Diploma thesis

Verification of Buffer Management in L4 Microkernel

Alexandra Tsyban
CONTENTS

1  The concept of virtual memory  2
  1.1  Paging  2
  1.2  Address translation  3
  1.3  Multilevel Page Tables  5
2  memory mapping in the L4 microkernel  7
  2.1  The L4 abstractions and mechanisms  7
  2.2  Mapping Database  11
  2.3  Structure of the Buffer Pool  13
  2.4  Function description  16
    2.4.1  Mdb_alloc_buffer()  16
    2.4.2  Mdb_free_buffer()  18
  2.5  Criteria  19
3  Elaboration of the formal model  22
  3.1  Translation of data types  23
  3.2  Translation of the functions  27
    3.2.1  The mdb_alloc_buffer function  27
    3.2.2  The mdb_free_buffer function  33
  3.3  Dynamic model  39
  3.4  Initial condition  48
  3.5  Input condition for the mdb_free_buffer function  49
4  Proving the correctness of buffer management  50
  4.1  Proof of some lemmas  51
References  60
Computers never seem to have enough memory, no matter how much memory is installed. One of most complex and difficult tasks of any operating system is managing the limited physical memory of a computer. This challenge is increased by the fact that operating systems must divide physical memory among many processes that might be running simultaneously, giving each process an appropriate memory share.

In order to solve the problem of managing the limited physical memory, a technique called virtual memory was developed. A resource is called virtual, if it seems to have some properties but it does not really have them. So, for example, virtual memory creates the illusion that a computer has more memory than it actually has. This allows programmers to write applications independent of the real amount of memory.

Obviously, virtual memory cannot directly correspond to physical memory, therefore parts of the memory fragments must be stored sometimes in the external memory.

The most widely used virtual memory mechanism is called paging.

1.1 Paging

The virtual address space of each process is divided into parts of the same size called virtual pages. This size is fixed for the given system. The physical memory is also divided into parts of this same size which are called physical pages. The page size is usually selected equal to a power of two: 512, 1024 Bytes etc., this allows for a simple the mechanism of address translation.

The operating system creates for each process an information structure -
the page table. This table contains information about correspondence between virtual and physical pages for the pages loaded in the main memory, or a mark that the virtual page is located on the disk. The page table also contains control information, such as the page modification bit, the page unloading bit (offloading of some pages can be prohibited) and others, which are used by the mechanism of virtual memory.

At each memory access, information about the virtual page is extracted from the page table. If the given virtual page is in the main memory, a translation of the virtual address into the physical address is performed. If the necessary virtual page is located on the disk at the moment, the so-called page fault interruption occurs. The executed process enters the waiting state. The handler of the page interruption finds the required virtual page on the disk and tries to load it into the main memory. If there is any free physical page in the memory, loading is performed immediately; if there are no free pages in the main memory, the handler offloads some page from the memory using a built-in algorithm in order to decide which page to offload.

1.2 Address translation

Let us consider the mechanism of translation of the virtual address into the physical address for a page-organized memory (figure 1).

The virtual address is represented as a pair \((p, s)\), where \(p\) is a number of a virtual page which is used as an index in the page table, and \(s\) is an offset inside the virtual page. As the page size is equal to a power of two \((k)\), the offset \(s\) can be obtained as the \(k\) lowest bits in the binary representation of the virtual address. The remaining higher bits are the binary representation of the page index \(p\).

At each access to the main memory the following operations are performed:
1. determining the address of the necessary entry in the page table using the initial address of the page table (the content of a special processor register) and the virtual page index (higher bits of the virtual address),

2. getting the address of the physical page from this entry,

3. computation of the corresponding physical address using the address of the physical page and the offset (lower bits of the virtual address).

The fact that the page size is equal to a power of two allows us to apply simple concatenation instead of a more time-consuming addition for address computation; that reduces the amount of time for obtaining the physical address. This is important because productivity of a paged system depends on the time needed for the processing of page fault interruptions and the translation of the virtual address into the physical address.
1.3 Multilevel Page Tables

The purpose of the page table is to map virtual pages to physical pages. A single page table can consist of an array of fast CPU registers with one entry for each virtual page or can be entirely located in the main memory.

The page table described above has one level: the content of an entry is a physical address of a physical page. But in order to get around the problem of having huge page tables in the memory all the time, many computers use a multilevel page table. In this case page tables which do not contain any valid entries are stored on the disk.

For example, in the common case of a two-level page table, a virtual address has three fields: the page directory index, the page table index, and the offset (figure 2).

![Address translation in 2-level paging system.](image)

Figure 2: Address translation in 2-level paging system.
The two-level page table has the following structure: there is one first-level table called page directory, which contains addresses of second-level page tables. Each second-level page table contains addresses of physical pages. The page table and the page directory entries have the same format as in a one-level organization.

The operation system uses each index in a three-step address translation process. First, an operating system gets the address of the corresponding page table from the page directory using the page directory index. Then, using the address of the page table and the page table index, the address of the physical page is obtained. Finally, by adding the offset, the physical address is computed.

The two-level page table system can be expanded to three, four or more levels, which we will not consider in this thesis.
L4 is an operating system microkernel developed and implemented by Jochen Liedtke. The basic idea is that the kernel implements only a minimal set of abstractions on which operating systems can be built flexibly. Since an operating system should provide secure sharing of resources, the word minimal is understood as only with respect to functionality: “A service (feature) is to be included in the microkernel if and only if it impossible to provide that service outside the kernel without loss of security.”

2.1 The L4 abstractions and mechanisms

The main concepts in L4 are threads and address spaces. A thread is the basic execution abstraction. An address space contains all data which are directly accessible by a thread. An address space and threads associated with it are referred to as a task. All threads belonging to one task share the same address space. There is a one-to-one correspondence between tasks and address spaces. Creating and deleting tasks implies creating and deleting address spaces.

There is the fixed number of tasks which can be in active or inactive state. Thus, strictly speaking, tasks are not created or destroyed, but their state is changed from inactive to active or vice versa. At system boot time there is only one active task called $\sigma_0$. The other tasks are inactive. Inactive tasks have no address space and do not use any system resources. An active task has a fixed number of threads, and at least one of them is active.

Any task (parent) can create other tasks (children) and have some limited control over them. The parents and children form a hierarchy of tasks. In order to control the information flow between different address spaces and ensure security,
the Clans & Chiefs mechanism is used.

The basic idea of this mechanism is simple. All tasks created by one task belong to one clan and can communicate with each other. The task-creator is their chief. If a thread of one task wants to send a message to a thread of another task which does not belong to the same clan, that message must be passed through the sender’s chief and receiver’s chief. This process is depicted in figure 3, where tasks (circles) are united in clans (ovals) with their chief (circles on top). Arrows represent the transfer of a message: the bold arrow is the intended dispatch, and the dotted arrows are the real way of the message.

![Figure 3: Message redirection by clans & chiefs.](image-url)

In case of redirection of a message, the message is sent from the sender to its chief, the latter sends the message to the chief of the receiver or, if necessary, to his chief and so on until the message reaches the recipient. All chiefs have full control over messages. They have the right not to forward the message or to
Protection in L4 is based on address spaces. An address space is just a mapping that associates each virtual page with a physical page frame or marks it as non-accessible. A thread can only access data in its own address space.

L4 supports the recursive construction of address spaces. For such a mechanism, we need one starting address space. This address space is called $\sigma_0$ (like the task associated with it). As mentioned before, it is created at boot time and represents the physical memory (except the parts reserved for kernel use). Further address spaces are created by the following operations:

- granting,
- mapping,
- unmapping (flushing).

These three operations are performed on pages.

**Map.** One thread can map some its pages into the address space of another thread. These pages can then be accessed in both address spaces. The owner, called *mapper*, retains all rights over the pages and can revoke mappings at any time by the unmapping operation. The thread accepting the pages is called the *mappee*.

**Unmap.** This operation is the opposite of mapping. A thread can unmap any of its pages from the address spaces they were mapped to before. Let us denote a thread initializing the unmap operation as *unmapper*. The unmapped pages remain accessible in the unmapper’s address space, but they are removed from all other address spaces that received the page directly or indirectly from the unmapper.
Grant. One thread can also grant some of its pages to another address space. In analogy to a mapping operation, the owner is called granter, and the accepting thread grantee. In contrast to mapping, the granted pages are removed from the granter’s address space. Thus, the grantee gets full control over these pages; although the parents of the granter may still revoke the page at any time (as it was mapped by them before).

