfn_hint * pfn)
```

The legal write access for a read only marked PTE is handled by this function. It simply copies the current page frame content to the newly allocated one. If the hint is not NULL, the strategy allocator is used rather than the standard.

The function implements one optimization for situation when the current page frame can be preserved. This occurs when a page frame is mapped only to a faulted virtual anonymous page frame (e. g. A process maps the anonymous private memory, forks a new child process which has the mapped area too with COW table entries. When the parent unmaps area, the child is the only one who has the mapped page frames.). Allocation of a new page would be wasting, because the current page is fried after a new mapping is created. Therefore the page frame is kept and PTE is just marked also as writable.

Optimization for the present hint case needs to check the current mapping and if it does not comply to the hint, the strategy allocator is used for a hint page. If allocation succeed, `migrate_pfn_page` is used for correct remapping.

8.2.4 Memory Layout Remapping

The Linux kernel implements the page migration code which is responsible for virtual memory remapping. It is primarily used for NUMA architectures, when the mapped pages are to be moved to a different node for a better communication with CPU [7]. We have reused this code for our remapping purposes with some little changes which do not affect the original implementation.

Remapping API provides two levels of abstraction

- *Low level API* which is responsible for remapping of a concrete page frame to an already existing and ready to be used one. It is represented by

```
migrate_pfn_page(struct page * original, struct page * target)
```

This function (defined in `mm/migrate.c`) simply prepares data for the page migration and uses the kernel migration code. There is only one difference, that the standard migration code implicitly returns the original page frame to the page cache (which is used for kernel caches) and this one just decreases reference count (if it falls down to 0, it is freed). This approach was chosen because of the further usage of the original page frame (e. g. for new mapping, because it is not mapped anywhere after the migration). Note that the original implementation, with the page cache, works as well.

- *High level API* is responsible for the strategy mappings manipulation. It is implemented in the `mm/physmaps.c` file. The main and only visible from outside is the function

```
set_pfn_page(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, gfp_t gfp_mask, struct pfn_hint * pfn)
```

The only responsibility is to check given parameters and prepare parameters for the internal `set_pte_pfn_page` function.

```
set_pte_pfn_page(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pte_t * pte, pmd_t * pmd, struct
    pfn_hint * pfn)
```

Internal function for remapping. In the first step it checks whether the current mapping exists and if not or it is a COW entry, uses `handle_pfn_pte_fault` with the given hint. Otherwise it checks if the given mapping complies the given hint. If not, a new page frame is allocated with the strategy allocator. The prepared page frame is then used for the target when the low level `migrate_pfn_page` is called.

8.2.5 Proc User Interface

The user interface for communication with the previously mentioned layers is provided by the process specific proc infrastructure, as already mentioned in *7.3. File User Interface*. The `/proc/<pid>/physmaps` file is used for two types of communication operations, `read` and `IOCTL`.

Both are implemented independently, thus can be configured separately (see *9.1. Patch Set Installation*). The read approach has advantage, that the file can be processed by any standard text manipulation tools very easily (e. g. snapshot of the current memory layout can be stored with a simple redirection to a file: `cat /proc/$PID/physmaps > $PID.physmaps`). `IOCTL`, on the other hand, provides a set of operations for both manipulation and querying for the current layout state. However, it needs a special program which knows commands and data structures.

The code for process the specific proc file registration is located in the `fs/proc/base.c` file. We have added a new entry to the `tgid_base_stuff` and

tid_base_stuff arrays with owner and group permissions for the physmaps file. The first array describes files for all *tgid* specific files (thread group ids - i. e. standard UNIX process) while the second one *tid* specific files (thread id – i. e. tasks). pid_directory_inos was updated to contain an identifier for both the process and task specific files. The proc_physmaps_operations structure is defined in the fs/proc/task_mmu.c file and contains all supported file operations. The proc_pident_lookup function was updated to handle both the PROC_TGID_PHYSMAPS and PROC_TID_PHYSMAP cases and provides with the proc_physmaps_operations structure with file operations for the physmaps file (see Figure 8.4 for generic case).

```
struct file_operations proc_physmaps_operations = {
	.open		= physmaps_open,
	.read		= seq_read,
	.llseek		= seq_lseek,
	.release	= seq_release_private,
#ifdef CONFIG_PROC_PHYSMAPS_IOCTL
	.unlocked_ioctl = physmaps_ioctl,
#endif
};
```

where

physmaps_open – reuses the do_maps_open function with the proc_pid_physmaps_op seq_file operations (see Read Operation for more information).

read, llseek, release – uses the generic seq_file implementation for file reading, seeking and closing.

unlocked_ioctl – callback for the ioctl call (see the IOCTL Implementation for more information). Note that the standard ioctl callback is not used, because it holds the big kernel lock [8, Chapter 5] and this approach is not recommended anymore. The callback is enabled only if IOCTL is configured before compilation.

Read Operation

The physmaps file reading provides the current memory layout in the plain text format style. This requires to iterate through the process VMAs and querying for mappings of all virtual pages from areas. As a result, a big amount of data can be produced, thus we cannot assume, that everything fits in a buffer provided by the user from the read call and need to keep track of the current VMA to be used and last data printed.

The VMA iteration functionality is already implemented by the /proc/<pid>/{maps, smaps} files (see the fs/proc/task_mmu.c file and

proc_smaps_operations, proc_maps_operations), which use the seq_file concept and implement common iteration functions for process memory regions traversing. We have reused the initialization and iteration code.

```
do_maps_open(struct inode *inode, struct file *file, struct
    seq_operations *ops)
```

Initializes data structures (private data for task identification and current state of iteration) and opens seq_file with given operations.

The physmaps as well as maps and smaps files defines the seq_file operations by structure:

```
static struct seq_operations proc_pid_physmaps_op = {
    .start     = m_start,
    .next      = m_next,
    .stop      = m_stop,
    .show      = show_physmap
};
```

where

m_start, m_next, m_stop – are common for all mentioned files and implement the iteration logic for VMAs.

show_physmaps – defines the unique logic for printing information to the user buffer. The physmaps's implementation calls show_physmap_range for the current VMA given (current) to the show callback. The function gets mappings for all pages from the VMA's range and prints them to seq_file which simply forwards data to the user buffer.

