System Architecture Group
Faculty for Computer Science
University of Karlsruhe

Study Thesis

dm_phys:
A Dataspace Manager for Physical Memory

Tobias G. Dussa

2002–03–05

This study thesis was typeset with the computer typesetting system TEX/LATEX and the `koma_scr` document class.

Abstract

This document describes `dm_phys`, a dataspace manager for physical random-access memory within the experimental L4/SawMill operating system environment.

The study thesis includes a brief introduction to SawMill's dataspace model and an overview of existing memory allocation and release strategies. Furthermore, the principal design requirements and model constraints for `dm_phys` are considered, after which the design decisions used in `dm_phys` are discussed.

Finally, implementation details as well as testing setup, results and interpretations are given.

Acknowledgements

My thanks go to Dr. Kevin Elphinstone for supporting and supervising my thesis as well as for his insights and discussions.
Most of all, my gratitude goes to Stefan Götz, who has spent an obscene amount of time introducing me to the SawMill development environment and helping me troubleshoot my SawMill setup. Without him, I would have been hopelessly lost.
Finally, I would like to thank all those who supported my work in one way or another. Thank you all!

Karlsruhe, 2002–02–15 — *Tobias Dussa*

Contents

1 Introduction **7**
- 1.1 The SawMill Dataspace Concept . . . 7
- 1.2 Thesis Contributions . . . 8
- 1.3 Structure of the Thesis . . . 9

2 Background and Related Work **10**
- 2.1 Memory Allocation Strategies . . . 10
  - 2.1.1 Linear Fit . . . 10
  - 2.1.2 Segregated Fit . . . 11
  - 2.1.3 Buddy System . . . 12
- 2.2 Memory Release Strategies . . . 12
  - 2.2.1 Instant Coalescing . . . 13
  - 2.2.2 Delayed Coalescing . . . 13
- 2.3 Linear-Linked Lists Versus Binary Trees . . . 14

3 Design **15**
- 3.1 Prerequisites . . . 15
  - 3.1.1 Requirements . . . 15
  - 3.1.2 Model Contraints . . . 16
- 3.2 Design Decisions . . . 17
  - 3.2.1 Functionality Considerations . . . 17
  - 3.2.2 Memory Allocation Strategy . . . 21
  - 3.2.3 Memory Coalescing Strategy . . . 22

4 Implementation **23**
- 4.1 Notes On the Data Structures . . . 23
  - 4.1.1 The Memory Region Data Type . . . 23
  - 4.1.2 The Dataspace Data Type . . . 24
  - 4.1.3 The Owner Data Type . . . 24
- 4.2 Notes On Some Interface Functions . . . 25
  - 4.2.1 The Share Function . . . 25
  - 4.2.2 The Transfer Function . . . 25
  - 4.2.3 The Unshare Function . . . 26
  - 4.2.4 The Close Function . . . 26
- 4.3 Miscellaneous Notes . . . 26

4.3.1 Portability With Regard To Pagesizes 27
4.3.2 Memory Acquisition At Startup 27

5 Validation **29**
5.1 Test Setup 29
5.2 Measurements 30
5.3 Interpretation 34

6 Conclusion and Future Work **35**

1 Introduction

Currently, the dominant approach to operating systems is to create a single, monolithic kernel, with virtually all lower-level system services provided by the operating system kernel itself. With this design, it is necessary to include all those services in the kernel, where they are executed in kernel mode, which introduces some unnecessary degree of insecurity into the system.

On the other hand, there are operating system design patterns that implement so-called *microkernels*. Microkernels offer very few operating system services themselves; the idea is to restrict the amount of code executed in kernel mode as much as possible. Any service that does not absolutely need to be run in kernel mode is removed from the kernel and therefore run in non-privileged mode, also known as user mode, thus offering a higher degree of security for the system as a whole.

One class of services that can partially be transferred from kernel to user mode is resource management in general, and — with some restrictions — management of the physical RAM installed in the system in particular.

This thesis is based on the *L4KA* microkernel and the *SawMill* operating system that is built around it. Since SawMill is relatively new, there is no memory management facility consistent with the philosophy found in SawMill; so far, physical memory can only be allocated to certain tasks at boot time, which is clearly unsatisfactory.

1.1 The SawMill Dataspace Concept

Throughout SawMill, the concept of *dataspaces* is used. A dataspace is an abstract, unstructured entity containing data. A dataspace may be a file, a whole disk, a frame buffer, physical memory, virtual memory, and so on.

Dataspaces, being abstract objects, do not necessarily have an intrinsic size, but may contain other data-containing objects partially or entirely. For instance, a dataspace may be attached to a region of a task's address space, or it may be attached to a consecutive block of physical memory.

Each dataspace is provided by a dataspace manager (although, of course, each dataspace manager may provide multiple dataspaces). The dataspace manager controls access to the dataspaces it manages. For example, the dataspace manager may grant read/write access to part or all of one of its dataspaces to a client task, or it may only grant read rights, and so on. The exact semantics of a particular dataspace are entirely dependant on the nature of the underlying storage facilities: a dataspace that contains ("is backed by") disk files may offer write operations, while a dataspace that offers access to

a PCI card ROM naturally cannot perform writes. Therefore, the semantics are mostly left to the dataspace manager, albeit some operations (for example, open or close) are mandatory for all dataspace managers, while others (for instance, share or transfer) are optional but strongly encouraged.

Dataspace managers may be stackable on top of each other, so that one dataspace manager exports a dataspace to another dataspace manager, which in turn provides some access service backed by the dataspace from the first dataspace manager. A possible scenario for this is the physical memory dataspace manager `dm_phys`, which may export a block of physical memory to a virtual memory dataspace manager, which in turn provides virtual memory by using standard paging techniques on the physical memory. Thus, a user task may request, for instance, 512 megabytes of memory from the virtual memory dataspace manager and receive it even though the virtual memory dataspace manager has been given access to only 256 megabytes of memory by the underlying `dm_phys`. In this example, the virtual memory dataspace manager may provide the missing 256 megabytes of memory by swapping to secondary storage.

Further introduction to SawMill's virtual memory concept can be found in [APJ+99].

1.2 Thesis Contributions

It is the aim of this thesis to provide a reasonable facility for management of physical memory that is consistent with SawMill's dataspace model. In accordance with the philosophy that each dataspace manager should be as specialized as possible and serve only one basic function, the resulting dataspace manager for physical memory (which shall be referred to as `dm_phys` further on) should only manage physical memory. In particular, services that should be provided by other dataspace managers — which may be stacked on top of `dm_phys` — include:

- virtual memory services, i. e. swapping, paging, and similar activities, and
- compaction, i. e. relocation of used memory blocks to improve overall memory utilization.

To begin, we will restrict ourselves to the management of the RAM installed on the system's mainboard. Physical memory in general would include such instances as PCI card ROMs is too broad a basis for `dm_phys` as presented in this study thesis. Simply speaking, the goal is to provide a memory allocation/release method within SawMill that is comparable to standard Unix `malloc`/`free` calls.

Thus, the key properties of `dm_phys` are:

- memory allocated to a client is guaranteed not to be taken away from the client later without its agreement,
- memory allocated to a client is also guaranteed not to be paged out to secondary storage at any time, and finally

- memory allocated to a client is guaranteed not to be relocated within the physical memory. Thus, it is possible for a client to request memory at a certain physical address in order to communicate to peripherals, for instance.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 gives an overview over common memory allocation strategies and implementation techniques. Furthermore, existing SawMill dataspace managers are presented.

Chapter 3 addresses the specific requirements and constraints for the given scenario. Chapter 4 explains the design decisions and describes some actual implementation details. Testing and measuring is found in chapter 5. Finally, Chapter 6 gives the conclusion of the results found.