Any thread can grant, map and unmap only pages that have already been mapped into its own address space. These are all just operations on page tables - no data is actually copied. Figure 4 shows an example using the above mentioned operations. Arrows represent the operations, and thin lines represent the resulting mapping. A thread in address space $A$ maps a page into address space $B$, which further maps it into address space $C$. At the unmapping by $A$, this page is removed from both address spaces $B$ and $C$. However, if $B$ unmaps the mapping, the page will be removed from address space $C$, but it will remain in $A$’s address space. After mapping of a page from $A$’s address space into $D$’s address space, it is granted to address space $C$. As a result, this mapping disappears in $D$.

The three basic operations are secure in the sense that since they work on virtual pages (page tables), the data in the physical memory not reserved by L4 is
not copied and cannot be changed by these operations. Obviously, it is not secure in the sense that when a page is mapped to another address space, the owner does not know if that page will be mapped further.

2.2 Mapping Database

Address spaces in the L4 are constructed by mapping sections of other address spaces, as described in the last section. All information about current mappings in the system is stored in a special kernel data structure - the Mapping Database (MDB). The MDB is closely related to the page tables. It includes data structures to keep track of sharing between address spaces. Those data structures implement recursive address space.

Let us consider the MDB in detail. The MDB is represented as a tree, which contains nodes of different types. The number of node types is fixed, and their size depends on the architecture. There are the following node types:

- mapping node, representing a mapping of one page to some thread;
- root node, representing one physical page (4 MB or 4 KB);
- dual node, used as a connecting element.

Initially, there exists only the address space $\sigma_0$ corresponding to physical memory space of size 4 GB. Thus, the MDB contains only one element at the top of the mapping tree (figure 5). The address space $\sigma_0$ is splintered on 1024 pages of size 4 MB. This is represented as an array of root nodes, one for each page. In order to construct an address space of a just created task, $\sigma_0$ can map some of its pages to the new address space, for example the third page from $\sigma_0$ to $D$’s address space (figure 5). In this case, one mapping node is added. This node
represents the given page in the address space of the mappee. It is connected to root node representing a page which is mapped.

There is a second possibility: mapping not of the whole page, but only some part of it. For this purpose, one page of 4 MB is represented as array of root nodes, describing 1024 pages of size 4 KB. In figure 5 $\sigma_0$ maps part of its second page to the address spaces of $B$ and $C$.

In order to unite these two possibilities for one node which represents the mapping of that page in the mapper’s address space, dual node is used; thus, this page can be mapped as a whole or only some parts of it. The third page of $\sigma_0$ in figure 5 is an example for this situation: it is completely mapped to the $D$’s address space, and a part of it is mapped to the address spaces of $E$ and $A$.

All threads accepting the same page from the same mapper are represented as a list of mapping nodes connected to their mapper. In figure 5 an arrow indicates the beginning of the list of some mapper, and lines connect nodes of these threads in a list. Thus, the unmapping operation from this thread is executed.
as deleting that list from the MDB.

At runtime of the operating system, many nodes for the MDB are created and deleted. A piece of memory for storing one such a node is called a buffer. Buffers are allocated from a special memory space, the Buffer Pool. From now on, we denote a buffer which is allocated as *busy* and a buffer which is not yet allocated as *free*.

2.3 Structure of the Buffer Pool

For each system there exists a certain number of admissible buffer sizes. This number depends on the sizes of the physical memory pages supported by the operating system, and on the sizes of node structures in the MDB that can be requested to be allocated.

There are two types of elements to construct the Buffer Pool:

– **page**

A page is a piece of memory, whose size is equal to one physical memory frame (4 KB for the x86 architecture). Each page contains a certain number of buffers of one of the possible sizes and a structure, which allows to organize pages in lists. These lists of pages contain only the pages which have at least one free buffer.

– **headers**

Headers contain information about a buffer size and a pointer to a list of pages with buffers of the same size.

Figure 6 shows the structure of the Buffer Pool. It is organized as an array of headers and a set of pages which are organized in doubly linked lists. The num-

13
ber of headers is equal to the number of buffer sizes supported by the operating system. Thus there is only one header and only one page list for each buffer size.

Initially, the Buffer Pool consists only of the array of headers. For efficiency reasons, the first element defines the buffer size that will be allocated most frequently. The array is terminated by a header for the zero buffer size.

Each header contains the following elements:

- **size**

  The buffer size which the header is representing. It is needed in order to find the page list with buffers of this size.

- **list_of_lists**
A pointer to the list of pages. If it points to NULL, there is no page with free buffers of this size.

- **max_free**

  Determines the maximal number of buffers which can be allocated in one page.

  In order to find a free buffer in a page quickly, all free buffers in one page are linked in a list. Each buffer contains a `next` pointer which is used for joining the free buffers in a list.

  As mentioned before, the page also contains a management information structure which has the following elements:

  - **freelist**

    A pointer to the first free buffer in the page (the first element of the list of free buffers).

  - **num_free**

    The number of free buffers in the page. It is used for checking whether it is necessary to insert the page into or remove it from the list of pages. Obviously, if it is equal to the maximal number of buffers in the page (the `max_free` element in the corresponding header), then all buffers are free; if it is equal to zero, then all buffers are busy.

  - **next_freelist, prev_freelist**

    These are pointers that join the pages in a doubly linked list.

  - **bl**

    A pointer to the corresponding header.
The buffers and the memory information are located in the page according to the following scheme:

- the first 20 bytes are occupied by the management information;

- the rest of the page is used to store buffers; therefore it is divided into parts of the corresponding buffer size. The last buffer is aligned to the last address of the page, so there might be unused space between the management information and the first buffer (figure 7).

![Figure 7: Created page.](image)

2.4 Function description

There are two functions to work with the Buffer Pool: `mdb_alloc_buffer()` and `mdb_free_buffer()`. They can be found in the file "mapping_alloc.c" of the source code of the L4 microkernel.

2.4.1 Mdb_alloc_buffer()

This function is used for buffer allocation. It takes as an input parameter the size of a buffer to be allocated. First, it looks for the header representing the
buffer size to be allocated. As mentioned above, this is done with the help of the size field of the headers and by comparing the input size and the field value.

After this we can have two possible situations (figure 8):

- The size is not supported by the system. The function returns an error message that this buffer size is illegal and terminates.

- There exists a header for such a buffer size. Now again we have two cases:

1. The pointer of the header points to NULL.

   In this case, the function has to create a new page by the system call \texttt{kmem\_alloc()} which allocates a new piece of memory of the necessary size, initializes it and adds it at the beginning of the list of pages of that header. Initialization of a page includes the following operations:

   - division of the page space into buffers;

   - joining all buffers using the next pointers;

   - setting the freelist pointer to the first buffer in this page;
– initialization of the number of free buffers in this page with maximum value (the max_free field of the corresponding header);

– connecting the management structure with the corresponding header (the bl pointer to a header);

– updating the header information (the header should point to the new page).

After these operations the pointer of the header does not point to NULL. The function proceeds as in the second case.

2. The pointer of the header to the list of pages does not point to NULL.

The function allocates the first free buffer in the first page of the list, which is the one the freelist element from the management information of this page points to, sets this pointer to the next free buffer and decrements the number of free buffers in the page.

We check if all buffers in the page are busy (num_free is equal to zero). In this case, the corresponding page is removed from the list of pages.

As the result, the function returns a pointer to the allocated buffer.

2.4.2 Mdb_free_buffer()

This function is used to deallocate buffers. It takes as input parameters a pointer which points to an allocated buffer and the buffer size. First, the function computes the base address of the page which contains this buffer and gets the corresponding header (with help of the bl element from the page management information). Then, the function adds the buffer to the free buffer list in the current page and increments the counter of free buffers. After that, there are two special cases:
1. This buffer is the first free buffer in the page (num_free is equal to one). The function moves this page to the beginning of the list of pages.

2. This buffer is the last free buffer in the page (num_free is equal to the max_free element of the corresponding header). The function removes this page from the list of pages and deletes it by the corresponding system call kmem_free() deallocating the piece of memory, the size and the first address of this page are passed as input parameters of the call.

2.5 Criteria

Having the information about the construction of the Buffer Pool and the functions, we can formulate the criteria about correct functions behavior with the Buffer Pool. All those criteria can be divided into four groups.

The first group concerns the allocation of buffers in a page and the management information. From the source code we know that the whole page is allocated by the system call. But the location of the buffers inside of a page is the task of the mdb_alloc_buffer function. Thus, we have to check that all buffers and the management information do not overlap and are located inside of the necessary page.