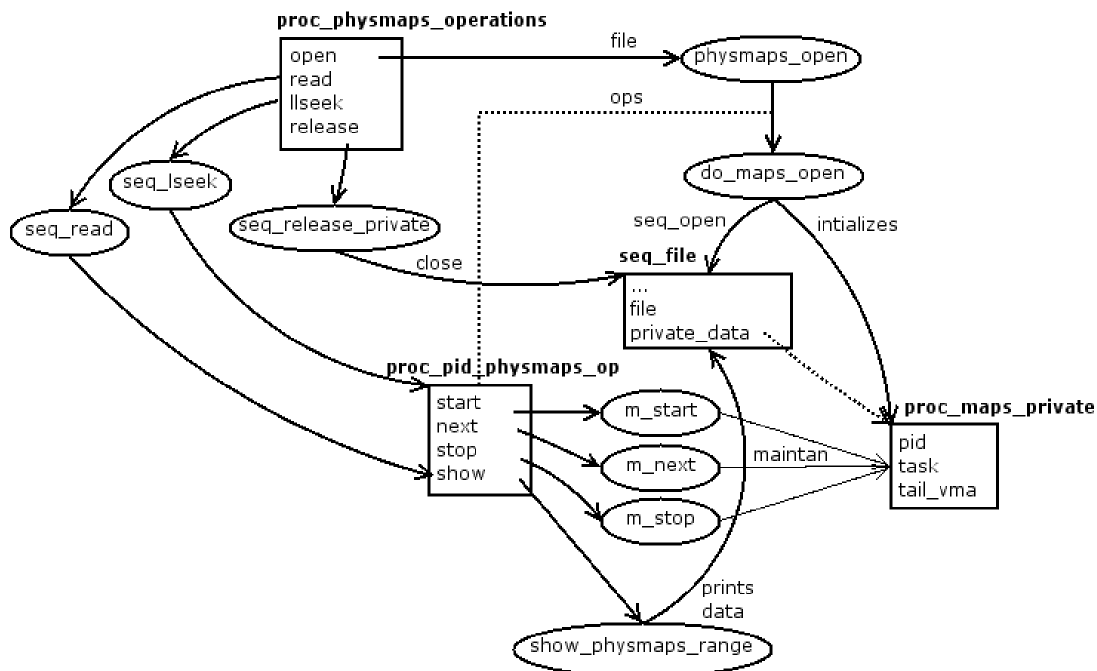


Figure 8.9: physmaps seq_file infrastructure.

The Figure 8.9 describes the call graphs, relations and data structures used for

the `seq_file` infrastructure. It is important to note, that most parts are already implemented, thus they can be reused without modifications and only specific behavior needs to be written for particular purpose (the `show_physmaps_range` and `physmaps_open` functions for this call graph).

IOCTL Implementation

The VFS code handles all ioctl calls in a generic way by the `vfs_ioctl` function. It handles common ioctl commands and calls file specific callback, defined in the `file_operations` structure, for all others. Therefore it is important that the ioctl commands have unique values without any lashes. The kernel provides helper macros for a comfortable construction of command values defined in `include/asm-generic/ioctl.h`.

The IOCTL commands for the `/proc/<pid>/physmaps` file are defined in the `include/linux/physmaps_ioctl.h` header file which is intended for both the user and kernel usage. This file also defines the data structures which are used for commands:

```
struct physmaps_virt {
     unsigned long vaddr;
     unsigned long pfn;
     unsigned flags;
};
```

Describes mapping for a virtual page

where

`vaddr` – is the virtual address of the page.

`pfn` – is the PFN address. This value is filled in by the command handler.

`flags` – are flags for mapping. This value is usually filled in by the command handler.

```
struct physmaps_virt_area {
     unsigned long vm_start;
     unsigned long vm_end;
     unsigned long flags;
     struct physmaps_virt * pages;
     unsigned pages_size;
};
```

Describes an area of virtual memory.

where

`vm_start`, `vm_end` – is the address range for a memory area given by the ioctl caller.

`flags` – are lags for the area filled in by the command handler.

`pages`, `pages_size` – is an array with a given size allocated by the ioctl caller and used by command handlers to fill mappings from a specified address range.

```
struct physmaps_vma {
    int pos;
    unsigned long addr;
    struct physmaps_virt_area area;
};
```

Describes a VMA according to the position or virtual address. If `pos` has a non-negative value, it is used rather than `addr`. Mappings of the VMA are described using `physmaps_virt_area` with the whole range of VMA.

where

`pos` – is the position of the area. The value is supplied by the ioctl caller and it is ignored if it is negative.

`addr` – is the address within VMA. The value is supplied by the caller too and it is ignored if `pos` is used.

`area` – describes the whole VMA area.

```
struct physmaps_fault_hint {
    unsigned char strategy;
    unsigned char flags;
    unsigned long value;
};
```

Describes the hint for a page fault or remapping. It is a counterpart of the kernel `struct pfn_hint` and the fields have the same meaning.

```
struct physmaps_fault_hints {
    unsigned long addr;
    int hint_count;
    struct physmaps_fault_hint * pfn_hints;
};
```

Describes a group of hints for a memory range starting from `addr` with `hint_count` pages.

where

`addr` – is the first address for hints.

`hint_count`, `pfn_hints` – is an array of hints for a continuous virtual memory range.

An IOCTL call is handled by the `physmaps_ioctl` function (defined in `fs/proc/task_mmu.c`) which checks the given command and determines the handler for it with a correctly casted parameter. The generic handler must get `struct task` for a file, where ioctl was called (it is stored in the `seq_file` private data). The command specific handler gets this structure with the ioctl parameter, therefore it can work on the per process base.

8.3 User Space Library Implementation

Apart from the kernel implementation, the user space `libvmtune` library is also provided for simpler usage and abstraction from the proc file system usage. It is distributed separately and can be found on the associated DVD in the `libvmtune` directory (see *9 User Guide* for more information on how to use it).

The library API is based on the process specific opaque handle which is created by the `init_physmaps` function and destroyed by `destroy_physmaps`. The library translates the handle value into a proc `physmaps` file descriptor.

Most functions exposed by the library are simple wrappers for direct ioctl calls (with the handle translated to the file descriptor for the call) and we will not discuss them here. However, there are also functions with a higher logic which are discussed in the further paragraphs.

The user space memory allocator is usually represented by the `malloc` function. It is responsible for the virtual memory (does not do any physical memory) allocation. The physical memory is allocated in the standard on-demand way. We have provided the `physmaps_malloc` function which uses the standard `malloc` function and sets the fault hints for all virtual pages according to the given parameter. Note that this approach does not work for 100% of cases because the user space allocator keeps the freed blocks of memory and reuses them for the future allocations [17]. Mappings can already exist for the allocated memory, thus hints are not used at all.

The standard `mmap` function maps the file or anonymous memory to the virtual address space. The `physmaps_mmap` (like `physmaps_malloc`) library function provides a wrapper for the standard call with an additional fault hints setting.

Finally, the `physmaps_fork` and `physmaps_execve` functions are provided for the standard `fork` and `exeve` functions. Both get an additional constructor parameter defined as a function pointer which is executed after the wrapped function returns. It gets the initialized handler for the created process and can perform an additional setting as the default strategy setting etc.

9 User Guide

The thesis is accompanied with an attached DVD which contains a vmtune patch set, `libvmtune` library source codes, results from the benchmark and an VMWare image with a patched, compiled and ready to use kernel.

The DVD contains the following directories:

- `copying` – a directory with copying terms and a copy of the GPL licence.
- `fft` – a directory with use case FFT benchmark source codes [14]
- `kernel` – a directory with the Linux kernel source tar ball which can be used for a patch set
- `libvmtune` – a directory with user space library sources
- `scripts` – a directory with useful scripts
- `thesis` – a directory with the thesis attached documents
- `vmtune` – a patch set directory
- `vmware` – a directory with the VMWare image file and configuration

9.1 Patch Set Installation

All *vmtune patch set* files (located in the DVD `vmtune` directory) need to be copied to the unpacked kernel root directory (`/usr/src/linux` symbolic should show to the source tree root directory). Afterwards, the patch set needs to be applied. Either a standard *patch* utility for all patch set files (with *-p4* parameter) or the simple `scritps/apply_vmtune_patch.sh` shell script, which does everything automatically (must be run from the root of kernel source tree), can be used.