2 Background and Related Work

This chapter presents an overview over common memory management strategies as well as a brief introduction into two implementation techniques. Additionally, a presentation of the existing dataspace managers for SawMill is given.

2.1 Memory Allocation Strategies

Although there are a number of memory allocation strategies, all techniques can coarsely be classified into three categories: Linear fit, segregated fit, and buddy system.

An excellent, more in-depth description of many of the strategies presented in this section can be found in [Sta98]. [Knu97a] does not feature many of the strategies, but covers first fit and buddy system algorithms with great mathematical detail. For an in-depth, comprehensive literature review and strategy comparison, see [WJNB95].

2.1.1 Linear Fit

This is probably the most common class of memory management techniques. Linear fit strategies keep a single linked list of the memory available in the system. Whenever a request for a memory block is issued, the memory manager linearly searches through the entire list of available memory blocks until it finds a suitable region. At startup time, there is just one large chunk of memory available, of course, but each time a region of memory is handed out to a client task, that memory block needs to be removed from the available list.

As long as memory is only requested, then everything is straightforward. As soon as a block of memory is released by a client, however, then that memory is available for other client tasks again, and therefore needs to be re-introduced into the available list. At this point, *external fragmentation may occur:* it is unlikely that the released block of memory happens to be directly adjacent to one of the blocks that are in the available list. This means that even though, for instance, there may be a total of 128 megabytes of memory in the available list, stored in two separate, non-adjacent chunks of 64 megabytes each, a client request for 128 megabytes of memory will fail.

Different approaches have been tested to minimize the external fragmentation. The most common algorithms are first fit, next fit, and best fit.

First fit simply iterates through the list of available memory until it encounters a block of memory large enough to satisfy the request. While this may sound naïve, it turns out that this approach is on average the fastest linear fit strategy.

Next fit is a derivate of the first fit approach. Since first fit always uses the first suitable memory region it finds, it tends to produce more cluttered memory at the beginning of the memory. While this in itself is not a problem, it does introduce a possible disadvantage: if the clutter is very intense, and if there are relatively many requests to be processed, then first fit suffers because it needs to search through all the very small memory blocks left over at the very beginning of the available list. Next fit tries to face this disadvantage by not using the first memory block that is large enough, but rather starts the list iteration at the block that was last given to a client. Although the approach may sound promising, it does not quite reach the performance of first fit, because it tends to break up memory blocks at the end of the memory list, where usually the largest blocks can be found, and thus may lead to greater external fragmentation.

Best fit, despite its name, turns out to be the worst-performing approach presented. This algorithm always scans over all memory blocks available and uses the one that is closest in size to the request. Therefore, it guarantees that the memory fragment left over by the request is as small as possible, and thus minimizes the chance that a subsequent request will be able to make use of the fragment.

All of the above approaches have in common that whenever a memory block is released and added to the available list, then sooner or later adjacent available blocks of memory are coalesced into a single larger block. Obviously it is beneficial to merge adjacent blocks as soon as possible, but doing so takes time, while processing the whole available list at certain time intervals in order to perform such merging can be performed more efficiently. However, delayed coalescing has the drawback that between insertion and coalescing any request that may require a block of the size of the two regions to be merged will fail. Which technique is more advisable depends on the circumstances. In the `dm_phys` context, the relevant factors for this decision are the frequency of allocations and, even more importantly, deallocations. The sizes of the memory blocks involved do not matter as much, because it is expected that dataspaces managed by `dm_phys` will be relatively large.

2.1.2 Segregated Fit

While linear fit techniques store the available blocks of memory in a single linked list, segregated fit strategies maintain multiple lists of available memory blocks, each list being reserved for block of a certain size. (Usually, there is not one list for each possible block size, but rather one list for a range of block sizes.)

Within each block size category, the available memory chunks are maintained in linked lists. Basically, all fitting strategies described above for linear lists apply here, too, only they are applied separately for each list.

Upon insertion of a released block of memory into the appropriate list, no coalescing is performed whatsoever. More sophisticated approaches, however, allow for passing two adjacent smaller blocks up into a list of larger regions by merging them together, or vice versa, by splitting a region.

With segregated fit, memory blocks that are given to clients are usually larger than the requested memory size, because the blocks of available memory are not split up again. When a request for memory comes in, the block list with blocks of the smallest possible size that is still large enough is used, and one block simply handed out. Thus, block sizes are usually fixed.

2.1.3 Buddy System

The buddy system is a meet-in-the-middle approach between the fixed-size system of segregated fits and the variable-size system of linear fits. With this strategy, all blocks are of size 2^k. Initially, the entire memory available in the system is a single block of size 2^n for some n. The system maintains an available list for every size 2^k for all $k \leq n$.

Whenever a request comes in, the available list is used whose blocks are just large enough to hold the request. For an example, let the request ask for a memory chunk of 255 bytes. Then the available list of size 256 bytes ($= 2^8$) is used. If there is a block of size 2^8 available, then that block is handed out. Otherwise, the next-larger block size is considered; in out example, 2^9 bytes. If there is such a block available, then that block is split up in half to form two blocks of half size, the so-called *buddies.* One of those blocks is handed out to the request, the other one is inserted into the smaller-size available list. The cascading goes up until either 2^n is reached or an available block is encountered, which is then split up.

On the other hand, when a memory block is released again, then it is inserted into the appropriate available list. If its buddy is also in the available list, then the two buddies are coalesced and the larger block is inserted into the next-higher available list. The advantage here is that it is very easy to check whether a memory block's buddy is available or not, because buddies of size 2^k in a system with 2^n share their upper $n-k-1$ base address bits. This means that in a system with 2^8 bytes of memory, the buddy of a block of size $16 = 2^4$, starting at address 10110000, is a block of size 16, starting at address 10100000. The two buddies share the upper 3 bits of their base addresses, which happen to be 101 in this example.

2.2 Memory Release Strategies

This section of the thesis presents a short overview of the most common techniques used for efficient release of memory blocks.

Coalescing of freed memory blocks is not always necessary. The necessity of coalescing depends on the memory allocation strategy used. If, for instance, segregated lists of free memory blocks are used, with a fixed block size within each list, and only whole blocks are handed out, then no coalescing whatsoever is needed: whenever a memory block is freed, then it merely needs to be reintroduced into the proper available list. (Note that finding out which list is the appropriate one is not necessarily trivial either. Typically, the release of a block of memory consists of a call to the `free` function or an equivalent, which usually takes a pointer to the block's base address as an argument, but not the block's size. This means that the block's size needs to be deduced from other sources.

We will not address the problem in this section, though, as we regard this more an implementation problem than a design decision.) If, on the other hand, the buddy system is used, then every block that is freed should be coalesced with its buddy if the buddy is available also.

In principle, there are two coalescing strategies, *instant* and *delayed,* also known as *deferred,* coalescing. The difference between those two principles will be shown here.

2.2.1 Instant Coalescing

With instant coalescing, whenever a memory block is released, the block is immediately reintroduced into the appropriate available pool. If under the general memory management strategy coalescing with another block or other blocks in the available pool is appropriate, then the coalescing is performed immediately, without further delay.

The advantage of this strategy is that whenever two blocks of memory can be coalesced, they are coalesced at once. Therefore, it is guaranteed that at any time every free block of memory in the system is as large as it can be, given the external fragmentation of the memory at that instant. Specifically, it can never happen that there are two adjacent blocks of memory of sizes s_1 and s_2 available, but a request for a block of memory of size $s_1 + s_2$ fails. This could only happen if the two adjacent blocks are not coalesced and thus recognized as a single free block of memory of size $s_1 + s_2$.