**criterion 1.** Two buffers never share the same address space.

**criterion 2.** Buffers do not share their address space with management information.

**criterion 3.** The buffers are located inside the page.

**criterion 4.** A buffer of a particular size is allocated in the page which contains buffers of this size.

The information about free buffers in the page should really reflect the
current state (of the page):

**criterion 5.** The amount of memory space in the page which is not used by the memory information and busy buffers is large enough to allocate `num_free` buffers.

Also, all pointers to the buffers should point correctly, namely:

**criterion 6.1.** The pointer to the first free buffer points to a buffer inside the page.

**criterion 6.2.** Each free buffer in a page except the last buffer (in the list of free buffers) points to a free buffer in the same page.

The criterion about the `freelist` pointer and the `num_free` counter:

**criterion 7.** The pointer to the first free buffer in the current page points to `NULL` if and only if there are no free buffers in this page.

The second group concerns a list of pages. It describe only the pages with buffers of the same size linked in the list, and this list should not contain pages without free buffers.

**criterion 8.1.** If the pointer of the header does not point to `NULL`, it points to an existing page.

**criterion 8.2.** If the pointer to the next page with free buffers does not point to `NULL`, it points to an existing page.

**criterion 8.3.** If the pointer to the previous page with free buffers does not point to `NULL`, it points to an existing page.

**criterion 9.** The pointer of the header of a particular buffer size points to the empty list if and only if there are no pages with free buffers of the same size.

**criterion 10.** The pointer of the header of a particular buffer size points to the list which consists only of pages with buffers of this size.
criterion 11. A list which is pointed to by a header does not contain any pages without free buffers.

criterion 12. A page with a particular buffer size contains a pointer to the header which defines the same size.

The third group of criteria concerns rational memory usage. The new page should be created only if it is necessary and it will be freed as soon as it is possible to free it.

criterion 13. A new page for a buffer size will not be created if there is at least one free buffer of the same size.

criterion 14. If the page does not contain any busy buffers, the memory space of that page is freed.

The fourth group concerns the results of the functions.

For the mdb_alloc_buffer function:

criterion 15. After the execution of the mdb_alloc_buffer function, there exists a buffer which can be used to store an MDB element of the specified size.

For the mdb_free_buffer function:

criterion 16. If the mdb_free_buffer function is invoked in order to deallocate a buffer, that buffer is marked in the Buffer Pool as free after execution of the function.
The present chapter is dedicated to the verification of the source code, dealing with buffer management which is represented by the two functions described above. Theorem provers typically support Higher-Order Logic. Therefore, it is enough to formally describe the specification in order to use it in any verification system. In general, the verification process includes the following significant steps:

- creating data types in formal language;
- elaborating the memory model;
- translating the C code of these functions into formal language;
- formalizing criteria;
- searching for methods to prove the formally described specification;
- proving.

Indeed, this list is not complete. This is due to the fact that Higher-Order Logic is not intended to prove programs written in C. Therefore, in the translation of C-code there are some difficulties which can be resolved, but the resulting specification is not very close to the original code. Thus, in this situation we need to analyze the created model and to check whether it satisfies all the necessary requirements.

The specification of the data types and functions is represented in the following section using a formal description.
3.1 Translation of data types

The first two steps of the verification process cannot be separated from each other, since during constructing of the model it might be necessary to change an existing data types and its specification.

Any composite type (such as a structure in C) can be realized as a record type, which is a tuple with named elements. By this method the translation of the following structures can be performed:

– structure for the header,

– structure for the management information,

– structure for free buffers linking.

Here we encounter the first problem: pointers. There is no such type in plain mathematics. Therefore, all objects of each data type are gathered in one array. We can model an access to an object by a pointer by its array index. There exist three types of arrays in the memory:

– an array of headers for the lists of pages with free buffers;

– an array of pages with buffers;

– and an array of buffers inside of one page.

The reason why there are separate arrays for each page will be made clear below.

There exists one more difficulty: it is not enough to model objects only by their memory content because they might actually overlap in physical memory.

Therefore, we have to determine which objects need to store their memory address.
All headers are present in the memory before the first invocation of any function and are allocated in a memory space which is separated from the memory used by functions (we assume that system calls of the kernel are correct). So the proof that buffers cannot share the address space with any of the headers is not needed. Thus headers do not need addresses.

The pages are also allocated by the system call and each page which is returned by this function is not allocated in the same address space with another page. Their starting addresses have no importance for the described functions, so we do not need to know the address of a page.

In order to prove the correctness of the functions concerning buffer allocation it is necessary to know the address of a buffer. Since two pages do not share their address space with each other, two buffers from different pages cannot be allocated at the same address and one can just use the offset of a buffer inside the page as its address.

Therefore, only buffers have addresses.

Now let us consider the formal representation of all data types. During the function specification it is convenient to unite all types in one general data type, the memory structure. It represents the simplified memory model used by the buffer management functions. Let us denote the object of this type by $MEM$. Obviously, there is only one such object at each time.

This type consists of two parts:

– the array of headers: $MEM.HEAD$

– the array of pages: $MEM.PAGE$.

We use $HEAD$ and $PAGE$ as shorthands for $MEM.HEAD$ and $MEM.PAGE$, correspondingly. Also, we introduce the following notations: for any array $Ar$ we
denote its size by $|Ar|$ and the $i$-th element of this array by $Ar[i]$, where $i \in \mathbb{N}$. All elements are numbered starting from one.

Each element $i$ of the header array has the following items:

- $HEAD[i].size$ - size of the buffers in pages contained in the list, whose header is $HEAD[i]$. The value of this item is greater than zero (the last header with $size$ equal to zero from the source code is used only for determining the end of the array, thus it is absent in the $HEAD$ array);

- $HEAD[i].max_free$ - maximal number of buffers in any page contained in the list, whose header is $HEAD[i]$. The value of this item is greater than zero and is computed by

$$
HEAD[i].max_free = \left\lfloor \frac{MDB_ALLOC_CHUNKSZ - MNG_SIZE}{HEAD[i].size} \right\rfloor,
$$

where $MDB_ALLOC_CHUNKSZ$ is a constant defining the page size (4096KB) and $MNG_SIZE$ is the size of the management information;

- $HEAD[i].list_of_lists$ - pointer to the page list. If the value of this item is equal to zero, this is interpreted as a NULL pointer.

Each element $j$ of the page array has the following items:

- $PAGE[j].mng$ - the management information of page $PAGE[j]$, described later;

- $PAGE[j].BUF$ - the array of buffers in this page. $BUF_j$ is a shorthand for $PAGE[j].BUF$;

- $PAGE[j].size$ - the size of the buffers in this page.

The management information consists of the following fields:
- $PAGE[j].mng.bl$ - the index of the header which defines the same size as the size of buffers in this page. The value of this item is greater than zero;

- $PAGE[j].mng.next_freelist, PAGE[j].mng.prev_freelist$ - indices of the next and of the previous pages for page $PAGE[j]$ in the doubly linked list; can be equal to zero (i.e., there is no previous or next page for this page);

- $PAGE[j].mng.num_free$ - number of free buffers in this page. The value of this item is restricted by the following inequality:

$$0 \leq PAGE[j].mng.num_free \leq HEAD[i].max_free,$$

where $i = PAGE[j].mng.bl$;

- $PAGE[j].mng.freelist$ - the pointer to the first free buffer of the buffer list in the page $PAGE[j]$:

Buffers are used for storing MDB nodes. Thus, the content of buffers does not matter for described functions, so we use only information about buffers. Each element $k$ of the buffer array of the $j$-th page has the following items:

- $BUF_j[k].address$ - the address (offset) of this buffer;

- $BUF_j[k].size$ - the size of this buffer;

- $BUF_j[k].free$ - a flag with the following value:

$$BUF_j[k].free = \begin{cases} 0 & \text{if the buffer is free,} \\ 1 & \text{if the buffer is busy;} \end{cases}$$

- $BUF_j[k].next$ - pointer to the next free buffer of the buffer list in the page $PAGE[j]$, if this buffer is free, otherwise it is equal to zero.
The whole type hierarchy is represented in figure 9.

It is necessary to define the *pointer* type for the buffer which is an output of the allocation function or an input for the deallocation function. Such a pointer has two items:

- *pointer.page_in* - the index of the page which contains this buffer,
- *pointer.buf_in* - the index of the buffer in the array of that page.

### 3.2 Translation of the functions

In order to specify the functions, we divide the whole source code into smaller parts called subfunctions. They can be described as functions which have the object of memory type and other necessary data as input parameters and the object of memory type with some changed objects as output. In all specifications, only changed elements are described; all the other elements stay the same.