Configuration

After the kernel is patched with no conflicts, it has to be properly configured. The configuration starts with a `make config`, `make menuconfig` or `make xconfig` command [18]. We will prefer the `menuconfig` approach.

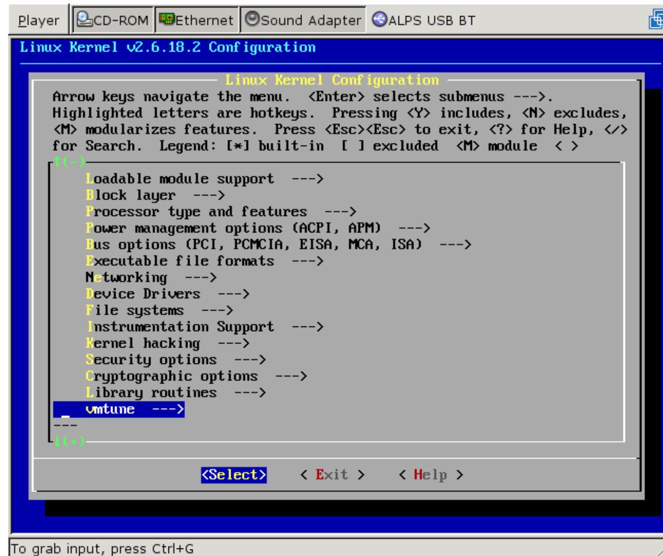


Figure 9.1: Main configuration screen.

The patch set comes with its own [***vmtune***] configuration menu item located in the main configuration menu (see Figure 9.1).

The configuration process is divided into 3 parts. The first (see Figure 9.2), represented by the [***Page fault handling***] item maintains the on-demand layout manipulation. It is disabled by default and the [***Hint table interface***] entry needs to be enabled for the page fault hints to work. Additionally, the [***Hint table entry cluster size***] specifies the number of fault hints stored in the table node (see *8.2.3 Hint Page Fault Handling*). This value does not need to be changed and the default value should work well.

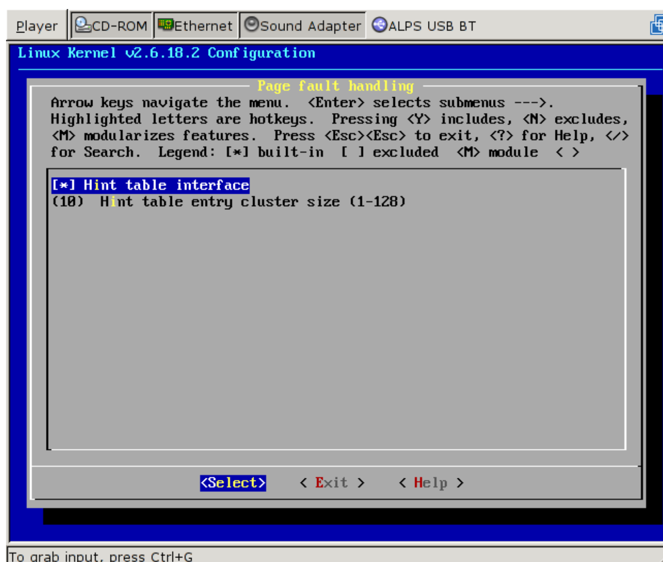


Figure 9.2: Fault hint configuration screen.

The second configures the proc user interface (see Figure 9.3) and is represented by the [***Physical memory manipulation through proc***] menu item. The [***/proc/<pid>/physmaps entry***] item enables the process specific proc `physmaps` file. Only the read operation is supported by default and IOCTL operations has to be explicitly enabled by the [***IOCTL calls for /proc/<pid>/physmaps***] entry.

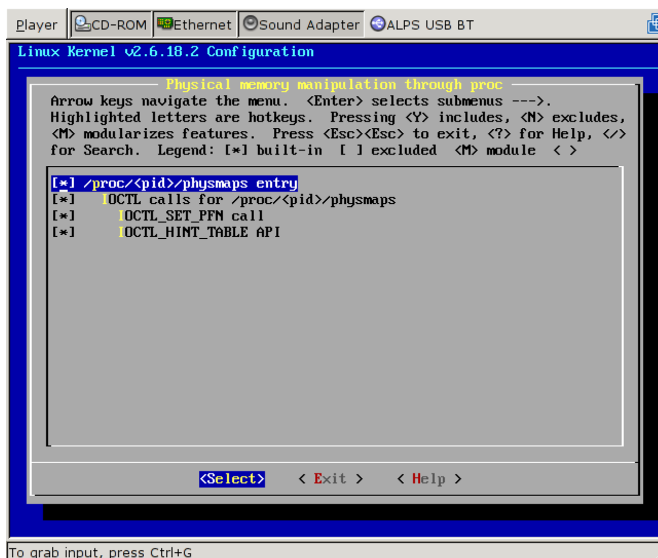


Figure 9.3: Proc file interface configuration screen.

The memory layout manipulation is not enabled by default either and has to be explicitly enabled by specifying the [***IOCTL_SET_PFN call***] entry for direct and [***IOCTL_HINT_TABLE API***] for on-demand manipulation (note that [Hint table interface] must be enabled).

Finally, the implemented strategies can be configured in the [***pfn hint strategies***] menu item (see Figure 9.4). The first entry [***Maximum number of hint strategies***] sets the number of possible strategies. The default value can be used without a problem, unless more than 10 strategies are to be used. A concrete strategy configuration follows. Each strategy can be compiled directly to the kernel (* selection) or as a kernel module (M value). An empty selection means that a strategy is not used at all.

[***Modulo hint strategy***] configures the *Modulo* strategy (see *8.2.2 Strategy Allocator*).

[***Page coloring hint strategy***] configures the *page coloring* strategy (see *8.2.2 Strategy Allocator*).. The additional [***Number of colors***] value should be also set for

the target system (L2 cache size / page_size should work well). [Modulo hint strategy] must be enabled (as a module or build in) for this entry.

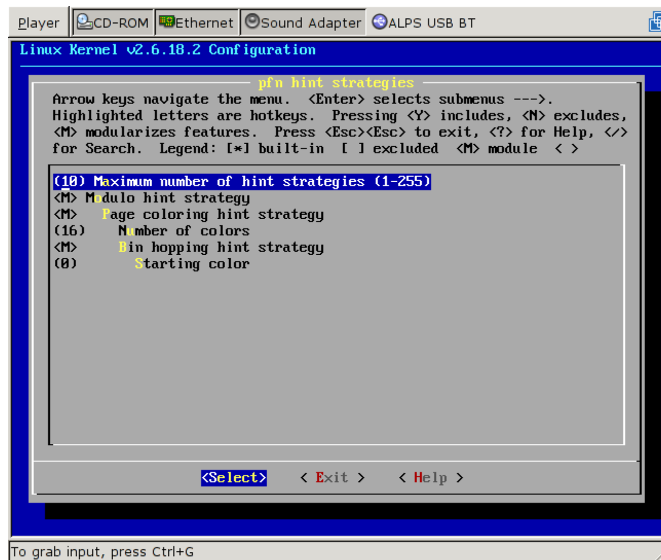


Figure 9.4: Strategies configuration screen.