The major drawback of this approach is its increased effort. Under certain circumstances, a coalescing attempt may be a very expensive thing to do. This is especially the case if the pool of free memory blocks is not ordered by base addresses and no structure whatsoever is imposed on the memory blocks, for example. In this situation, whenever a block is to be coalesced with its neighboring blocks — if there are any —, it is necessary to perform an exhaustive search through all memory regions in the available queue. If this queue is well-filled, then the search may be quite extensive, and the immediate coalescing of the freed block of memory rather costly. Furthermore, instant coalescing very much increases the workload whenever a block of memory is allocated and deallocated again, then perhaps being reallocated soon, and so on.

2.2.2 Delayed Coalescing

In the view of the considerations of the previous section, the other approach presented here can immediately show its benefits.

If coalescing is not done immediately when a block of memory is released, but delayed to a later point in time, then the costs for recoalescing may be dramatically reduced.

Consider the example from the last secion. In that scenario, free memory blocks are stored in such a way that their respective base addresses cannot be deduced quickly, but need to be found by an exhaustive search. The problem was that doing an exhaustive search for each coalescing is very expensive: the costs for this naïve approach are $O(n) \times O(m)$, with n being the number of free blocks in the ready queue and m being the number of memory blocks that are released.

Now let all freed memory blocks are be immediately recoalesced with their appropriate neighbors, but rather be stored in a separate ready queue, which is sorted by the memory blocks' base addresses. Then, whenever the queue has reached a threshold size, the coalescing procedure could be triggered, which could, for example, loop sequentially over all available memory blocks in the ready queue and perform a binary search on the (sorted) list of released memory blocks in order to find any partner memory blocks appropriate for coalescing. Even this simple approach already reduces the cost for coalescing significantly: while with the immediate coalescing strategy the cost was $O(n) \times O(m)$, the deferred coalescing approach reduces the cost to $O(n) \times \log O(m)$.

Of course, this approach also has a drawback, which is exactly the strength of the immediate coalescing method: with delayed coalescing, there may be two adjacent blocks of memory of sizes s_1 and s_2 available in the system, but no block of size $s_1 + s_2$, because the two adjacent blocks are not recognized by the memory manager as a single block of larger size, since coalescing has not yet been performed.

Which of the two approaches is more favorable depends strongly on the circumstances and the system behavior. It may even turn out to be most beneficial to employ delayed coalescing and let a non-fulfillable memory allocation request trigger the coalescing process. With this strategy, coalescing is delayed as long as possible. Yet, in general it is not trivial to predict whether it is beneficial to use this coalescing technique.

2.3 Linear-Linked Lists Versus Binary Trees

In this section, a brief comparison between linear lists and binary trees is presented. The traditional and intuitive approach to use for maintaining sets of available memory blocks is the use of linked lists, as has been used in the description of the various memory allocation strategies in the previous section.

Recent developments have shown, however, that binary-tree representations can also be used for this purpose. Indeed, large improvements can be accomplished, as would be expected, by the use of binary trees instead of linked lists. Very promising details can be obtained from [RK00].

The basic idea is to use binary tree structures instead of linked lists for managing available blocks of memory. Since searching for a particular element in a linked list of length n costs about $O(n)$, a great benefit can be drawn if binary trees are used instead: searching a particular element in a binary tree with n nodes takes $O(l)$ operations, with l being the height of the tree, so that $\log(n) \leq l \leq n$. The authors of the quoted paper come to the conclusion that according to empirical results, it can safely be assumed that in practice the binary trees do not degenerate very easily into linear lists, so that usually $\log(n) \leq l \leq 2\log(n)$. This, of course, is a significant improvement. The proof-of-concept and test implementations of this idea use the base address of the memory blocks as an ordering base for the binary tree; additionally, the size of the largest blocks in the left and right subtrees are tracked in each node. These implementations indeed show the speedup that is expected due to the above considerations.

3 Design

In the first section of this chapter, we will present the initial requirements that have triggered design decision considerations. The second section features the actual design decisions as well as some considerations on how the decisions were reached.

3.1 Prerequisites

In order to be of use within the SawMill environment, a physical dataspace manager needs to meet certain requirements. Some of these requirements are imposed by the nature of the memory that is to be managed, others are introduced by the SawMill dataspace concept. Constraints of the latter type are described in section 3.1.2, while those of the first type are presented in section 3.1.1.

3.1.1 Requirements

The primary task of a physical dataspace manager within the SawMill operating system environment is to keep track of physical memory and to provide easy access to physical memory pages for higher-level applications. In particular, it may be necessary for some task to access a well-defined page in physical memory, for instance for DMA transfers from or to an I/O extension in the system, such as a sound card or a networking card. This leads to the first requirements for the dataspace manager:

- It should be possible for a client task to request and get any particular, but client-defined memory page, and
- once a client task has been given access to a particular dataspace, which is backed by a set of particular pages of physical memory, then the dataspace should not be moved to another physical memory location.

Additionally, it may be important for some client tasks to have access to memory that is guaranteed by the system, in other words, that is not taken away from the client later on. For some client tasks, however, it is not sufficient to guarantee that memory will not be taken away permanently, but it is necessary that the entire dataspace remains in physical memory all the time. Typically, this is a property that is desirable for security-relevant applications: Any program that needs access to temporary storage in order to keep deciphered data or deciphering keys or the like requires those storage facilities to be safe in the sense that their whereabouts are well-known all the time. In order to maintain security, it is very important, for instance, that cryptographic keys or plaintext

never be stored in permanent storage such as magnetic disks. Therefore, it is necessary to ensure that memory used for storing such sensitive material is never written to secondary storage for paging purposes. Owing to the above considerations, further requirements for the physical dataspace manager follow:

- Once a client task is given access to a dataspace backed by physical memory, those memory pages used to back the dataspace must never be taken away from the client, and
- any memory page contained in a physical dataspace must never be written to secondary storage.

Fortunately, these requirements are easy to follow, since they essentially prohibit more complex functionality. Thus, the requirements can be met by keeping the dataspace manager simple.

Some of the functionality that is prohibited by the above requirements for the physical dataspace manager may be desirable in other parts of the system, however. In order to provide more virtual memory than the physical memory installed in the system allows for, paging may be desirable, for instance. Such features must be implemented by higher-level dataspace managers. For example, it is thinkable to have a swap dataspace manager that is backed by the physical dataspace manager, but offers virtual memory to higher-level applications by use of paging techniques.

3.1.2 Model Contraints

In addition to the requirements outlined in the previous section, there are some more constraints that are imposed by the SawMill dataspace model.

Mandatory Functions

In the SawMill concept, there are functions that every dataspace manager *must* provide. These functions are:

Open: Obviously, it is necessary that every dataspace manager provides a method to open a dataspace.

Close: It is obvious that there must be a method to close an open dataspace again.

The above methods are called by a client task. In contrast, there is another mandatory function a dataspace manager must provide which is not called directly by the client, but by the instance that handles the client's page faults, which in L4 is called the region mapper:

Map page: This function is called whenever a client task generates a page fault on a dataspace. The appropriate dataspace manager's `map_page` function is then called by the client's region mapper. It is then the responsibility of this procedure that the faulting memory page is actually mapped into the client task's address space.

Since the above functions are mandatory, it is clear that `dm_phys` has to implement those. The only design decision associated with these methods are their respective prototypes, in the sense that the number and semantics of the calling parameter of some functions need to be defined.

Optional Functions

Additionally, there are some functions that a dataspace manager may offer, but is not obliged to do so. These functions include:

Clone: With this function, an identical copy of the original dataspace is created.

Migrate: This function moves the dataspace to a different dataspace manager.

Copy: With this method, a dataspace copy is created in a different dataspace manager. This function is semantically identical with calls to *clone* and *migrate,* in that order.

Share: This procedure shares access to a dataspace with another client task. That access right may subsequently be revoked with the function *unshare.*

Unshare: With this function, a client task is able to take away access rights from another task with which it has shared the dataspace earlier.