#### 3.2.1 The *mdb_alloc_buffer* function

Let us consider the allocation function. We can declare it in the following
way. Let $MEM$ be a state of memory before the function invocation and $size$ be the size of a buffer to be allocated, then we have

$$\text{mdb.alloc.buffer}(MEM, size) = (MEM', pointer)$$

where $MEM'$ is the state of memory after the function execution and $pointer$ is the pointer to the allocated buffer.

Based on the description of function behavior, one can distinguish the following blocks:

- searching the corresponding header ($searchBl$),
- creating a new page ($createPage$),
- allocating a buffer ($allocBuffer$),
- removing the page with all busy buffers ($remPage$).

**The searchBl function**

The $searchBl$ function looks over all elements of the header array and returns the index of the element $i$ that fulfills $HEAD[i].size = size$. If such an element does not exist, it returns zero. The declaration is the following:

$$searchBl(HEAD, size) = i.$$  

This function uses the auxiliary function $rec.search$, which searches in the header array from $HEAD[p]$ to $HEAD[1]$ the element defining the size $size$:

$$rec.search(HEAD, size, p) = \begin{cases} p & \text{if } HEAD[p].size = size \lor p = 0, \\ rec.search(HEAD, size, p - 1) & \text{otherwise.} \end{cases}$$
It is invoked with the size of $HEAD$ array as the third argument:

$$
searchBl(HEAD, size) = rec_search(HEAD, size, |HEAD|).
$$

Thus, the function is realized recursively and in order to use the result of this function, we need the following lemma:

**Lemma 1.** Assume that there are no two headers which define the same size in the $HEAD$ array. If there exists a header which define the size $size$ then $searchBl(HEAD, size)$ returns the index of this header, otherwise it returns zero.

$$
\exists i_1, i_2 : 0 < i_1, i_2 \leq |HEAD| \land i_1 \neq i_2 \land HEAD[i_1].size = HEAD[i_2].size \Rightarrow
$$

$$
search(HEAD, size) = \begin{cases} 
  i & \text{if } \exists i : 0 < i \leq |HEAD| \land HEAD[i].size = size, \\
  0 & \text{otherwise.}
\end{cases}
$$

**The createPage function**

The $createPage$ function is used for creating a new page in the Buffer Pool. Its declaration is the following:

$$
createPage(MEM, bl) = MEM',
$$

where $MEM$ and $MEM'$ are the state of the memory before and after function execution and $bl$ is the index of the header defining the size of the buffers which should be located in the new page.

In accordance with the source code, this function gets a piece of memory from the kernel whose size is equal to one physical frame (4096KB). We denote this object by $page$. It can be used as a typical page in the Buffer Pool for allocation of the buffers of necessary size.

$$
page.size = HEAD[bl].size
$$

First, of course, this page should be initialized. The management information reflects the situation where all buffers are free:
The number of buffers in the page is maximal:
\[ |\text{page.BUF}| = \text{HEAD}[bl].\text{max\_free} \]

All buffers located in the page are linked in the list as shown in figure 7. For all indices \( k \) of the page.BUF array, we need to define the pointer to the next element:

\[
\text{page.BUF}[k].\text{next} = \begin{cases} 
  k + 1 & \text{if } 0 < k < |\text{page.BUF}| \\
  0 & \text{if } k = |\text{page.BUF}| 
\end{cases}
\]

The buffers are aligned by the last page address (figure):

\[
\text{page.BUF}[k].\text{address} = \text{MDB\_ALLOC\_CHU\_NKSZ} - \\
  - (\text{HEAD}[bl].\text{max\_free} - k + 1) \ast \text{HEAD}[bl].\text{size}
\]

\[
\text{page.BUF}[k].\text{free} = \text{TRUE}
\]

\[
\text{page.BUF}[k].\text{size} = \text{HEAD}[bl].\text{size}
\]

The new page is inserted in the list of pages (it is added to the page array and the pointer of the corresponding header is redefined in order to point to this page):

\[ |PAGE'| = |PAGE| + 1 \]

\[
\text{PAGE}'[j] = \begin{cases} 
  \text{page} & \text{if } j = |PAGE'|, \\
  \text{PAGE}[j] & \text{otherwise.}
\end{cases}
\]

\[
\text{HEAD}'[i].\text{list\_of\_lists} = \begin{cases} 
  |PAGE| + 1 & \text{if } i = bl, \\
  \text{HEAD}[i].\text{list\_of\_lists} & \text{otherwise.}
\end{cases}
\]
After these operations are executed, \( HEAD' \) and \( PAGE' \) form the new state of memory \( MEM' \).

**The allocBuffer function**

The declaration of the allocBuffer function is the following:

\[
allocBuffer(MEM, page_in) = (MEM', pointer),
\]

where \( MEM \) and \( MEM' \) are the memory state before and after function execution, \( page_in \) is an index of the page from which a buffer is allocated, and \( pointer \) is the pointer to the allocated buffer; it is also the output result of the whole \texttt{mdb_alloc_buffer} function. The allocation of a buffer is performed and information about the current number of free buffers is changed in this function.

The \( pointer \) is defined to the first free buffer in the \( page_in \) page:

\[
pointer.page_in = page_in
\]
\[
pointer.buf_in = PAGE[page_in].mng.freelist
\]

The buffer becomes busy:

\[
BUF_j[k].free = \begin{cases} FALSE & \text{if } j = page_in \land k = PAGE[j].mng.freelist, \\ BUF_j[k].free & \text{otherwise.} \end{cases}
\]

The buffer is removed from the list:

\[
PAGE'[j].mng.freelist = \begin{cases} BUF_j[PAGE[j].mng.freelist].next & \text{if } j = page_in, \\ PAGE[j].mng.freelist & \text{otherwise.} \end{cases}
\]

The number of free buffer is decreased:
\[ \text{PAGE}'[j].\text{mng.num_free} = \begin{cases} \text{PAGE}[j].\text{num_free} - 1 & \text{if } j = \text{page_in}, \\
\text{PAGE}[j].\text{mng.num_free} & \text{otherwise}. \end{cases} \]

The header array \( \text{HEAD} \) stays the same.

**The remPage function**

The \( \text{remPage} \) function has the following declaration:

\[ \text{remPage}(\text{MEM}, \text{page_in}, \text{bl}) = \text{MEM}' . \]

\( \text{MEM} \) and \( \text{MEM}' \) are once again the state of memory before and after function execution, \( \text{page_in} \) is the index of the page to be removed, and \( \text{bl} \) is the header which points to the list containing the page. The function removes a page from the list of pages if the page has only busy buffers.

Operations performed by this function are:

The previous pointer in the next page is redefined:

\[
\text{if } \text{PAGE}[\text{page_in}].\text{mng.next_freelist} \neq \text{NULL} \text{ then}
\]

\[ \text{PAGE}'[j].\text{mng.prev_freelist} = \]

\[
\begin{cases} 
\text{NULL} & \text{if } j = \text{PAGE}[\text{page_in}].\text{mng.next_freelist}, \\
\text{PAGE}[j].\text{mng.prev_freelist} & \text{otherwise}. 
\end{cases}
\]

The pointer of the header now points to the next page:

\[ \text{HEAD}'[i].\text{list_of_lists} = \begin{cases} 
\text{PAGE}[\text{page_in}].\text{mng.next_freelist} & \text{if } i = \text{bl}, \\
\text{HEAD}[i].\text{list_of_lists} & \text{otherwise}. 
\end{cases} \]

Thus, \( \text{PAGE}' \) and \( \text{HEAD}' \) form the new memory state \( \text{MEM}' \).

**Subfunction combination**

Now we can define \texttt{mdb_alloc_buffer} function using all functions described above:
\[ bl = \text{searchBl}(\text{HEAD}, \text{size}) \]

if \( bl = 0 \) then

    return error message

else

    if \( \text{HEAD}[bl].\text{list_of_lists} = 0 \) then

        \[ MEM^1 = \text{createPage}(MEM, bl) \]

    else

        \[ MEM^1 = MEM \]

    endif

endif

\[ page\_in = \text{HEAD}^1[bl].\text{list_of_lists} \]

\[ (MEM^2, pointer) = \text{allocBuffer}(MEM^1, page\_in) \]

if \( \text{PAGE}^2[page\_in].\text{mng.num_free} = 0 \) then

    \[ MEM' = \text{remPage}(MEM^2, page\_in, bl) \]

else

    \[ MEM' = MEM^2 \]

endif

return pointer

endif

3.2.2 The mdb_free_buffer function

We now formalize the deallocation function. It can be declared in the following way:

\[ \text{mdb_free_buffer}(MEM, pointer) = MEM', \]

where \( MEM \) and \( MEM' \) are the states of memory before and after the function execution and \( pointer \) is the pointer to the buffer to be deallocated. Based on the function behavior description, one can distinguish the following blocks:
- restoring the buffer into the page (*putBuffer*),
- inserting a page into the list of pages if the deallocated buffer is the first buffer in the page (*putPage*),
- removing and deallocating a page with only free buffers (*freePage*).