[***Bin hopping hint strategy***] configures the *bin hopping* strategy (see *8.2.2 Strategy Allocator*).. [***Starting color***] sets the number of the first color used for allocation. The default value can be used without a problem. [**Page coloring hint strategy**] must be enabled (thus [**Modulo hint strategy**] too) for this entry. The [**Number of colors**] value is also used for this strategy.

Use Help for entries for more information, including the entry meaning and its dependencies.

Compilation

After the previous configuration step (including the other kernel parts configuration which were not discussed), the kernel needs to be compiled. Either the standard `make` and `make install` can be used or the DVD `scripts/debian_kernel.sh` script can be used for comfortable Debian package creation and simple installation.

The helper script runs in 3 modes. With no parameters, the compilation starts with configuration and continues with Linux image package creation. With one parameter (the value is not considered), no configuration is performed and only a Linux image is created. This option should be used with an already configured kernel (e. g. after a bug fix etc.). Finally, if 2 parameters are specified, a Linux header files

package is created too besides an image package.

The created packages are located in the /usr/src directory and they are prepared for installation. See script for more information.

9.2 User Space Library Installation and Usage

The user space library is distributed separately and sources are available in the DVD libvmtune directory which contains the source codes, header file and Makefile for library compilation and installation.

The library can be compiled with the make (with no parameters) command. Static and dynamic libraries are created after a successful compilation. make install can be used also for the system wide installation when both libraries are copied to the /opt/lib and header files to /opt/include directory (this can be changed in Makefile by changing the LIB_DIR and LIB_HEADERS variables). A shared library is additionally registered by the ldconfig command.

Applications have to include a header file and a link with a library. Both the vmtune.h header file and libvmtune.a static library can be directly included to the user source tree, if no system wide installation is available.

See the header file where its documentation is stored for more information about the library usage.

9.3 VMWare Image

The ready to use patch set installation is available on the attached VMWare disk image which contains the installed Debian system with both patched and standard distribution kernel. VMWare or VMplayer (http://www.vmware.com/) needs to be installed on a host system for the disk image usage. VMplayer is a free version of VMWare and can be downloaded from http://www.vmware.com/download/player/. See the installation instructions provided on the web pages.

VMplayer uses the DVD vmware/vm.vmx configuration file as a parameter if one enables the CDROM usage and networking for the guest system and provides 512MiB of RAM. The disk image has the capacity of 2,8GB. Note that files need to be copied to the hard drive, because VMplayer needs to have write access to the

directory with the configuration file in the standard installation.

The user with a *test* login is available (with a password). The user is a member of the src group, thus they are allowed to work with `/usr/src` and especially the kernel source directories. The root account has the same password as the test user.

A patched kernel source is available in `/usr/src/linux` (link target directory). The unchanged kernel is available in `/usr/src/linux-vanilla` (link target directory). Kernel header files needed for ioctl definitions are available in `/usr/src/linux-headers` (link target directory).

`libvmtune` is installed in the `/opt/lib` resp. `/opt/include` directory.

A compiled and ready to use FFT benchmark is available directly in the home directory (in `/home/fft`). An additional `split.sh` script is provided for simple processing of results printed by application.

10 Evaluation

We have implemented a new Linux kernel strategy based on the physical memory layout manipulation infrastructure. It can be used from the user space through the process specific proc file. The physical memory layout is exposed only in the read-only form and an application can change its mappings only by hints used by the strategies implemented in the kernel.

The new infrastructure enhances the standard kernel physical memory allocator and provides an additional logic defined by the strategy which can be implemented as a kernel module, thus loaded and unloaded during runtime as needed.

Strategies can be developed aside from the standard kernel on per application requirements. We have provided four strategies, *Exact*, *Modulo* and the well-known cache sensitive *page coloring* and *Bing hopping*, to show the internal API and data structures usage.

The generic strategy allocator is implemented as part of the infrastructure to facilitate the implementation of concrete strategies. Their code does not need to handle the low level physical memory maintaining data structure and can focus on the high level logic only.

An application can use either the direct proc file interface or the more convenient `libvmtune` library for the layout manipulation. It is also possible to change mapping to an application which is not aware of it, because it transparently uses the virtual memory and mappings are changed by another application. This implies that the target application for memory tuning need not be changed for that purpose.

10.1 Limitations of the Solution

The current kernel implementation is tested only on the x86 32b UMA architecture. Although the code does not use a platform specific code, no tests were done for other architectures. The low level physical memory allocation code and strategies implementation are not NUMA or UMA dependent, but the high level logic used for NUMA is not supported at all. Therefore no changes in the infrastructure need to be done after NUMA specific handling has been provided.

The low level physical memory generic strategy allocator cooperates with a swap daemon only in the standard way in memory tight situations. It does not provide any additional information which could be used during the reclaim process to free the strategy page frames preferably and thus decreases the number of retry attempts and the whole allocation time process in such situations.

Non-linear file mappings are not supported at all. Most of the code is hidden by file system specific callbacks, therefore it is rather hard to provide transparent generic implementation without many modifications to the driver specific code.

Although a strategy can provide its own allocation routine which can do some caching for pages, the swap daemon does not cooperate with strategies in low memory situations. It is possible that the system is out of the memory for processes even if there is a lot of memory cached by strategies which can be released. Note that there is a way how to tell a strategy that it should free cached pages (`shrink` callback see *8.2.1 Strategy Infrastructure*) but it is not used.

The current implementation of the Exact strategy is not suitable for the same layout creation for separate runs on the loaded system, because the required page frames may be used at the time when required by a strategy. In fact, it is hard to guarantee the same layout requirement without keeping the page frames for repeat mappings aside from the rest of the usage, because the free page frames are widely used for kernel caches. However, this could lead to the total memory consumption and uselessness of the system which is not acceptable.

There is no way how an application can set its hints or the default hint in a very early stage of running (e. g. during the fork or exec code path when a process is initialized). This means that all modifications can be done after some mappings are already created. We did not want to do any offensive changes into the fork or exec code, thus they do not use any hint information. However, it may be possible to provide the whole fault table for a new process in the future releases. The user space library provides a simple workaround where the initialization is done as soon as possible after the process is created (the `physmaps_fork`, `physmaps_execve` functions see *8.3 User Space Library Implementation*).

Finally, the virtual memory layout is not deterministic between runs. The Linux kernel randomizes the stack beginning position to prevent from code inject attacks when an

exec shield is used [1]. The user space allocator represented by the `malloc` function uses `mmap` for big block allocations without any requirements for the virtual memory position, thus repeated runs with the same sequence of allocations result in different usage of the virtual memory. Therefore it is impossible to have fault hints for situations when the virtual addresses are unknown. We have addressed this problem by simple wrappers for both the `malloc` and `mmap` function (`physmaps_malloc` and `physmaps_mmap` see *8.3 User Space Library Implementation*) which set hints immediately after allocation; resp. mapping is done by the system library function. However, this is not a solution for situations when the target application cannot be changed.

10.2 Related Work

A large amount of research has been performed on hardware and software based techniques for memory subsystems performance optimizations.