Transfer: This method transfers ownership of a dataspace from one client task to another. This is much like a *share* operation, except that if a client transfers a dataspace to another client, it loses access rights to that dataspace. Incidentally, a *transfer* operation cannot be revoked later on.

It is clear that some of the functions described above are sensible enhancements to `dm_phys`, while other functions make absolutely no sense in the context of the design targets outlined in the introduction.

3.2 Design Decisions

In this section, we will present considerations and decisions that the above requirements and constraints have lead to.

3.2.1 Functionality Considerations

Decisions on which abstract functionality `dm_phys` should offer is fairly simple, since the design goals introduce fairly tight limits. In particular, the following decisions regarding the policy of `dm_phys` have been made:

- The `dm_phys` dataspace manager will try to acquire as much available physical memory as it can for itself at start-up time. This is a direct consequence of the vision that `dm_phys` should entirely replace all need for interaction between lower-level resource management instances and higher-level user tasks.

- In direct accordance with the requirements, once physical memory has been given out to a client, that memory will not be taken away from the client by `dm_phys` later on, nor will `dm_phys` relocate any dataspace in physical memory.

  It is noteworthy, however, that there is no way that `dm_phys` itself can guarantee that memory will never be taken away from a client. This is due to the fact that the original resource allocator, for instance σ_0, may only map the physical memory pages to `dm_phys`. Thus, in the L4 design, the underlying resource manager could at any time take any physical memory page away from `dm_phys`, and consequently, from any `dm_phys` client. This can in no way be avoided by `dm_phys`. Yet, if the original resource allocator actually transfers the physical memory to `dm_phys` so that it cannot unmap the memory from `dm_phys` later on, then `dm_phys` can indeed guarantee that no memory will be taken away from any client.

- Physical memory dataspaces will be handed out on a first-come, first-served basis. Because of the requirement that once memory is handed out to a client task, that memory may not be taken away from the client later on, the only alternative to this is to keep lists which allow certain clients to acquire certain dataspaces or amounts of memory under certain circumstances. While this may turn out to be desirable for certain scenarios, such as real-time systems in which certain tasks need to be guaranteed to have access to certain resources at certain times, we have decided to keep the design simple for starters.

With the above considerations, it follows that `dm_phys` will support the mandatory functions, with the following particulars:

Open: This function opens a new dataspace. In all cases, it is necessary to provide a requested size of the new dataspace. This is obviously necessary.

Additionally, there are two optional parameters. The first parameter is concerned with the requested page size. In other words, a client may ask for a dataspace consisting of memory pages of a particular size only.

The second optional parameter carries the base address of the requested dataspace. With this parameter, it is possible to ask for a certain physical page in memory.

If any of the optional parameters are left unspecified, then they are interpreted as "arbitrary," meaning that the corresponding values are completely up to the `dm_phys` server.

Of course, for each open request a number of plausibility checks are performed, such as alignment checks for both the requested size and base address, if any is specified. Both of these values need to be aligned to at least the smallest page size available in the system, and if another page size is requested in the *open* call, then both values need to be aligned to that page size.

Note that this implies that it is not possible to open a dataspace of arbitrary size. Instead, if a chunk of memory is required that happens to be not aligned with the smallest page size available, so that the size is not an integer multiple of the

available page size, then it is necessary to request a dataspace of the size of the next-larger integer multiple of the page size. The same argumentation goes with the base address, of course.

The *open* call also defines the *owner* of the newly-created dataspace, namely the client task that actually called the *open* procedure. Ownership of a dataspace can only be changed if the current owner task uses the *transfer* function to give up its own rights of access to the dataspace and hands them over to another task, which then becomes the new owner of the dataspace.

Close: This procedure returns all access rights to the dataspace it is used on. The task calling the *close* operation tells `dm_phys` that it does not want to access the dataspace any longer. Subsequently, `dm_phys` removes the access rights for that dataspace-client combination.

The *close* call is transitive. This means that if a client task has shared access to a dataspace with another task, and subsequently closes the dataspace, then the dataspace is also closed for all tasks it has been shared with by the closing task.

Furthermore, if there are no client tasks that have access rights to a given dataspace, that dataspace is destroyed and the physical memory used for backing that dataspace is freed and returned to the list of available memory blocks. This means that a dataspace is destroyed if and only if the owner task calls the *close* procedure: if the owner task does this, then access is taken away from all other tasks it has shared the dataspace with too, and since the owner task by definition is the task that all other tasks got their access rights from, there are no tasks left which still hold access rights to the dataspace. If, on the other hand, some other task closes its access handle to the dataspace, then there is always at least one task left which still has access to the dataspace: namely, the owner task.

Map page: This procedure is not called by any client tasks directly, so the discussion about it will be rather brief. This method is called whenever a client task triggers a page fault, in other words, whenever a client task tries to access a memory page that has not been mapped into its address space by the L4 kernel.

In general, this happens every time a memory page of a dataspace is accessed by the client for the first time after creation (or sharing) of a dataspace, since `dm_phys` does not map all memory pages of a dataspace to the client task at creation-time or when a dataspace is shared or transfered to another task.

In such a case, all that `dm_phys` needs to do is to actually map the faulting memory page to the client task. This is done only after due checking for proper access rights, of course.

In addition to the required methods, `dm_phys` will support three optional functions. These functions are their particulars are described in the following section.

Share: With the *share* function, a client task, the *sharer,* may share a dataspace with another task, the *sharee.* The sharer may or may not intend to share the dataspace

with the sharee indefinitely; in any case, it is possible for the sharer to use the *unshare* method to take away access to the dataspace from the sharee.

If a complete hand-over of a dataspace to another task is desired, then the *transfer* method should be used instead.

It is also noteworthy that any task with access to a dataspace may share its access with another task. In particular, it is not necessary that the task calling the *share* function be the task that originally opened the dataspace. In other words, use of the *share* method is *not* limited to the owner task.

Unshare: This function is the "antidote" to the *share* method. With the *unshare* method, a task, the sharer, may revoke a previous sharing of access rights to a dataspace with another task, the (former) sharee. For this procedure, it is not required that the sharee signal its agreements with the sharer's intention of unsharing a dataspace. Therefore, it should be noted that the guarantee that a once-assigned memory page is never taken away from the client is valid only with respect to the actual `dm_phys` server and the dataspace-owning task, *not* with respect to an arbitrary other task.

Furthermore, note that the *unshare* operation is transitive. This means that if a task t_1 shares a dataspace with another task t_2, which subsequently shares the dataspace with yet another task t_3, then if t_1 *unshares* the dataspace with t_2, access is also taken away from t_3. This behavior is fairly obvious: if the *unshare* operation were not transitive, then it would be completely pointless, because any client could retain access to a dataspace by sharing it with itself, thereby avoiding the *unshare* operation from the task from with it originally got access to the dataspace.

Transfer: This function is used to completely hand over a dataspace to another task. All access rights are given from the transferring task to the transfer-receiving task, which means that the transfer-originating task gives up all its access rights.

For this function, two annotations must be made. The first is that a transfer of access rights also includes a transfer of shared-access sponsorships. Thus, if a task t_1 shares access to a dataspace with another task t_2 and subsequently transfers its access rights to yet another task t_3, then from that point on effectively t_2 will have received its access rights from t_3. Therefore, only task t_3 can unshare the dataspace from t_2.

The other comment is that with the above remark, a transfer also includes transfer of the ownership of a dataspace. Thus, if a task t_1 owns a dataspace, meaning that it has originally opened the dataspace, and transfers that dataspace to another task t_2, then as far as `dm_phys` is concerned, t_2 owns the dataspace, and everything appears as if t_2 originally opened the dataspace. Of course, there is no way to determine whether a task really was the original creator task of a dataspace or whether it obtained ownership of the dataspace by the transfer of ownership from the original owner task.