**The putBuffer function**

The declaration of the *putBuffer* function is the following:

\[
putBuffer(MEM, pointer) = MEM',
\]

where \(MEM\) and \(MEM'\) are the states of memory before and after this function execution, and \(pointer\) is the pointer to the buffer to be deallocated. In this function, a buffer is deallocated and the information about the current number of free buffers in the corresponding page is changed.

The buffer is inserted at the beginning of the list of free and the pointer to the first free buffer in the list is changed:

\[
BUF_j[k].next =
\]

\[
= \begin{cases} 
PAGE[j].mng.freelist & \text{if } j = pointer.page_in \land k = pointer.buf_in, \\
BUF_j[k].next & \text{otherwise}. 
\end{cases}
\]

\[
PAGE'[j].mng.freelist = \begin{cases} 
pointer.buf_in & \text{if } j = pointer.page_in, \\
PAGE[j].mng.freelist & \text{otherwise}. 
\end{cases}
\]

The buffer becomes free:

\[
BUF_j[k].free = \begin{cases} 
TRUE & \text{if } j = pointer.page_in \land k = pointer.buf_in, \\
BUF_j[k].free & \text{otherwise}. 
\end{cases}
\]
The number of free buffers is increased:

\[
PAGE'[j].mng.num_free = \begin{cases} 
  PAGE[j].mng.num_free + 1 & \text{if } j = \text{pointer.page_in}, \\
  PAGE[j].mng.num_free & \text{otherwise}.
\end{cases}
\]

The \textit{putPage} function

The \textit{putPage} function is used in order insert the page in the Buffer Pool in case that page has the first free buffer. It is declared in the following way:

\[
\text{putPage}(MEM, page\_in, bl) = MEM',
\]

where \(MEM\) and \(MEM'\) are the state of memory before and after this function execution, and \(page\_in\) is the index of the page which should be inserted in the list pointed to by \(bl\). The page is inserted at the beginning of the list.

Pointers to the previous and next page are redefined:

\[
\begin{align*}
PAGE'[j].mng.prev_free_list &= \begin{cases} 
  page\_in & \text{if } j = \text{HEAD}[bl].list\_of\_lists, \\
  \text{NULL} & \text{if } j = \text{page}\_in, \\
  PAGE[j].mng.prev_free_list & \text{otherwise}.
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
PAGE'[j].mng.next_free_list &= \begin{cases} 
  \text{HEAD}[bl].list\_of\_lists & \text{if } j = \text{page}\_in, \\
  PAGE[j].mng.next_free_list & \text{otherwise}.
\end{cases} \\
\end{align*}
\]

The pointer to the beginning of the list is redefined:

\[
\begin{align*}
\text{HEAD}'[i].list\_of\_lists &= \begin{cases} 
  page\_in & \text{if } i = bl, \\
  \text{HEAD}[i].list\_of\_lists & \text{otherwise}.
\end{cases}
\end{align*}
\]
and HEAD' make up MEM'.

The freePage function

The freePage function has the following declaration:

\[
\text{freePage}(MEM, \text{page\_in}, bl) = MEM''.
\]

MEM and MEM'' are also the state of memory before and after this function execution. page_in is the index of the page to be freed, and bl is the header which points to the list containing that page. We need to free the page from the corresponding list by changing the pointers (removing it from the list) and by deleting it from the array.

Removing of a page from the list depends on the location of that page in the list. If the page is not the last page of the list, we need to redefine the prev_freelist pointer of the next page (figure 10.a). If the page is the first page in the list, we need to redefine the pointer of the corresponding header (figure 10.b); otherwise we redefine the next_freelist pointer of the previous page (figure 10.c).

\[
\begin{align*}
\text{if } PAGE[\text{page\_in}].\text{mng.next\_freelist} \neq \text{NULL then} \\
\text{PAGE'}[j].\text{mng.prev\_freelist} &= \\
&\begin{cases} 
\text{PAGE}[\text{page\_in}].\text{mng.prev\_freelist} & \text{if } j = \text{PAGE}[\text{page\_in}].\text{mng.next\_freelist}, \\
\text{PAGE}[j].\text{mng.prev\_freelist} & \text{otherwise} 
\end{cases} \\
\text{if } HEAD[bl].\text{list\_of\_lists} = \text{page\_in} \text{ then} \\
\text{HEAD'}[i].\text{list\_of\_lists} &= \begin{cases} 
\text{PAGE}[\text{page\_in}].\text{mng.next\_freelist} & \text{if } j = bl, \\
\text{HEAD}[i].\text{list\_of\_lists} & \text{otherwise} 
\end{cases} \\
\text{elsif } PAGE[\text{page\_in}].\text{mng.prev\_freelist} \neq \text{NULL then} \\
\end{align*}
\]
Figure 10: Removing the page from the list.

\[
PAGE'[j].mng.next_freelist =
\begin{cases} 
PAGE[page_in].mng.next_freelist & \text{if } j = PAGE[page_in].mng.prev_freelist, \\
PAGE[j].mng.next_freelist & \text{otherwise} 
\end{cases}
\]

Afterwards, the page is removed from the array of pages (see figure 11). The size of the page array is decreased:

\[
|PAGE''| = |PAGE'| - 1
\]
All elements with index greater than the index of the removed page should be shifted to the beginning of the array by one index:

\[
PAGE''[j] = \begin{cases} 
  PAGE'[j + 1] & \text{if } j \geq \text{page\_in}, \\
  PAGE'[j] & \text{if } j < \text{page\_in}.
\end{cases}
\]

All pointers to the pages (to the beginning of the lists, to the next pages and to the previous pages) with changed index are redefined:

\[
A = \{\text{HEAD}'[i].\text{list\_of\_lists} \mid \forall i : 0 < i \leq |\text{HEAD}'| \}
\cup \{\text{PAGE}'[j].\text{mng\_next\_freelist} \mid \forall j : 0 < j \leq |\text{PAGE}'| \}
\cup \{\text{PAGE}'[j].\text{mng\_prev\_freelist} \mid \forall j : 0 < j \leq |\text{PAGE}'| \}
\]
∀ Point' ∈ A :

\[ \text{Point''} = \begin{cases} 
  \text{Point'} - 1 & \text{if } \text{Point'} > \text{page_in}, \\
  \text{Point'} & \text{if } \text{Point'} < \text{page_in}, 
\end{cases} \]

where Point'' is the new value of each pointer from set A. The value for Point' = page_in is not defined, because according to the source code there are no such pointers.

**Subfunction combination**

Using these functions we can describe the `mdb_free_buffer` function which deallocates a buffer pointed to by `pointer`:

\[
\begin{align*}
bl &= \text{PAGE}[\text{pointer.page_in}].\text{mng.bl} \\
\text{MEM}^1 &= \text{putBuffer}(\text{MEM}, \text{pointer}) \\
\text{if } \text{PAGE}^1[\text{pointer.page_in}].\text{mng.num_free} = 1 \text{ then} \\
\text{MEM'} &= \text{putPage}(\text{MEM}^1, \text{pointer.page_in}, bl) \\
\text{elseif } \text{PAGE}^1[\text{pointer.page_in}].\text{mng.num_free} = \text{HEAD}[bl].\text{max_free} \text{ then} \\
\text{MEM'} &= \text{freePage}(\text{MEM}^1, \text{pointer.page_in}, bl) \\
\text{else} \\
\text{MEM'} &= \text{MEM}^1 \\
\text{endif}
\end{align*}
\]

This formal function description will be used in the proof of the criteria in order to determine which objects are changed.

3.3 Dynamic model

In the previous chapter the organization of data in the memory was described. Now we will introduce the time model. At each point of time, there exists only one object of type memory structure. The `mdb_alloc_buffer` and `mdb_free_buffer` functions change some elements of this object. Thus, the...
effect of the functions over time can be represented as a sequence of invocations of both functions over the initial memory structure (figure 12).

Figure 12: Time model.