When considering the operating system, coloring techniques are the most spread. page coloring is implemented on e. g. Solaris [20], AIX [26] or MS Windows NT [4]. Bin hopping is implemented on e. g. Digital UNIX [4]. However, these allocation algorithms are usually hardwired into the kernel and there is no way how to add or remove a new one without the system modification.

Coloring policies are usually implemented as *static* policies in that they do not change the color of the page after the fault. There are also *dynamic* policies which optimize the case misses by recoloring the existing pages when a new physical page frame is allocated with a different color and mapping is changed accordingly. The operating system uses either special hardware for the cache miss observation (a form of a cache-miss look aside the buffer) [2] or a combination of TLB state information and cache miss counters to detect conflicts [22].

We have addressed the problem by a static approach with an additional modular architecture which enables further algorithms implementation without the need of kernel modifications. A dynamic policy is not supported though there is an API available for transparent page remapping.

10.3 Use Case Application

The Fast Fourier Transformation (*FFT) benchmark* application, as described in [14], was used as the use case application. It was modified to demonstrate a new strategy infrastructure impact on the application performance optimality and determinism. Additionally, memory layout initialization is done according to the given parameters (see *FFT Description* for more information).

FFT is a memory intensive application which shows significant influence of the initial random state on performance as stated in [14]. We have tried to decrease the initial random state impact by the CPU cache sensitive strategy memory layout manipulation. This approach should increase the performance and reduce its non-determinism.

FFT Description

The application is provided with source codes[17] available on the attached DVD (in `fft` directory) as well as a ready to use compiled `fft_rep` program on the VMWare disk image (in `/home/test/fft` directory).

```
fft_rep [strategy [default]]
```

The benchmark starts with initialization. Without any parameters, no memory layout initialization is performed. Single strategy parameter results in a page fault hints initialization for memory used by the FFT calculations (two global arrays `real` and `imag`). Two values are possible, `2` for page coloring and `3` for bin hopping. If the *default* parameter is also specified, the given strategy is used for the default hint. Therefore the data do not need any special initialization and a strategy is used for all mappings[18].

After initialization, a warm up part is executed. This involves the `TSTTrain` (defined in `/home/test/fft/defs.h`) number of FFT executions (called iterations). As a side effect, the memory used for calculation is accessed and fault hints are used for on-demand mapping creation. Results from these iterations are not considered in the result.

17 An executable program is compiled with the `make` command. Note that `libvmtune` must be available for successful compilation.

18 This is true for all mappings created after the default hint is set, those created before are not touched.

Finally, the `TSTSnapshots` number of FFT iterations is executed and the time for each execution is stored (the time is measured in CPU cycles consumed during the execution). The results for all iterations as well as the total time of execution including the warm-up (also in CPU cycles) are printed. A simple `split.sh` script is provided to separate the iteration times and total time from the `fft_rep` output.

Testing Environment

Fujitsu Siemens E8020D notebook with 2 GHz Pentium M CPU[19] (without the CPU scaling and other dynamic frequency features turned on), 2GiB RAM memory and 2MiB L2 Cache was used with an installed Debian (Sarge) linux distribution.

Test cases were run in a single user mode with minimum services enabled and the normal run-level with X window system and several applications running during the testing. Just the results from the single user mode are discussed here, but the complete results are available in the attached file located on the DVD in the `thesis/bench.ods` file.

The patched kernel supports all available functionality and with *modulo_strategy* compiled directly to the kernel and *page_coloring* and *bin_hopping* are compiled as loadable modules and loaded with

```
modprobe page_coloring colors_count=512
modprobe bin_hopping colors_count=512 color=0
```

Test Cases

`fft_rep` is compiled with `TSTTrain=3` (3 iterations for a warm up) and `TSTSnapshots=15` (15 measured iterations). The results from iterations are collected in *Iteration Data* table while the total times (including the warm up) are stored in the *Whole Run Data* table in the `thesis/bench.ods` file which contains all measured data and tables. The following configurations are examined:

- The standard distribution kernel used for runs with no parameters. The result are stored in the *Distro kernel normal case* column.
- The patched kernel used for a run with no parameters with the results in the *Patched kernel normal case* column.
- The patched kernel used for a run with a page coloring parameter and the results in the *Page coloring case* column.
- The patched kernel used for a run with a bin hopping parameter and the

19 One CPU cycle is then approximately $2x10^{-9}$ s.

results in the *Bin hopping case* column.

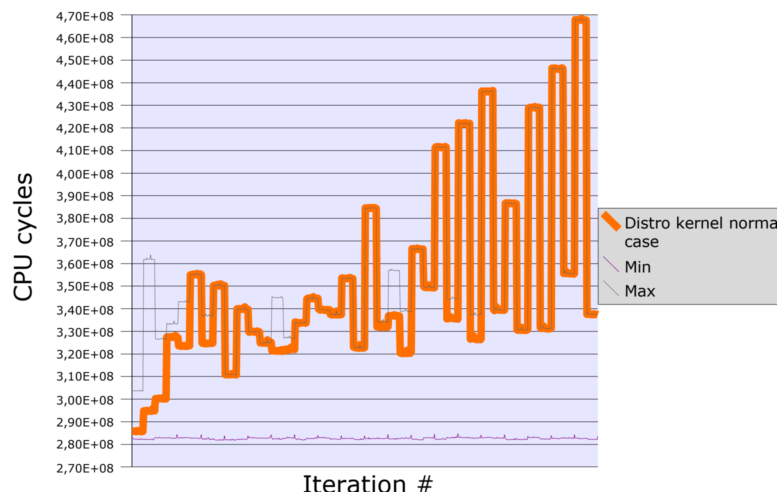
- The patched kernel used for a run with the default page coloring and the results in the *Default Page Coloring* column.
- The patched kernel used for a run with the default bin hopping and the results in the *Default Bin hopping* column.

Each configuration is repeated 40 times and the results are cumulated in the appropriate column. We are interested in iteration times from one run and differences to the rest runs and times which includes also warm up.

Therefore the minimum, maximum, standard error and difference between maximum and minimum for each column is calculated (see the brief results in Table 10.10 and Table 10.10). What is more, each iteration and run (depending on the table) is compared for all configurations and the minimum and maximum is calculated (Min and Max on graphs). The results are stored in the DVD `thesis/bench.ods`.

Test Cases Evaluation

The following graphs show the complete data from iterations from all runs (each 40 iterations stand for one run). The *x-axis* represents the separate iterations and the *y-axis* the CPU cycles consumed for each of them (the lower the number, the better the performance). *Max* resp. *Min*. values are the maximum resp. minimum value from all configurations and the same iteration (according to the order).

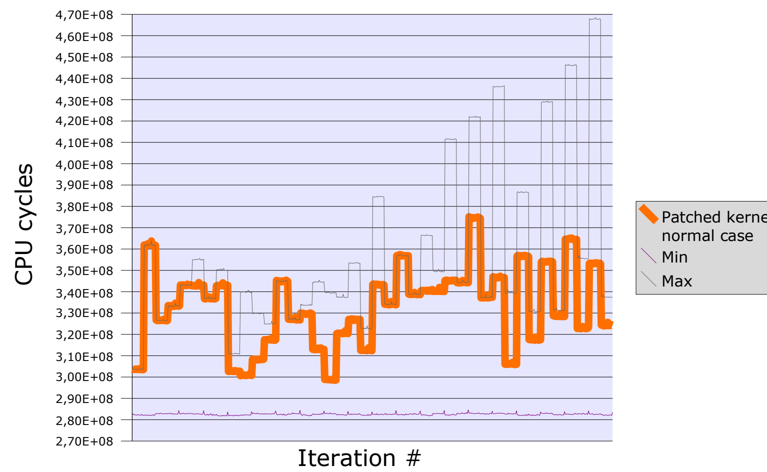


Graph 10.1: Distribution kernel with no memory layout initialization.