In contrast, all of the functions *clone, copy* and *migrate* are absolutely senseless for `dm_phys`: It is impossible to hand over control over a dataspace consisting of RAM pages to another dataspace manager. Furthermore, it is equally impossible to create a clone of the dataspace, since each dataspace is conceptually linked to the physical memory pages it consists of.

The *copy* operation, however, may turn out to be useful. Even though, strictly speaking, a dataspace exported by `dm_phys` cannot be moved to another dataspace manager without losing some of its inherent characteristics, it is thinkable to include support for opening a dataspace of the same size with another dataspace manager, and then copying all of the contents of the physical dataspace into the newly-created dataspace. Due to the philosophy that every system component should only provide functionalities that are essential to its original task, it is not the responsibility of `dm_phys` to mediate such an operation. If such a function is required, no further intervention of the `dm_phys` server is needed; instead, the client task wishing to create a shadow copy of its dataspace can perform all necessary tasks itself, since the desired operation can be done by using only functions that are exported by `dm_phys`. Therefore, `dm_phys` does not offer the *copy* operation.

3.2.2 Memory Allocation Strategy

The following section will describe the memory allocation strategy employed in `dm_phys` and outline the argumentation in favor of the strategy used.

Within `dm_phys`, simple *linear first fit* allocation policy is used. The reasoning for this decision is based on the goals of simplicity and compactness, both of which are key concepts of SawMill policy. SawMill design is founded on the introduction and use of many very small and simple but well-adapted server tasks. Each of these servers is designed and implemented to perform a specific and simple job. The servers are particularly optimized for their original tasks; they are not at all expected to serve other functions.

With this in mind, it is very desirable to have an easy-to-understand, easy-to-troubleshoot and easy-to-implement design. Since it is not envisioned that `dm_phys` be used on a large scale by end-user tasks — there are many functionalities that need to be added on top of `dm_phys` in order to make memory allocation as readily and easily usable as that of existing operating systems, such as paging functionality in order to provide more virtual memory than physical memory in the system —, it is expected that as such dataspace managers become available, the majority of requests for physical dataspaces will occur at system startup time when higher-level dataspace managers are loaded.

Since linear first fit is usually among the top-performing memory allocation strategies, it seems reasonable to trade in a little peak performance, which is expected to be rarely made use of anyway, for the virtues of simplicity and compactness.

3.2.3 Memory Coalescing Strategy

For coalescing policy decisions, the same reasoning holds true as for memory allocation policy, as described in the previous section.

Again, it is expected that user tasks will very rarely request dataspaces from `dm_phys` directly. Most programs do not need the guarantees that `dm_phys` is specifically designed to offer, and indeed, most client tasks will find it desirable to not be bound to those guarantees, specifically when it comes to issues of virtual memory and providing additional memory by employing secondary storage swapping methods. Instead, it is probable that a few other server tasks, which implement exactly such desired functionality, will request a large amount of physical memory from `dm_phys` at startup time in order to be able to back their own dataspaces somewhat. By the same virtue, most dataspaces exported by `dm_phys` are expected to be opened, but never closed until system shutdown time. Therefore, it seems undesirable to add much complexity to the `dm_phys` server by allowing for more complicated, albeit possibly asymptotically faster, coalescing strategies that are not expected to be used very often anyway.

With this reasoning, it seems advisable to use straightforward immediate coalescing, which is what is done in `dm_phys`.

4 Implementation

This chapter describes some of the issues that arose during the implementation of the `dm_phys` server. The issues addressed are

- notes on the data structures,
- notes on some interface functions, and
- miscellaneous notes.

4.1 Notes On the Data Structures

The three main data structures used in `dm_phys` are

- the memory region data type `region_t`,
- the dataspace data type `dataspace_t`, and
- the owner data type `owner_t`.

Each of these data structures is described in its own subsection.

4.1.1 The Memory Region Data Type

The memory region data type `region_t` is used for management of available blocks of memory. In `dm_phys` terminology, a memory *region* is a block of consecutive memory pages. Associated with each memory region are its base address and its size. Note that both size and base address are always aligned to the smallest page size available in the system; in other words, both the size and the base address are integer multiples of the smallest available page size.

The pool of available memory regions is kept in a double-linked queue. Since `dm_phys` supports requests for specific base addresses, the queue is kept sorted in ascending order by the base addresses of the regions so that the search for the requested base address is shortened. Therefore, the memory region data type needs to have another field that allows for double-linked list support.

4.1.2 The Dataspace Data Type

This data type is `dm_phys`'s internal representation of a dataspace handed out to a client task. In accordance with the dataspace properties, the dataspace data type contains variables that hold the memory region that is contained in the dataspace, the pagesize that is used for the dataspace, and the owning task of the dataspace.

The memory region and pagesize variables are straightforward, but the owner information is stored in a somewhat non-intuitive manner in order to allow for storing of sharer-sharee relationships between tasks. This is necessary to provide the *unshare* operation. The owner data structure is discussed in greater detail in the next section.

Dataspace structures in `dm_phys` are not stored by `dm_phys` directly, but rather by SawMill's handle library. The handle library allows for a very flexible management of access right and security policies as well as quick changes in those policies. The library takes a task ID and a dataspace structure, stores those pieces of information in a rather complex manner, and returns a unique *handle.* Each handle is valid only for the dataspace-client task ID combination it was generated with, and so implies access control with the help of the handle library. Later on, `dm_phys` can provide the library with a client task ID and its dataspace handle, and the handle library will return the proper dataspace structure. Thus, there is no need to have `dm_phys` keep an additional copy of the dataspace structure.

4.1.3 The Owner Data Type

This data type is used for storing sponsorship relationships between tasks. It consists of fields containing the owner's task ID, the owner's *sponsor task,* the owner's handle for the corresponding dataspace, and the owner's queue of subshares. Additionally, a queue link field is necessary, because owners are stored in double-linked queues.

The sponsor task s of a client task t is the task that has granted t access to the dataspace. Therefore, the owner task, meaning the task that has originally opened the dataspace, is its own sponsor task. If it shares the dataspace with any other task, then the owner task is the other task's sponsor.

Despite the name of the data structure, every task that has access to a given dataspace gets its own owner structure. This means that for every dataspace there possibly exist more than one owner structures. Therefore, these structures need to be organized.

In `dm_phys`, the owner structures are organized in the following fashion. The actual *owner task,* i. e. the task that originally created the dataspace, has its own owner structure entry in the dataspace data structure. All other tasks are stored in the owner's subshare queue, which is contained in the owner structure. Each of those tasks, having its own owner structure, also has its own subshare queue. If any task shares the dataspace with another task, then that other task's owner structure, in turn, gets inserted into the subshare queue of the sharer task. Thus, a tree-like structure is created. This is discussed in more detail in the next.

4.2 Notes On Some Interface Functions

In this section, some notes on the implementation of some of the interface functions are presented. These are mainly related to the treatment of shared dataspaces.

4.2.1 The Share Function

The share function's aim is straightforward: it provides a method for a task with access to a dataspace to grant the same access to another task. As indicated in section 4.1.3, in order to provide reasonable unshare functionality, it is necessary to store some information about sponsorship, or sharer-sharee, relationships. In `dm_phys`, this is done within the owner data structure.

The owner data structure provides a subshare queue and a pointer to the sponsoring task's owner structure. Abstractly speaking, the subshare queue is one level deeper than the owner data structure, while the pointer to the sponsor provides a pointer to the next-higher level. Furthermore, the highest level in this tree structure is the original owner task, i. e. the task that opened the dataspace. Therefore, the sponsor pointer is useless for the owner task, as it cannot point to a higher level. For the owner task, it is set to point to the owner task's owner structure, so in a way the pointer points to itself.

Whenever the share function is called, a new dataspace structure and, subsequently, by registering the dataspace structure and the share-receiving task ID with the handle library, a new dataspace handle are created, regardless of whether or not the receiving task does already have another handle for, and thus access to, the dataspace!