Let us consider a set $S$ consisting of all possible memory states with the initial state $s_{init}$ which satisfies all requirements for the header array and does not have any page array (because before the first function invocation, there are no pages with buffers in the memory). So, each element of type memory in the sequence of the Buffer Pool updates can be obtained in the following way:

$$
s_0 = s_{init},
$$

$$
s_i = \begin{cases} 
\text{mdb\_alloc\_buffer}(s_{i-1}, alloc\_par) \\
\text{mdb\_free\_buffer}(s_{i-1}, free\_par), 
\end{cases}
$$

where $alloc\_par$ and $free\_par$ are input parameters of `mdb\_alloc\_buffer` and `mdb\_free\_buffer`, correspondingly. Therefore, we use induction in the proof. In order to prove that all criteria hold for all elements which belong to the set $S$ we have to show that:

- all criteria hold for the initial element $s_{init}$ (induction base),
- if all criteria hold for element $s_i$, then they hold for element $s_{i+1}$ after `mdb\_alloc\_buffer` or `mdb\_free\_buffer` function invocation, too (induction step).

For this model the main part of the criteria can be formulated as predicates, that hold in all states $s_i \in S$. For the following formulation of the criteria, there
are some restriction for the indices:

\[ 0 < i, i_1, i_2, \ldots \leq |HEAD| \]

\[ 0 < j, j_1, j_2, \ldots \leq |PAGE| \]

\[ \forall j : 0 < k, k_1, k_2, \ldots \leq |BUF_j| \]

**criterion 1.** Two buffers never share the same address space. This means that the last address of a buffer should be less than the first address of another buffer or the first address of a buffer should be greater than the last address of another buffer.

\[ \forall j, k_1, k_2 : k_1 \neq k_2 \Rightarrow \]

\[ BUF_j[k_1].address + PAGE[j].size \leq BUF_j[k_2].address \land \]

\[ BUF_j[k_1].address \geq BUF_j[k_2].address + PAGE[j].size \]

**criterion 2.** Buffers do not share their address space with management information.

\[ \forall j, k : BUF_j[k].address \geq MNG_SIZE \]

**criterion 3.** The buffers are located inside a page.

\[ \forall j, k : BUF_j[k].address + PAGE[j].size \leq MDBALLOC_CHUNKSZ \]

**criterion 4.** A buffer of a particular size is allocated in the page which contains buffers of this size.

\[ \forall j, k : PAGE[j].size = BUF_j[k].size \]
criterion 6.1. The pointer to the first free buffer points to a buffer inside
the page.
In other words, this buffer is in the buffer array.
\[ \forall j : \quad PAGE[j].mng.freelist \leq |BUF_j| \]

criterion 6.2. Each free buffer in a page except the last buffer (in the list
of free buffers) points to a free buffer in the same page.
\[ \forall j, k : \quad BUF_j[k].next \leq |BUF_j| \]

criterion 7. The pointer to the first free buffer in the current page points to
NULL if and only if there are no free buffers in this page.
\[ \forall j : \quad PAGE[j].mng.freelist = 0 \iff PAGE[j].mng.num_free = 0 \]

criterion 8.1. If the pointer of the header does not point to NULL, it points
to an existing page.
\[ \forall i : \quad HEAD[i].list_of_lists \leq |PAGE| \]

criterion 8.2. If the pointer to the next page with free buffers does not
point to NULL, it points to an existing page.
\[ \forall j : \quad PAGE[j].mng.next_freelist \leq |PAGE| \]

criterion 8.3. If the pointer to the previous page with free buffers does not
point to NULL, it points to an existing page.
\[ \forall j : \quad PAGE[j].mng.prev_freelist \leq |PAGE| \]

criterion 9. The pointer of the header of a particular buffer size points to
the empty list if and only if there are no pages with free buffers of the same size.
\[ \forall i : \quad HEAD[i].list_of_lists = 0 \iff \]
\[ j : \ (\text{PAGE}[j].size = \text{HEAD}[i].size \land \text{PAGE}[j].mng.num\_free > 0) \]

**criterion 10.** The pointer of the header of a particular buffer size points to the list which consists only of pages with buffers of this size.

This criterion includes two conditions:

1. The pointer of the header of a particular buffer size points to the page with buffers of the same size

\[
\forall i : \ \text{HEAD}[i].list\_of\_lists \neq 0 \Rightarrow \\
\text{PAGE}\left[ \text{HEAD}[i].list\_of\_lists \right].size = \text{HEAD}[i].size
\]

2. The pointer to the next page with free buffers of the page with a particular buffer size points to the pages with buffers of the same size

\[
\forall j : \ \text{PAGE}[j].mng.next\_freelist \neq 0 \Rightarrow \\
\text{PAGE}\left[ \text{PAGE}[j].mng.next\_freelist \right].size = \text{PAGE}[j].size
\]

**criterion 11.** A list which is pointed to by a header does not contain any pages without free buffers.

This criterion includes two conditions:

1. The pointer of the header points to the pages with some free buffers:

\[
\forall i : \ \text{HEAD}[i].list\_of\_lists \neq 0 \Rightarrow \\
\text{PAGE}\left[ \text{HEAD}[i].list\_of\_lists \right].mng.num\_free \neq 0
\]
2. Each page with some free buffers points to the pages with some free buffers

\[ \forall j : \ (PAGE[j].mng.num_free \neq 0 \land PAGE[j].mng.next_freelist \neq 0) \Rightarrow \]

\[ PAGE[PAGE[j].mng.next_freelist].mng.num_free \neq 0 \]

**criterion 12.** A page with a particular buffer size contains a pointer to the header which defines the same size.

\[ \forall j : \ HEAD[PAGE[j].mng.bl].size = PAGE[j].size \]

The remaining criteria can be described as a correct result of function execution. In the following formulation of the criteria, there are some notations for functions results: the criteria for the `mdb_alloc_buffer` function hold for all `MEM` and `size`, the result of function execution is `MEM'` and `alloc_pointer`. The criteria for the `mdb_free_buffer` function hold for all `MEM` and `dealloc_pointer`, the result of function execution is `MEM'`. `alloc_pointer` and `dealloc_pointer` are objects of type `pointer` defined above and they consist of the page index `page_in` and the buffer index `buffer_in`.

The criteria 13 and 15 are for the `mdb_alloc_buffer` function, the criteria 14 and 16 are for the `mdb_free_buffer` function.

**criterion 13.** A new page for a buffer size will not be created if there is at least one free buffer of the same size.

\[ \exists j, k : \ BUF_j[k].size = size \land BUF_j[k].free = TRUE \]

\[ \Rightarrow \ |PAGE'| = |PAGE| \]
criterion 14. If the page does not contain any busy buffers, the memory space of that page is freed.

\[ \forall k: (k \neq \text{dealloc_pointer.buf_in} \Rightarrow \]

\[ \text{BUF}_{\text{dealloc_pointer.page_in}[k].\text{free} = \text{TRUE}} \Rightarrow \]

\[ |\text{PAGE}'| = |\text{PAGE}| - 1 \]

criterion 15. After the execution of the \text{mdb_alloc_buffer} function, there exists a buffer which can be used to store an \text{MDB} element of the specified size.

This criterion consists of the following conditions:

– the allocated buffer is marked as busy (therefore, this buffer cannot be allocated for another \text{MDB} element);

– the state of the other buffers is not changed;

– if this buffer is not allocated from the new page, then it was a free buffer in the existing page.

\[ \exists i: \text{HEAD}[i].\text{size} = \text{size} \Rightarrow \]

\[ \text{BUF}'_{\text{alloc_pointer.page_in}[\text{alloc_pointer.buf_in}].\text{free} = \text{FALSE}} \land \]

\[ (\forall j, k: j \neq \text{alloc_pointer.page_in} \land k \neq \text{alloc_pointer.buf_in} \]

\[ \Rightarrow \text{BUF}'_j[k].\text{free} = \text{BUF}_j[k].\text{free} \) \land \]

45
\( alloc\_pointer.page\_in = |PAGE| + 1 \lor
\)
\[ BUF_{alloc\_pointer.page\_in[alloc\_pointer.buf\_in].free = TRUE} \]

**criterion 16.** If the `mdb_free_buffer` function is invoked in order to deallocate a buffer, that buffer is marked in the Buffer Pool as free after its execution.

This criterion consists of the following conditions:

- the deallocated buffer was busy before the function invocation;

- the state of the other buffers is not changed;

- if this buffer is not deallocated from the deleted page, then it is marked as free (otherwise, it has become free because a page is deleted only if all buffers are free).

\[ BUF_{dealloc\_pointer.page\_in[dealloc\_pointer.buf\_in].free = FALSE} \land
\]
\( (\forall j, k : j \neq dealloc\_pointer.page\_in \land k \neq dealloc\_pointer.buf\_in
\)
\[ \Rightarrow BUF_j'[k].free = BUF_j[k].free) \land
\]
\[ (|PAGE'\| = |PAGE| - 1 \lor
\]
\[ BUF_{dealloc\_pointer.page\_in[dealloc\_pointer.buf\_in].free = TRUE} \]

There is one more criterion which has not been formalized yet:

**criterion 5.** The amount of memory space in the page which is not used by memory information and busy buffers is large enough to allocate `num_free` buffers.
For simplicity, this criterion is divided into two parts.