Graph 10.1 shows that the FFT benchmark has rather steady performance for

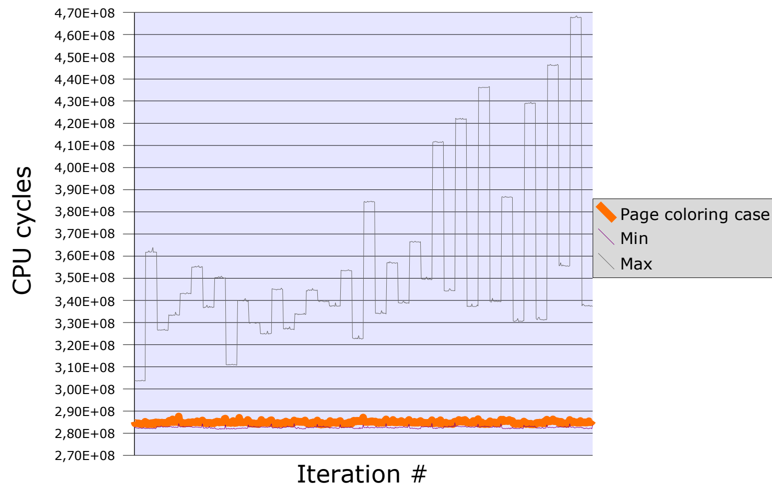
one run (represented by the vertical continuous lines) and very different values for the separate runs (jumps from the vertical lines). This implies that the memory mapping (created during the warm up) is highly significant for a separate run performance.

Another interesting piece of information is that the values are always worse than the minimum from all configurations; moreover, very often they have the worst values (maximum).



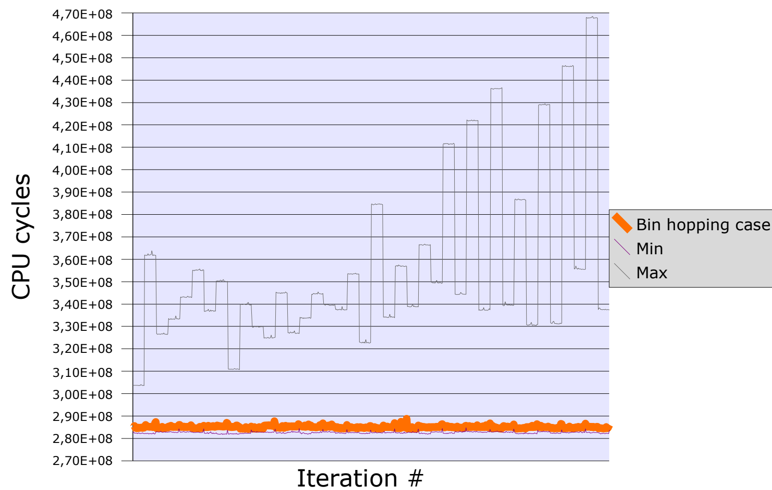
Graph 10.2: Patched kernel with no memory layout initialization.

Graph 10.2 shows the same scenario with a patched kernel. This configuration shows very similar results to the distribution kernel and the time consumed for the separate runs differs for separate runs. It seems to work a bit better on average when compared to the distribution kernel, which may be caused by the Debian specific patches applied to the standard kernel or simply a better sequence of random mappings for separate runs.



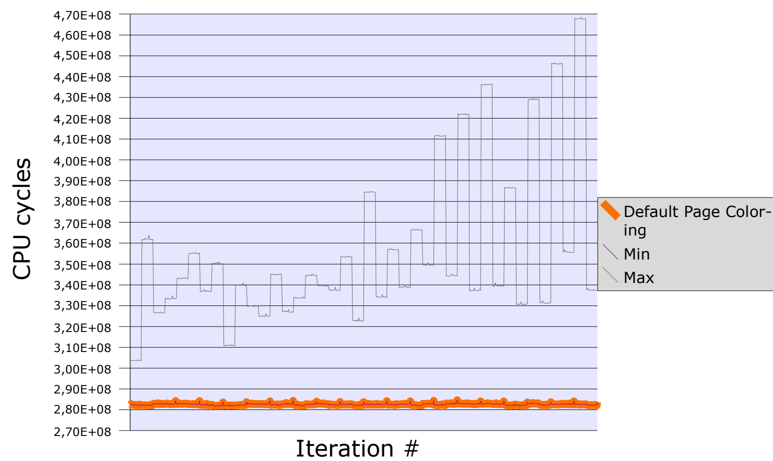
Graph 10.3: Page coloring strategy initialization for FFT data.

The page coloring case described by Graph 10.3 shows that initialization helped both improve the performance and determinism when compared to the distribution and patched kernel with no initialization. The performance keeps within a small range and values are near to the Min values.



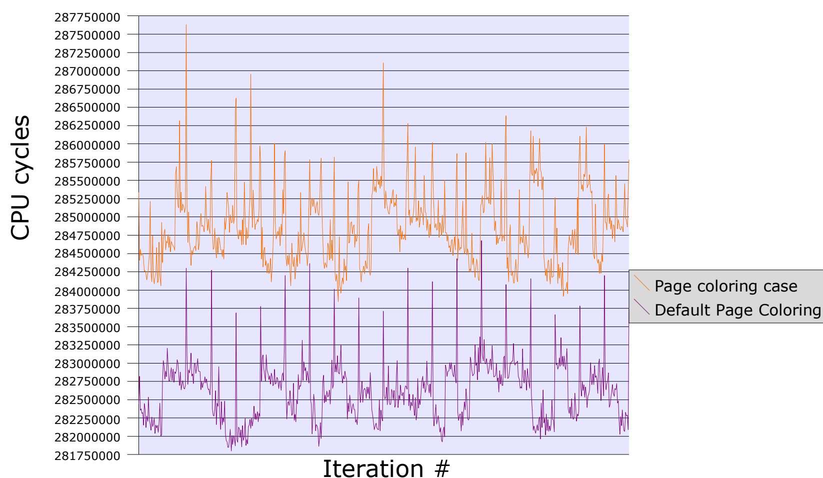
Graph 10.4: Bin hopping strategy used for FFT data.

Similar to the page coloring, the bin hopping initialization for data helped improve the performance and lower the differences between the separate runs (see Graph 10.4).



Graph 10.5: Default page coloring fault hint.