4.2.2 The Transfer Function

The transfer function is conceptually related to the share function discussed above. The main difference between those two functions is that while both hand over access rights to a dataspace to another task, the share method additionally retains these rights for the originating task, while the transfer function releases the dataspace handle, and thus gives up all access rights, for the sharing task. At the same time, the receiving task also acquires the sponsorship relationships of the originating task.

The implementation of this is fairly simple, if not obvious. When the transfer function is called, the only thing that is actually newly created is the dataspace handle for the receiving task, which is done by the handle library. Then the originating task's dataspace handle is deleted, and the new task's ID and dataspace handle is inserted in place of the old task's ID and handle in its owner structure. With this move, the owner structure that was originally created for the old task is recycled for the new task. The benefit of this is that no further variables or fields need to be changed, since all pointers to potential higher-level owner structures as well all pointers from owner entries in the subshare queue are still valid, because the owner structure itself has not changed.

4.2.3 The Unshare Function

The unshare function, as the name suggests, performs the inverse task to the share function. That means that while the share function gives access rights to other tasks, the unshare function takes away those access rights.

Its implementation is fairly simple. Each owner structure contains a subshare queue, which in turn contains a reference to every task that this task has shared the dataspace with. Thus, all that needs to be done is to step through the queue of subshares and subsequently take away the other tasks' access rights by destroying their respective dataspace handles and removing their owner structures. This, of course, is no guarantee that all access rights to the dataspace are actually revoked from all tasks: it is still possible that a task has been granted access rights to the dataspace not only by this task, but by yet another, completely unrelated task that did not receive its access rights from this task. Since the unshare function is supposed to be transitive, it needs to recursively step through the subshare queue, of course.

4.2.4 The Close Function

The close function is used for returning, or giving up, access rights to a dataspace. Just as the unshare function, the close function is transitive, so that if a task closes its access rights to a dataspace, then access is also taken away from all those tasks that the dataspace has been shared with by this task.

The implementation of the close function is as simple as that of the unshare function. In fact, the very same code is used, with the exception that at the end of the close call, the owner structure and dataspace handle for the closing task are also revoked. Furthermore, if the closing task is the owner task of the dataspace, then the dataspace structure itself is destroyed, and the memory region used for the dataspace is returned to the queue of available memory regions and, in accordance with the coalescing policy described above, immediately coalesced with the adjacent memory blocks if those are available.

Since a memory region carries its own base address and size, coalescing of two consecutive blocks is very simple: all that needs to be done is to increase the size of the memory block with the lower base address by the size of the other block, remove the latter block from the ready queue, and free its memory region structure.

4.3 Miscellaneous Notes

This section presents notes on miscellaneous topics that cannot gracefully be fit into any of the other sections. The topics described below are

- portability with regard to pagesizes, and
- memory acquisition at startup.

4.3.1 Portability With Regard To Pagesizes

Within each given hardware system, there is a well-defined set of pagesizes that the system supports. This set is entirely hardware-dependant, and handling of this set is therefore subject to portability considerations.

In order to keep portability at a maximum without giving up any comfort in implementation, the set of available pagesizes is organized in two constant arrays. One array contains the actual pagesizes, while the other array contains the logarithms to the base of two of the corresponding pagesizes. Both arrays are located in a header file and can easily be found and modified. Upon portation to another system with different pagesizes, all that needs to be done is adjustment of these arrays. All references to these constant arrays are dynamical.

In order for the dynamical access of pagesizes to work properly, the array of pagesizes needs to meet certain criteria. In particular, the following assumptions are made about the array:

- the first, i. e. zero-eth, entries of both arrays are marker entries and set to zero,
- the pagesize array is sorted in ascending order so that the zero-eth element is equal to zero, the next element is the smallest actual pagesize available in the system, and the last element is equal to the largest available pagesize,
- the pagesize logarithm array is also sorted in ascending order so that the zero-eth element is equal to zero, the next element is the logarithm of the smallest pagesize, and so on.

4.3.2 Memory Acquisition At Startup

This section briefly describes how `dm_phys` acquires memory at startup time.

At this point, `dm_phys` is loaded as a standalone server task at system boot-time. This means that it gets its resources from the resource manager, which in turn is controlled by its configuration file. In this configuration file, the resource manager is told which resources to give to which tasks. Among these resources is the range of memory that may be granted to each task by the resource manager.

Because of this mechanism, `dm_phys` has no way of knowing which pages of memory it is free to use. Therefore, it is necessary to either hard-wire this information into the `dm_phys` source code, which is clearly unfavorable, or have `dm_phys` scan the entire range of memory, trying to obtain every single page from the resource manager.

The latter approach is what is being done in `dm_phys`. At startup time, the entire range of memory — with the exception of the first megabyte of memory — is scanned for availability. The lower 1024 kilobytes contain memory designated for special purposes and accessible for all tasks, but since this range is special-purpose, `dm_phys` should not manage it as if it were regular memory. The upper limit of the memory range being scanned is not the upper limit of the physical memory installed in the system, but a constant hard-wired in the `dm_phys` header file, and taken from the resource manager source. Currently, the high memory limit is 128 megabytes.

`dm_phys` steps through the memory range specified above at steps of the smallest available pagesize, which is the entry at index one of the constant array of pagesizes, as discussed above. At every step, `dm_phys` checks the alignment of the memory address and requests from the resource manager the largest page size that is available in the system and aligned to the current address. If it gets the requested memory page, then it skips directly to the next address aligned to that pagesize. If not, then the next-smaller page size is tried, and so on, until finally either a page of some size is obtained or index zero is reached in the array of pagesizes. In that case, `dm_phys` again steps by the smallest pagesize.

5 Validation

In order to confirm both the functioning and the desired performance of the `dm_phys` dataspace manager implemented for this study thesis, a series of tests has been performed. These validation and timing tests are described in this chapter.

5.1 Test Setup

It can quickly be seen that it is not trivial to devise a test that performs a meaningful comparison of the Linux memory allocation/deallocation procedures and the `dm_phys` dataspace management system. The reason for this is the entirely different structure of the underlying operating systems: while Linux consists of a monolithic kernel that does not offer a concept comparable to dataspaces, SawMill is based on the microkernel operating system L4. The Linux `malloc`/`free` operations are therefore not entirely comparable to SawMill dataspace `open` and `close` operations. It is even more difficult to come up with a memory allocation/deallocation scheme that is actually significant in terms of performance measurements: simply repeating the same allocation/deallocation sequence obviously is not enough, and it is clear that in order to obtain significant data, random allocations and deallocations need to be performed in some arbitrary order.

Furthermore, matters are complicated by the fact that SawMill/L4 is an operating system that is still in the development stage. Thus, only a fraction of the functionality desired for more in-depth testing is available.

The test used for validation consists of five steps.

1. Allocate n dataspaces (for SawMill) or blocks of memory (for Linux) of size $s_i, i \in \{1, \ldots, n\}$ one after the other without deallocating any of these blocks. Thus, after this step, a total of n dataspaces or memory blocks are allocated, consuming $\sum_{i=1}^{n} s_i$ bytes of memory.
   It can be assumed that all allocated memory is consecutive.

2. Shuffle the pointers to the dataspaces/memory blocks.

3. Deallocate the first m dataspaces/memory blocks.
   Since the pointers to the dataspaces/memory blocks have been shuffled, essentially m randomly-positioned dataspaces/memory blocks are now deallocated, thus introducing (external) fragmentation.

4. Deallocate the next dataspace/memory block, then reallocate one memory block. Repeat n times.

This loop keeps deallocating randomly-placed dataspaces/memory blocks, while at the same time allocating a new dataspace/memory block for each deallocated one. Thus, after any loop iteration, there are exactly $n - m$ dataspaces/blocks of memory still allocated, while their base addresses are more or less random.