1. After the execution of the `mdb_alloc_buffer` function, the number of free buffers in the page which contains the allocated buffer is decremented:

   - for that page: if it is a new page, then the number of the free buffers is one less than the maximum number; otherwise it is decremented;
   - the number of free buffers in other pages stays the same.

\[
\exists i : \text{HEAD}[i].size = size \Rightarrow \left( \forall j : j \neq \text{alloc_pointer.page_in} \Rightarrow \text{PAGE}'[j].mng.num_free = \text{PAGE}[j].mng.num_free \right) \land \\
(\text{alloc_pointer.page_in} = |\text{PAGE}| + 1 \land \\
\text{PAGE}'[|\text{PAGE}'|].mng.num_free = \text{HEAD}[i].max_free - 1) \lor \\
\text{PAGE}'[\text{alloc_pointer.page_in}].mng.num_free = \\
= \text{PAGE}[\text{alloc_pointer.page_in}].mng.num_free - 1)
\]

2. After the execution of the `mdb_free_buffer` function, the number of free buffers in the page where the deallocated buffer is situated, is incremented:

   - if that page was not deleted, than the number of the free buffers is incremented;
   - the number of free buffers in other pages stays the same.
\( (\forall j : j \neq \text{dealloc\_pointer\_page\_in} \Rightarrow \)

\[ \text{PAGE}'[j].\text{mng}\_\text{num}\_\text{free} = \text{PAGE}[j].\text{mng}\_\text{num}\_\text{free} \] \&

\( \left( |\text{PAGE}'| = |\text{PAGE}| - 1 \right) \lor \)

\[ \text{PAGE}'[\text{dealloc\_pointer\_page\_in}].\text{mng}\_\text{num}\_\text{free} = \]

\[ = \text{PAGE}[\text{dealloc\_pointer\_page\_in}].\text{mng}\_\text{num}\_\text{free} + 1 \)

If both parts of the criterion hold for all elements in the sequence of the Buffer Pool updates, we can conclude that \text{num\_free} is the number of buffers which can be allocated in this page.

3.4 Initial condition

As it was said above the Buffer Pool is initialized before the functions are invoked. Let us formalize the properties of the initial memory object. All indices are restricted by the size of their array as in the previous section.

- There is no header which defines a size equal to zero:

\[ \exists i : \text{HEAD}[i].\text{size} = 0 \]

- There are no two headers, which define the same size (this condition was already used in the lemma 1):

\[ \exists i_1, i_2 : i_1 \neq i_2 \land \text{HEAD}[i_1].\text{size} = \text{HEAD}[i_2].\text{size} \]

- The \text{max\_free} element exactly determines the maximal number of free buffers in one page:

\[ \forall i : \text{HEAD}[i].\text{max\_free} = \]
= \left[ \frac{\text{MDB_ALLOC_CHUNKSZ} - \text{MNG_SIZE}}{\text{HEAD}[i].size} \right]

- All pointers to the list of pages point to NULL:

\[ \forall i : \text{HEAD}[i].list_of_lists = \text{NULL} \]

- The array of pages does not contain any element:

\[ |\text{PAGE}| = 0 \]

These conditions are used in order to prove that the criteria hold over the initial memory object.

3.5 Input condition for the \texttt{mdb_free_buffer} function

After analysis of the deallocation function, it turned out that for the correctness of the function, the input pointer must point to an existing busy buffer:

\[ 0 < \text{dealloc_pointer.page_in} \leq |\text{PAGE}| \quad \land \]

\[ 0 < \text{dealloc_pointer.buf_in} \leq |\text{BUF}_{\text{dealloc_pointer.page_in}}| \quad \land \]

\[ \text{BUF}_{\text{dealloc_pointer.page_in}}[\text{dealloc_pointer.buf_in}].\text{free} = \text{FALSE} \]
In order to prove the correctness of the introduced buffer management, we need to prove that each criterion holds for each memory state which can be obtained from the initial state by a sequence of function invocations (this sequence can be empty). Therefore, for criteria about memory states (all criteria except 5, 13, 14, 15, 16), three lemmas are formulated:

1. for the initial element

2. for an element obtained after execution of the \texttt{mdb_alloc_buffer} function over the previous element

3. for an element obtained after execution of the \texttt{mdb_free_buffer} function over the previous element

For each criterion these three lemmas are combined into one lemma which is proved by induction. The first lemma is the induction base, and the last two lemmas form the induction step.

For the initial element, the initial conditions described in section 3.4 are used. The proof is trivial because the page array is empty and the header array is explicitly given. In proofs for other elements, calls of both functions are considered and their specifications described in section 3.2 are used.

The rest of the criteria coincide with the lemmas; thus, it is not necessary to reformulate them.

At the end all lemmas are united into one main theorem stating the correctness of buffer management. This theorem is trivially proved by using all the other lemmas; thus, the correctness of buffer management in L4 is concluded.
4.1 Proof of some lemmas

In this section, we show the proofs of some lemmas. First, let us consider a trivial lemma stating that the ninth criterion holds for the initial state.

**Lemma 2.** In the initial state, the header pointer of a particular buffer size points to the empty list if and only if there are no pages with free buffers of the same size.

**Proof.** According to the initial conditions, the initial state has the following properties:

- all pointers to the list of pages point to NULL,
- the array of pages does not contain any elements.

This concludes the claim.

QED.

In the following lemmas, we denote criterion 6.1 as $\text{freelist\_range}$, criterion 6.2 as $\text{next\_range}$, criterion 9 as $\text{empty\_list}$ criterion 10 as $\text{list\_size}$, and criterion 12 as $\text{point\_to\_head}$. For any criterion $\text{crit}$, $\text{crit}(MEM)$ means that this criterion hold for the state $MEM$.

**Lemma 3.** If the criteria 6.1 and 6.2 hold for the state $MEM$, then the criteria 6.1 and 6.2 hold for the state $MEM'$ after the $\text{mdb\_alloc\_buffer}$ function is executed.

\[
\text{freelist\_range}(MEM) \wedge \text{next\_range}(MEM) \Rightarrow \\
\text{freelist\_range}(MEM') \wedge \text{next\_range}(MEM')
\]

**Proof.** We have that each pointer to a buffer in the $j$-th page of state $MEM$ has a value in the range $0, \ldots, |BUF_j|$, and we have to show that each pointer to a
buffer in the \(j\)-th page of state \(MEM'\) has a value in the range \(0, \ldots, |BUF_j|\). We divide the function execution into parts which correspond to the subfunctions and use notations as in the function specification (see 4.2.1). Now we show that both criteria hold after each subfunction execution. Let \(size\) be the size of the buffer for which the allocation function is invoked.

1. Search the corresponding header. By lemma 1, \(bl\) is the index of the header defining this size if the size \(size\) is supported by the system; otherwise, it is equal to zero. In the case of \(bl = 0\), the memory structure is not changed and both criteria trivially hold. In case \(bl \neq 0\), we check the pointer of the header.

2. There are two situation for the header pointer:

2.1. The header pointer points to \texttt{NULL}, and a new page have to be created. By the \texttt{createPage} function, the \(PAGE^1\) array consists of the \(PAGE\) array and one new element \(page\). Since this page is initialized, \(page.mng.freelist = 1\) holds.

\[
|page.BUF| = HEAD[bl].max_free > 0
\]

\[
\Rightarrow \quad 1 = page.mng.freelist \leq |page.BUF|
\]

For the buffers:

\[
page.BUF[k].next = \begin{cases} 
  k + 1 & \text{if } 0 < k < |page.BUF| \\
  0 & \text{if } k = |page.BUF| \\
\end{cases}
\]

\[
\leq (|page.BUF| - 1) + 1 = |page.BUF|
\]

2.2. If the list of pages is not empty, \(MEM\) stays the same (\(MEM^1 = MEM\)).

Thus, \(freelist_range(MEM^1)\) and \(next_range(MEM^1)\) are true.
3. A buffer allocation. For the *allocBuffer* function, *page_in* is the index of the page with free buffers of the size *size*. The value of the buffer pointer *next* is unchanged for each buffer. The value of the pointer to the list of free buffers is changed only for the *page_in*-th page. It is replaced by a pointer to the first free buffer. Since *next_range*(*MEM*₁) and |*BUF*₈| = |*BUF*₇|, we have:

\[
PAGE²[page_in].mng.freelist = Buf⁰_{page_in}[PAGE¹[page_in].mng.freelist].next \\
\leq |Buf⁰_{page_in}| \\
= |Buf²_{page_in}|
\]

4. Removing the page from the list of pages. The *remPage* function does not change the pointers to buffers. Thus, we have

\[
freelist\_range(MEM²) \land next\_range(MEM²) \Rightarrow \\
freelist\_range(MEM') \land next\_range(MEM'),
\]

which concludes the claim.