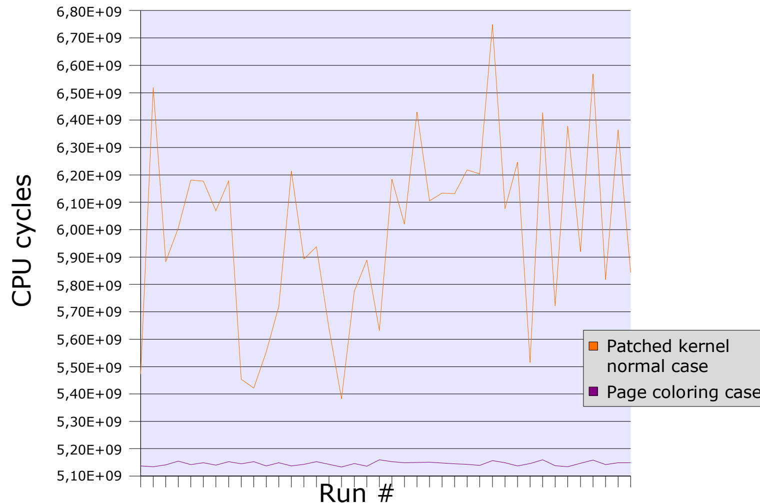
Graph 10.5 shows that the initialization with a page coloring default fault hint reached the best results for almost all iterations and differences between runs are very similar to the standard page coloring configurations. This means that page coloring mappings are good also for the rest of the memory, not only for the region used during calculations. A close comparison of both approaches can be seen in Graph 10.6. It seems that the default hint usage leads to a better performance with a similar variability of performance between runs when compared to a certain memory area page coloring hints.



Graph 10.6: Comparison of page coloring for calculation data and default page coloring.

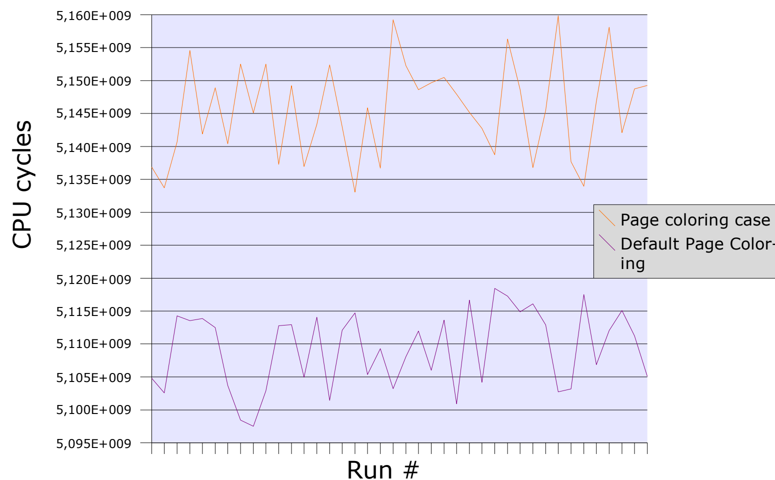
The previous graphs described iterations and performance relation on the per iteration bases. The following graphs compare configurations from perspective of the

whole run where also the warm up part is involved. Thus we also have the information about the time used for strategy fault hint handling. If the allocation overhead is too big, the total time is bigger than for the standard mapping even if separate iterations are faster with a strategy.



Graph 10.7: Comparison of patched kernel with no initialization with page coloring for FFT data.

However, Graph 10.7 shows that page coloring allocation for the FFT data is not a bottle neck and despite of the additional strategy logic the application benefits from better performance for all runs unlike the standard mapping.



Graph 10.8: Comparison of FFT data and default page coloring strategies for runs.

Finally, Graph 10.8 shows the run statistics for the page coloring initialization

used only for the FFT data and for the default fault hint. It is obvious that the default case is better, than the FFT data case, for all runs in performance while the determinism very similar. This means that the overhead for allocation is not significant.

Table 10.9 and Table 10.10 show the numbers for all configurations from iterations and per run points of view. Values in rows are calculated from all collected values and yellow fields stand for the best while the red for the worst values.

	Distro kernel normal case	Patched kernel normal case	Page coloring case	Bin hopping case	Default Page Coloring	Default Bin hopping
Min	285665047	298530191	283842001	284155460	281798979	284009575
Max	468439466	375152097	287631283	288761473	284674996	287911963
Max-Min	182774419	76621906	3789282	4606013	2876017	3902388
Average	349763916,72	333175078,74	284848234,77	285064442,81	282613353,1	285248269,15
Median	337249291,5	336038110	284779570,5	285055773	282630953,5	285162790

Table 10.9: Configurations data for iterations. Values are calculated from all collected iterations in CPU cycles. No further precision corrections were done.

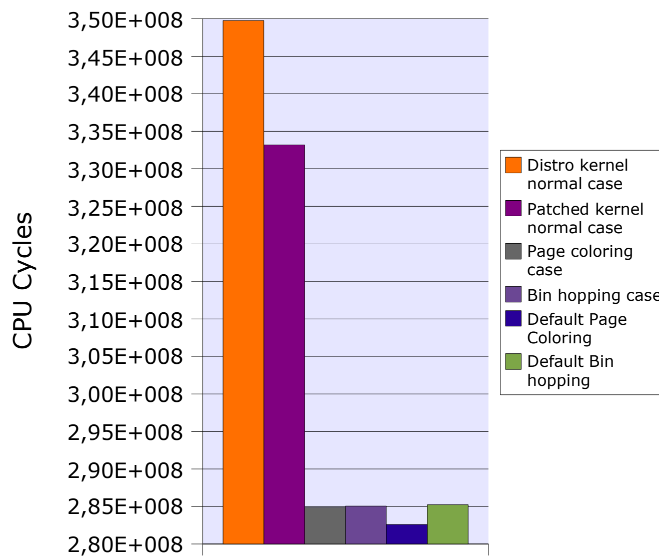
Min is the best (the lowest number of CPU cycles) for either one iteration (Table 10.9) or the whole run (Table 10.10). In the same way, *Max* is the worst performance. *Max-Min* row holds the difference between the best and worst performance. This value very roughly approximates the variance of the performance for each configuration. The lower the number, the more stable is the performance for iterations. *Average* represents the arithmetic mean from all measured values (iterations or per run data) for each configuration. Finally *Median* is found as the middle value from all arranged values for each configuration.

	Distro kernel normal case	Patched kernel normal case	Page coloring case	Bin hopping case	Default Page Coloring	Default Bin hopping
Min	5148476022	5382792757	5133066310	5139861162	5097521717	5135972333
Max	8425278239	6748788229	5159831984	5165997936	5118454634	5171992874
Max-Min	3276802217	1365995472	26765674	26136774	20932917	36020541
Average	6299403415,83	6001296685,35	5145584926,73	5150086457,55	5109247320,33	5151665658,18
Median	6073761010	6044804816,5	5145658874	5151319195	5111601612,5	5150776244,5

Table 10.10: Configurations statistics for whole runs. Values are calculated from all collected per run numbers in CPU cycles. No further precision corrections were done.

Even though no precision corrections were done on calculated values, we can see that the cache sensitive strategies increased performance on average for both iterations and per runs. While the average results are very similar for the page

coloring and the bin hopping configurations (the differences are approximately 10^6 CPU cycles ≈ 0,5 milliseconds and 10^8 CPU cycles ≈ 50 milliseconds for whole runs), there is a big difference when comparing to no strategy usage for both the distribution kernel and patched kernel (the difference is approximately 10^7 CPU cycles ≈ 0,5 seconds for iterations and 10^{10} CPU cycles ≈ 5 seconds for whole runs). See Graph 10.11 for illustration.