5. Deallocate the remaining $n - m$ dataspaces/memory blocks.

Throughout the whole procedure, the number of processor cycles used by each allocation or deallocation call is tracked and the corresponding minima, maxima and averages are updated.

The sizes s_i of of the dataspaces/memory blocks should obviously be random, for if all dataspaces/memory blocks were of the same size, then no fragmentation would ever occur. This is not a very realistic condition, and figures resulting from such a setup would not be very convincing. For the tests performed, the sizes s_i are random values from the set $\{i \times 4096 | i \in \{1, \ldots, 10\}\}$. In other words, the block sizes are integer multiples of 4096 bytes, the smallest L4/Intel pagesize available, and the maximum block size is 40960 bytes.

Obviously, the whole test is parametrized by two variables: the number of simultaneously allocated dataspaces/memory blocks, n, and the number of dataspaces/memory blocks deallocated before reallocating further blocks, m. For the validation process, a series of tests have been conducted. The test process has been performed for each combination of m and n, where $n \in \{100, 500\}$ and $m \in \{m \times (i \div 20) | i \in \{1, \ldots, 19\}\}$.

The test series were conducted both for `dm_phys` on SawMill/L4 as well as for plain Linux on the same machine, an Intel Pentium 2 with 300 MHz and a total of 128 megabytes of physical memory installed. `dm_phys` was compiled using the IDL 4 compiler.

5.2 Measurements

This section gives the results of the test series conducted for validation and performance measurement, as explained in the previous section.

For each combination of (n, m), the minimum, average and maximum number of processor cycles used for both an open and close call are presented. As the minimum, maximum and average numbers of cycles are apart by orders of magnitude, there is a separate graph for each set of values. This makes the graphs easier to read.

As can be seen in figures 5.2 and 5.2, Linux clearly outperforms `dm_phys` in best-case scenarios. The minimum number of cycles used by a Linux allocation is about 180, without regard to the number of simultaneously allocated memory blocks. The minimum for a deallocation operation in Linux is roughly the same. The same values for `dm_phys` under SawMill are much greater: the minimum cost of an open operation is around 3290 cycles for the 100-dataspace test and around 3480 cycles for the 500-dataspace test, while the minimum cost of a close call is always about 2450 cycles. All these values have a very small variance of less than five percent.

	Open			Close		
	Minimum	Average	Maximum	Minimum	Average	Maximum
100-Allocation Test						
Sawmill						
Minimum	3247	3467	4989	2412	5838	9493
Average	3290	3591	25791	2485	6034	13363
Maximum	3327	5294	346824	2570	6321	19965
Linux						
Minimum	179	4111	9989	167	423	15206
Average	183	5342	17186	190	564	31035
Maximum	190	6022	47473	212	737	61448
500-Allocation Test						
Sawmill						
Minimum	3417	3481	7605	2409	3978	8714
Average	3479	3565	46296	2446	4059	13639
Maximum	3532	3842	349939	2490	4150	17231
Linux						
Minimum	179	6013	18930	165	241	85491
Average	181	7128	36753	196	461	168765
Maximum	187	8269	92676	221	514	255727

Table 5.1: Measurement results for the 100-allocation and 500-allocation test series.

The respective maximum values show a much greater variance. This can be seen in figures 5.2 and 5.2. The maximum number of cycles for a Linux open operation range from about 10000 to around 48000 for the 100-allocation test; the average is around 17000. For the 500-allocation test, the values are nearly doubled: the range is from around 19000 to about 93000, with the average around 37000. Again, `dm_phys` is outperformed: costs for the open operation ranges from about 5000 to around 350000 cycles, with the average around 26000 for the 100-allocation test and around 46000 for the 500-allocation test. Looking at the graphs, it becomes obvious that those measurements around 350000 cycles are very rare; the average value is much more accurate.

This picture is somewhat reversed when the maximum cycles for close operations are considered. Now, `dm_phys` outperforms the Linux memory management: Both for the 100-allocation and the 500-allocation tests, the maximum number of cycles used by the close call ranges between around 9000 and 19000 cycles, with the average at around 13500. Linux, on the other hand, requires between 15000 and 61000 cycles for the 100-allocation test, with an average of 31000, and between 85000 and 255000 cycles for the 500-allocation test, with an average of 170000.

Finally, when we regard the average case, `dm_phys` clearly outperforms Linux for open operations, but is beaten by Linux for close calls. The graphs of these values can be found in figures 5.2 and 5.2.

The measurements discussed above are compactly shown in table 5.2.

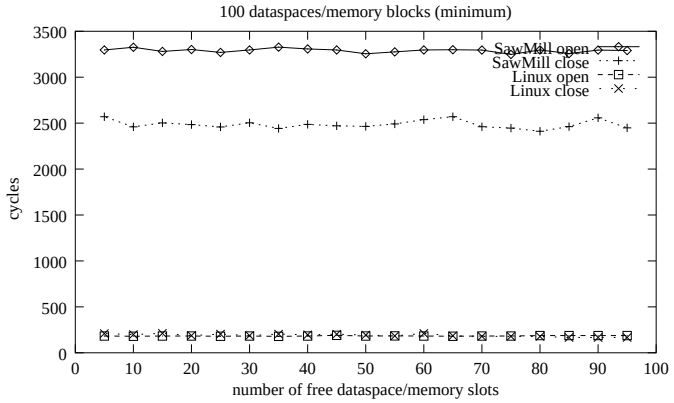

Figure 5.1: Minimum number of cycles per open/close operation on a random-size dataspace/block of memory for a test set of 100 dataspaces/blocks.

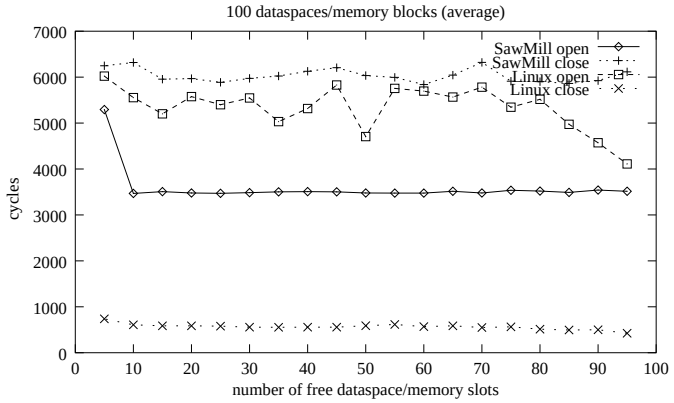

Figure 5.2: Average number of cycles per open/close operation on a random-size dataspace/block of memory for a test set of 100 dataspaces/blocks.

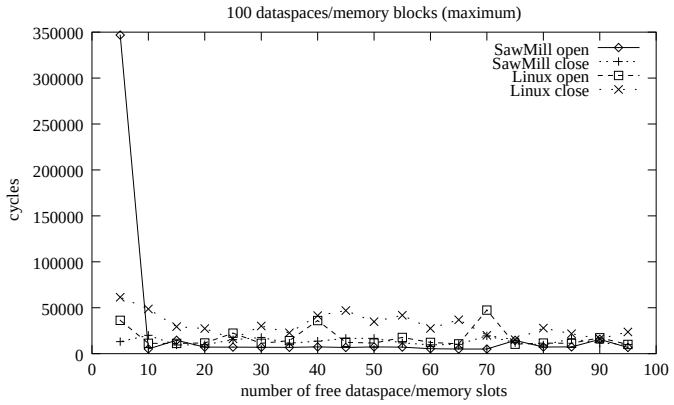

Figure 5.3: Maximum number of cycles per open/close operation on a random-size dataspace/block of memory for a test set of 100 dataspaces/blocks.