**QED.**

**Lemma 4.** If the criteria 10 and 12 hold for the state *MEM*, then the criterion 10 holds for the state *MEM'* after execution of the *mdb_free_buffer* function.

\[
list\_size(MEM) \land point\_to\_head(MEM) \Rightarrow list\_size(MEM')
\]

**Proof.** We have to show that after the deallocation function the following two statements hold.

- For each header pointing to non-empty list, the size defined in the header is equal to the buffer size in the first page of the list.

- Each pair of adjacent pages in the list contains buffers of the same size.
We divide the function execution into two parts: restoring the buffer into the page and operations with the page. The first part is realized by the `putBuffer` function. It does not modify the construction of lists and all the fields `size` (in headers and in pages) are not affected. Thus, the tenth criterion holds for the intermediate state $MEM^1$ after the `putBuffer` function: $list\_size(MEM^1)$.

The second part consists of either an insert or a remove operation. Here, only the header and pages which were adjacent to the removed page or will be adjacent to the inserted page are considered. For other pages, the criterion holds because of $list\_size(MEM^1)$. Thus, we have three cases:

1. The page is inserted into the list of pages. Let $i$ be the index of the header pointing to the list in which the page is inserted, and $page\_in$ the index of the inserted page. Because of criterion 12, we have for the header and the first page of the list:

$$PAGE'[HEAD'[i].list\_of\_lists].size$$

   $$= PAGE'[page\_in].size$$ \hspace{1cm} \text{by construction}$$

   $$= PAGE'[page\_in].size$$ \hspace{1cm} \text{putPage function}$$

   $$= HEAD'[PAGE'[page\_in].mng.bl].size$$ \hspace{1cm} \text{point\_to\_head}(MEM')$$

   $$= HEAD'[i].size$$ \hspace{1cm} \text{by construction}$$

   $$= HEAD'[i].size$$ \hspace{1cm} \text{putPage function}$$

If the list is not empty and $j$ is the index of the page which was the first in the list, then, using the equations above, we have for the first two pages of
the list:

\[
PAGER[i] = PAGER[i_1].mng.next_freelist.size
\]

\[
= PAGER[j].size \\
= PAGER[j].size \\
= PAGER[HEAD[i].list_of_lists].size \\
= HEAD[i].size \\
= HEAD[i].size \\
= PAGER[page_in].size \\
\]

by construction

putPage function

by construction

list_size(MEM)

putPage function

by above proof,

i.e., \( list_size(MEM') \) holds.

2. The page is removed from the list. Here, we consider the different positions of the page that is to be removed:

– The page to be removed is the first page in the list. Let \( i \) be the index of the header pointing to the page, \( page_in \) the index of the page to be removed, and \( j \) the index of the second page in the list. If the \( j \)-th page is absent, then the header points to NULL and there is nothing to show. Otherwise, the header points to the second page:

\[
PAGER[i] = PAGER[i_1].list_of_lists.size
\]

\[
= PAGER[j].size \\
= PAGER[j].size \\
= PAGER[PAGE[page_in].mng.next_freelist].size \\
= PAGE[page_in].size \\
= PAGE[HEAD[i].list_of_lists].size \\
= HEAD[i].size \\
= HEAD[i].size \\
\]

by construction

freePage function

by construction

list_size(MEM)

by construction

list_size(MEM)

freePage function

– The page to be removed is the last page in the list. The previous page then points to NULL and there is nothing to show.
The page to be removed is in the middle of the list. Let \( \text{page\_in} \) be the index of this page, \( \text{prev} \) the index of the previous page, and \( \text{next} \) the index of the next page. The \( \text{prev} \)-th and the \( \text{next} \)-th pages will be adjacent after execution:

\[
PAGEx[PAGE'[\text{prev}].\text{mng.next_freelist}].\text{size} = PAGE'[\text{next}].\text{size} \quad \text{by construction}
\]

\[
PAGEx[PAGE'[\text{prev}].\text{mng.next_freelist}].\text{size} = \text{list}\_\text{size}(\text{MEM}^1)
\]

\[
PAGEx[PAGE'[\text{prev}].\text{mng.next_freelist}].\text{size} = \text{list}\_\text{size}(\text{MEM}^1)
\]

i.e. \( \text{list}\_\text{size}(\text{MEM}^1) \) holds.

3. Neither 1. nor 2. is the case. In this case, the criterion holds because of \( \text{MEM}' = \text{MEM}^1 \).

**QED.**

**Lemma 5.** If the criterion 9 holds for the state \( \text{MEM} \), then during the execution of the \text{mdb\_alloc\_buffer} function which was invoked for size \( \text{size} \), a new page will not be created if there is at least one free buffer of the same size (the number of pages in the array of the new state \( \text{MEM}' \) stays the same).

\[
\text{empty}\_\text{list}(\text{MEM}) \Rightarrow \forall \text{size} : \exists j, k :
\]

\[
(\text{BUF}_j[k].\text{size} = \text{size} \land \text{BUF}_j[k].\text{free} = \text{TRUE})
\]

\[
\Rightarrow |\text{PAGE}'| = |\text{PAGE}|
\]
**Proof.** We show the claim by contradiction. Assume, that a new page is created and there is a free buffer of size \( size \), i.e., there exist \( j \) and \( k \) such that

\[
BUF_j[k].size = size \land BUF_j[k].free = TRUE.
\]

Hence, there exists a header defining the size \( size \). According to the specification, a new page is created if and only if this header points to NULL. But in this case, there are no free buffers of this size by the ninth criterion. This contradicts the assumption and concludes the claim.

QED.

**Lemma 6.** After the execution of the \texttt{mdb_free_buffer} function for the buffer which is pointed to by \texttt{dealloc_pointer}, the number of free buffers in the page where the deallocated buffer is located is incremented and for all other pages, the number is not changed.

\[
\left( \forall j : j \neq dealloc\_pointer.page\_in \Rightarrow
\end{equation}
\]

\[
PAGE'[j].mng.num\_free = PAGE[j].mng.num\_free \right) \land
\]

\[
\left( \left| PAGE' \right| = \left| PAGE \right| - 1 \land
\end{equation}
\]

\[
PAGE[dealloc\_pointer.page\_in].mng.num\_free =
\]

\[
HEAD[PAGE[dealloc\_pointer.page\_in].mng.bl].max\_free - 1 \right) \lor
\]

\[
PAGE'[dealloc\_pointer.page\_in].mng.num\_free =
\]

\[
= PAGE[dealloc\_pointer.page\_in].mng.num\_free + 1
\]
Proof. According to the specification, the value of the field \( \text{num\_free} \) is unchanged if the index of the page is not equal to \( \text{dealloc\_pointer\_page\_in} \). This concludes the first part of the claim.

Let us consider two cases for the second part:

– the buffer to be deallocated is the last free buffer in the page

\[
PAGEx\text{dealloc pointer.page\_in}.mng.num\_free = \\
\text{HEAD}[PAGEx\text{dealloc pointer.page\_in}.mng.bl].max\_free - 1
\]

In this case, after incrementing \( \text{num\_free} \) according to the specification, the page is removed and the number of page is decremented which concludes this case of the claim.

– the buffer to be deallocated is not the last free buffer in the page

\[
PAGEx\text{dealloc pointer.page\_in}.mng.num\_free < \\
\text{HEAD}[PAGEx\text{dealloc pointer.page\_in}.mng.bl].max\_free - 1
\]

In this case, no pages are removed from the array of pages. Therefore, \( PAGEx\text{dealloc\_pointer\_page\_in} \) and \( PAGEx'[\text{dealloc\_pointer\_page\_in}] \) are indices of the same page. Thus, in this page the number of free buffers is incremented which concludes the second case of the claim.

So all claims are satisfied.

QED.
In this chapter the most interesting lemmas were proved. All lemmas are completely proved in \textit{PVS}. In particular, the overall correctness theorem for buffer management is proved in \textit{PVS}. In those proofs the functions and model specification written in \textit{PVS} specification language is used.
REFERENCES


2. Internet: http://searchwin2000.techtarget.com

3. Internet: http://www.cs.unc.edu

4. Internet: http://www.citforum.ru