Graph 10.11: Iterations average comparison for configurations.

When focusing on the difference between the best and worst performance for all configurations, the page coloring and bin hopping strategies have values approximately 10^7 (≈0,5 seconds) for iterations and 10^8 (≈50 milliseconds) for whole runs while no strategy configurations 10^8 for iterations and even 10^{10} CPU cycles (≈5 seconds) for whole runs. The difference is in one resp. two orders.

Presented results are very rough and more precise statistic calculations need to be done for the more precise picture of cache sensitive strategies impact.

11 Conclusion

It is known that the physical memory layout, represented by virtual to physical memory mapping, has big influence on the CPU cache efficiency, and thus on the performance. Well organized mappings eliminate conflicts on real-indexed caches and the CPU can access data more effectively, whereas random mappings tend to have sub-optimal and non-deterministic behavior.

This thesis provides new extensible functionality for the Linux kernel which enables physical memory layout manipulation for user space applications. Current Linux kernel uses only one allocation algorithm for layout creation which produces quite random mappings. Therefore, some applications show big differences in their performance between separate applications runs. This can be a problem especially for benchmarking when random initial state of the application physical memory layout, set during allocations at application start-up, has significant influence on the measured data.

The provided infrastructure enables adding of new strategies into the system without the core kernel modifications. The strategies can be selected on per process basis, defaulting to the current strategy implemented in the kernel.

As a proof of concept of the kernel extension, page coloring and bin hopping algorithms strategies, which are known to have positive influence on a CPU cache efficiency, were implemented. Many operating systems use these strategies by default. We have demonstrated that the use case application which used cache sensitive strategies achieved both better performance and its determinism when compared to the original Linux kernel.

The new kernel extension can be used by applications with special memory requirements which can provide their own strategies and benefit from their knowledge of the memory usage. In addition to that, regression benchmarking can exploit the possibility of memory layout initialization for the target application, and thus simulate the same memory usage in actually different benchmark executions.

The current implementation of the kernel extension can be extended basically in two ways. By implementation of new strategies for special groups of applications and user interface with higher-level logic for special usage, such as benchmarking. Both

tasks can be accomplished without kernel modifications.

List of Abbreviations

COW – copy on write mapping
CPU – central processing unit
DMA – direct memory access
FPU – floating point unit
GNU – Gnu's Not UNIX: A UNIX-compatible operating system developed by the Free Software Foundation
HFT – Hint fault table
IOCTL – input output control.
ISA – Industry Standard Architecture (technology for connecting computer peripherals)
kiB, *MiB*, *GiB* – binary multiples for memory storage size (1kiB = 1024B, 1MiB = 1024kiB, 1GiB = 1024 MiB)
MMU – memory management unit
MMX – multi-media extension
NUMA – non uniform memory architecture
PAE – physical address extension
PFN – page frame number
PGD – page global directory
pid – process identifier
PMD – page medium directory
POSIX - Portable Operating System Interface for uniX, a family of standards developed by the IEEE.
PUD – Page upper directory
PTE – page table entry
RAM – random access memory
SMP – Symmetric multi-processor
TLB – translation look aside buffer, which is a small associative memory that caches virtual to physical mappings
UMA – uniform memory architecture
VFS – virtual file system
VMA – virtual memory area

Literature

[1] Arjan van de Ven (2004): "New Security Enhancements in Red Hat Enterprise Linux v.3, update 3"; *Red Hat Inc.*

[2] Bershad B., Lee D., Romer T., Chen J. (1994): "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches"; *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), pp. 158-170, ACM.*

[3] Brown N. (1999): "The Linux Virtual File-system Layer"; http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html.

[4] Bugnion E., Anderson J., Mowry T., Rosenblum M., Lam M. (1998): "Compiler-Directed Page Coloring for Multiprocessors"; *Proceedings of The Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems, ASPLOS.*

[5] Callahan D., Kennedy, Portfield A. (1991): "Software prefetching"; *In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 40–52, ACM.*

[6] Corbert J., (2003): "Driver porting: The seq_file interface, Linux Weekly News"; http://lwn.net/Articles/22355/.

[7] Corbert J., (2005): "Page migration, Linux Weekly News"; http://lwn.net/Articles/157066/.

[8] Corbert J., Rubini A., Kroah-Hartman G. (2005): "Linux Device Drivers (3rd edition)"; *O'Reilly & Media Inc. 1005 Gravenstein Highway North, Sebastopol, CA 95472.*

[9] Denning P. (2005): "The locality principle"; *Communication of the ACM, Vol. 48, No. 7, pp. 19-24, ACM.*

[10] Genua P. (2004): "A Cache Primer"; *Freescale Semiconductor..*

[11] Gorman M. (2004): "Understanding the Linux Virtual Memory Management"; *Prentice Hall.*

[12] Harty K., Cheriton D. (1992): "Application-Controlled Physical Memory using External Page-Cache Management"; *In Proc. of the 5th ASPLOS, pp. 187-199, ASPLOS.*

[13] Hyde R. (2001): "The Art of Assembly Language, Vol. 2"; *No Starch Press.*

[14] Kalibera T., Bulej L., Tuma P. (2005): "Benchmark Precision and Random Initial State"; *In Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems, SCS 2005.*

[15] Kenney J. F., Keeping, E. S. (1962): "Mathematics of Statistics, Pt. 1, 3rd ed."; *Princeton, NJ: Van Nostrand.*

[16] Kessler R., Hill M. (1992): "Page Placement Algorithms for Large Real-Indexed Caches"; *ACM Transactions on Computer Systems, 10(4), Nov 1992, ACM.*

[17] Korn G. and Kiem-Phong Bo. (1985): "In search of a better malloc"; *In Proceedings of the Summer 1985 USENIX Conference, pp. 489-506, Portland, OR.*

[18] Kroah-Hartman G. (2006): "Linux Kernel in a Nutshell"; *O'Reilly & Associates*.

[19] Krueger K., Loftesness A., Vahdat A., Anderson T. (1993): "Tools for the Development of Application-Specific Virtual Memory Management"; *OOPSLA, pp. 48-64.*

[20] McDougall R., Mauro J. (2006): "Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition)"; *Prentice Hall.*

[21] Mouw E. (2001): "Linux Kernel Procfs Guide"; *Delft University of Technology.*

[22] Romer T., Lee D., Bershad B., Chen B. (Nov. 1994): "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware"; *In Proceedings of the First Symposium on Operating Systems Design and Implementation, pp. 255-266, ACM.*

[23] Weisstein, Eric W. (2007): "Red-Black Tree."; *From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/Red-BlackTree.html.*

[24] Wolf M., Lam M. (1991): "A Data Locality Optimizing Algorithm"; *In Proceedings of ACM SIGPLAN 91 Conference Programming Language Design and Implementation, Toronto, Ont., pp. 30–44, ACM.*

[25] GNU Hurd Documentation (2007): *Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA. http://www.gnu.org/software/hurd/docs.html*.

[26] AIX documentation (2007): *IBM Corporation North Castle Drive Armonk, NY 10504-1785 USA. http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prftungd/doc/prftungd/replace_vmtune_schedtune.htm*.

[27] Linux Kernel home page (2007): *www.kernel.org*