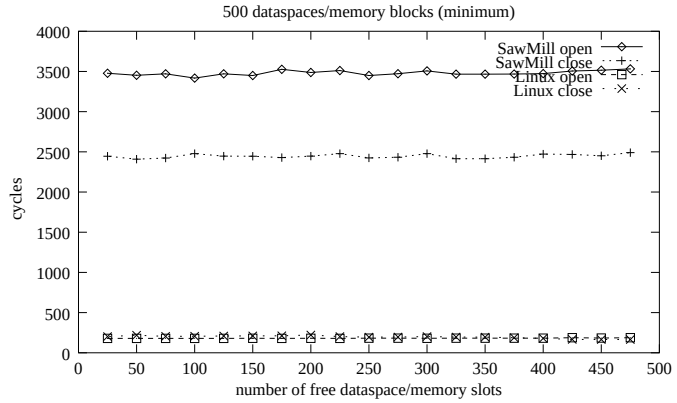

Figure 5.4: Minimum number of cycles per open/close operation on a random-size dataspace/block of memory for a test set of 500 dataspaces/blocks.

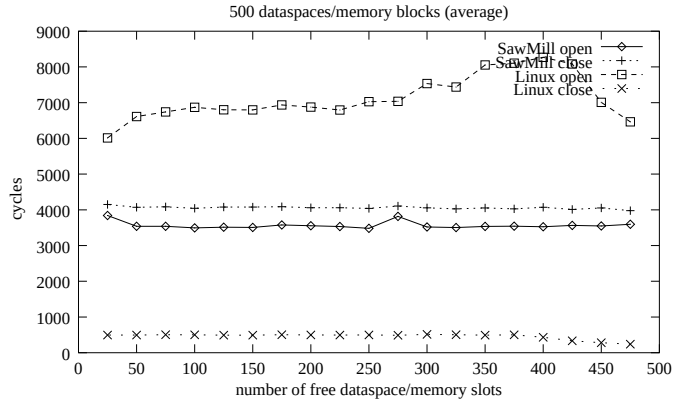

Figure 5.5: Average number of cycles per open/close operation on a random-size dataspace/block of memory for a test set of 500 dataspaces/blocks.

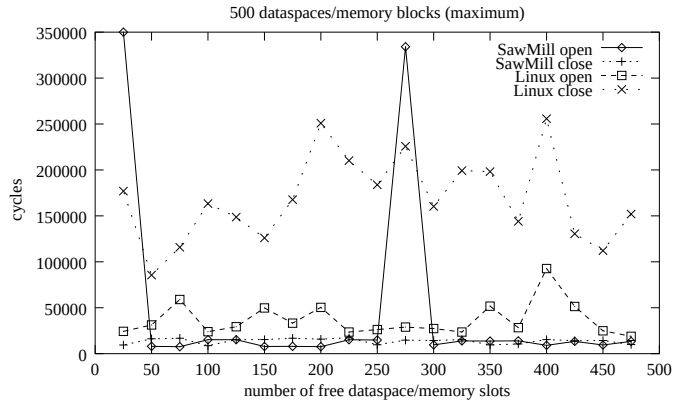

Figure 5.6: Maximum number of cycles per open/close operation on a random-size dataspace/block of memory for a test set of 500 dataspaces/blocks.

5.3 Interpretation

In general, the measurements confirm that the `dm_phys` dataspace manager performance is reasonably comparable with that of the Linux memory management subsystem.

The average cost for opening a dataspace is a little lower than for allocating a block of memory in Linux. This was expected, as the SawMill dataspace model favors simple, fast and lean solutions; `dm_phys` clearly falls in this category. The memory management in Linux, on the other hand, has more tedious and time-consuming tasks to do than the simple `dm_phys` model. Thus, Linux malloc calls tend to be more costly simply because they do more work.

On the other hand, Linux clearly outperforms `dm_phys` when it comes to closing a dataspace or deallocating memory. This is due to the delayed coalescion strategy used in Linux, as in many unixoid systems. As described in the design section of this thesis, it is envisioned that dataspaces managed by `dm_phys` are rarely closed, but rather kept in use during the entire system uptime. Therefore, it was not a design goal for `dm_phys` to be top-performing with regard to close calls. Instead, increased performance was traded in for increased simplicity in order to keep things simple and less error-prone.

In total, the measurements show that the projected goals for the `dm_phys` project have been reached.

6 Conclusion and Future Work

There are quite a few tasks that can be turned to in the future. The following list is by no means complete, but gives a feeling of the direction in which further effort could be directed.

Performance improvement: While this probably is not the most important effort to make, it may turn out to be good to have an even better-performing physical dataspace manager. This is especially true if higher-level dataspace managers take a while to be written: in that scenario, it might become important to at least have reasonable dynamical access to physical memory, and the precondition that certain performance aspects need not be as important to `dm_phys` design because of relatively few open/close interactions may not hold. Therefore, it might become desirable to trade in some of `dm_phys`'s simplicity in favor of better performance parameters.

One very interesting and promising approach for performance improvement is the use of tree structures wherever list or queue structures are used in the current implementation. In particular, this concerns the queue of available memory blocks; furthermore, a linear data structure can be found in the owner data type.

Higher-level RAM dataspace managers: This seems to be the most important future project. In order to help SawMill develop into a more realistic, usable operating system, it is necessary to implement memory access with features that are commonly present in today's operating system. The canonical example for this is swapping on secondary storage, but other tasks, such as avoidance of external fragmentation, development and implementation of more complex allocation and coalescing strategies optimized for frequent access within the SawMill environment, and implementation of support for memory compaction seem to be equally important.

Extension of `dm_phys`'s domain: To this point, `dm_phys` is specifically designed to manage the RAM installed in the system. It is thinkable to extend `dm_phys` so that it is also able to manage other forms of physical memory, such as the mainboard ROM, video framebuffers, or PCI card ROMs or buffers.

Similar dataspace managers: Just as the previous example, it is possible to think of a `dm_phys`-like but separate dataspace manager for tasks like management of ROMs, framebuffers and other similar devices.

Cryptographic dataspace manager: Finally, another potential future project is the design and implementation of a cryptographic dataspace manager. Such a dataspace manager's need for secure physical memory has originally lead to the implementation of `dm_phys`, so the originally envisioned `dm_crypt` seems to be a natural follow-up project to `dm_phys`.

In any case, there is much work left to be done in the general domain of dataspace managers in SawMill. The dataspace concept is a very flexible framework, and it is certain that it has much potential left to be exploited.

Bibliography

[AH99] Alan Au and Gernot Heiser. *L4 User Manual Version 1.14*. School of Computer Science and Engineering, University of New South Wales, Sydney 2052, Australia, March 15, 1999.

[APJ+99] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The sawmill framework for virtual memory diversity. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, NJ, March 1999. Available from `http://www.research.ibm.com/sawmill`.

[Del99] Luke Deller. Loading and debugging tasks in sawmill. School of Computer Science and Engineering, University of New South Wales, Sydney 2052, Australia, September 1999.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. In *ACM SIGOPS European Workshop*, September 2000.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Der LaTeX-Begleiter*. Addison-Wesley (Deutschland) GmbH, 1st edition, 1994.

[GRM97] Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LaTeX Graphics Companion*. Addison-Wesleyi Longman, Inc., 1st edition, 1997.

[Göt01] Stefan Götz. The bull-server. Internal documentation, 2001. System Architecture Group, Faculty for Computer Sciences, University of Karlsruhe.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Inc., 3rd edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, Inc., 3rd edition, 1997.

[RK00] Mehran Rezaei and Krishna M. Kavi. A new implementation technique for memory management. In *Proceedings of the 2000 SoutheastCon*, Nashville, TN, April 2000. The University of Alabama in Huntsville.

[Sta98] William Stallings. *Operating Systems — Internals and Design Principles*. Prentice-Hall, Inc., 3rd edition, 1998.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*. Springer, 1995.