Master Thesis

Documentation of memory management functions in the L4 microkernel

Elena Petrova

Prof. Dr. W.J.Paul

Prof. Dr.-Ing. H. Hermanns

Computer Science Department
Saarland University

June 2003

1 Introduction

One of the most complex and difficult tasks of an operating system is managing the limited physical memory present in a computer. For example an operating system must distribute the physical memory among many processes that might be running simultaneously. The memory management can be seen as a control mechanism above the physical memory resources.

L4 is a operating system microkernel [1]. It constitutes a minimal base allowing to create various operating systems on its top. One of the tasks of the L4 microkernel is providing an address space abstraction. An *address space* of a process is a range of memory which is directly accessible by it during running. If the virtual memory mechanism is used then the address space is a set of mappings from virtual to physical memory (by the use of address translation). The source code of microkernel includes a set of functions which can be used to perform and to control different memory events. In this work I will show some aspects of the L4 memory management by describing the source code.

The L4 microkernel can be run on the several hardware architectures. There are some hardware-dependent files to be compiled for a special architecture version. In this work I will consider only Intel x86 architecture.

2 Memory organisation [1]

The traditional computer has a one-dimensional (linear) address space where addresses are numbered sequentially from zero to the upper limit of the memory. There are difficulties to use a linear address space with several processes simultaneously. All processes use the same address space (each process uses a defined region) and data of one thread is not protected from other threads. In this case the presence of errors in a thread can be caused by unauthorised access to its data by another thread.

To solve this difficulties, one can use a mechanism called *virtual memory* creating the illusion that a computer has more memory than it actually has and that each thread (process) can access the whole physical address space. The possible size of the physical address space is defined by the number of bytes that can be described with the address bit width. For 32-bit processors the maximum available virtual memory is 2^{32} or 4 GB.

A process accesses real memory trough its virtual address space, but since there usually is less physical memory than virtual memory, an operating system must have possibilities to set a correspondence between actual addresses

[1] For more details about the processor architecture and memory organisation see [2]

in the hardware and addresses used in a program. The virtual address spaces of each process must be completely separate from each other in such way that a process running one application cannot affect any other. An operating system must provide this protection.

Although virtual memory allows processes to have separate address spaces, there are times when it is necessary to share memory between some processes. This is also a job for the operating system.

2.1 Virtual memory with paging

In a paged memory system the main memory space as well as the virtual memory space of each process is subdivided into parts of equal size called *page frames*. For the indication of correspondence between virtual pages and pages of the main memory the operating system must generate a *page table* for each program (process or thread) and place it in the main memory of the computer. This page table is used for the translation of linear (virtual) page addresses to physical (actual) page addresses.

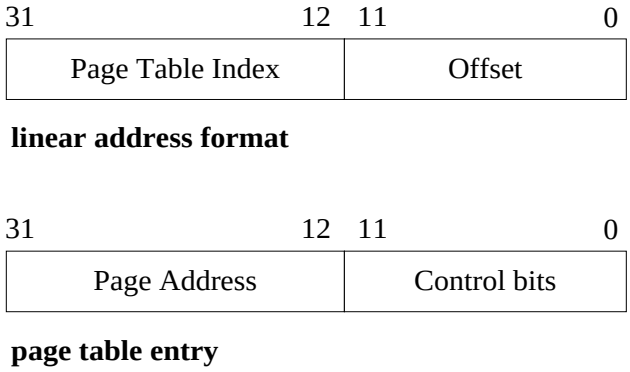

Figure 1: The linear address and the entry format

The virtual address consists of two parts: a page number and an offset (Figure 1). The first is used to find an entry in the page table, the latter is an offset in the physical page pointed to by this page table entry.

A page table entry contains a number (address) of a page frame and some control bits (flags). Since the physical memory can be less than the virtual address space a part of allocated memory fragments sometimes must be stored in the external memory and, hence, must be loaded from there when it is needed. One of the control bits of a page table entry indicates the presence of the page in the main memory. A page with an active presence

flag is called a *valid page*.

The process of the address translation is depicted in Figure 2. The address of the page table of the current task is stored in a special processor register CR3 also called as PDBR (Page Directory Base Register) by the task switcher.

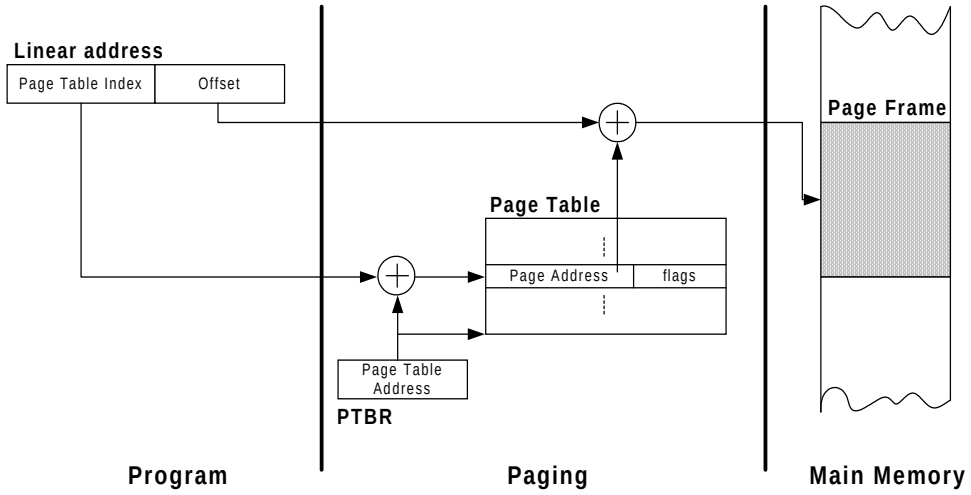

Figure 2: The address translation

If the addresses of all virtual pages were kept in one page table, then this table must be 4 MB large, since the whole linear space is 2^{32}= 4GB, the size of each page frame is 2^{12} = 4KB and each page table entry needs 4 bytes.

In practice, a two-level paging scheme is used. There are two types of page tables: one page table of the upper level, which is called a *page directory*. It has the size of one hardware page and contains 1 KB of entries. Each entry in the page directory (PDE - page directory entry) points to a page table of the second (lower) level. A page table in the second level has a size of 4 KB. Each entry of a page table (PTE) at the second level points to a page frame in the physical memory.

A page table entry and a page directory entry have the same format as in one-level paging. All changes concern only the linear address interpretation. The linear addresses are divided into three parts: page directory index, page table index and offset (Figure 3). The page directory index (10 upper bits of the address) is used to select an entry in the upper level table which points a page table of second level. The page table index (10 middle bits) is used to select an entry in the found page table and to find a corresponding page frame in the main memory. The address translation scheme is presented in

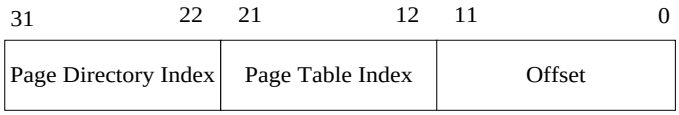

Figure 3: The linear address in 2-level paging

Figure 4.

Such a scheme allows to keep only a part of the tables in the memory simultaneously and to store all other tables in the external memory, i.e. it is not necessary to occupy 4 MB for table keeping.

In the following sections I will use the upper (or first) level for an entry location in the page directory and the lower (or second level) for an entry location in any page table.

3 Memory abstraction in L4[2]

3.1 Mapping and Granting

An address space of a thread contains all data which is accessible by this thread. As it was said earlier, an address space is a set of mappings from virtual to physical memory. Some portions of an address space can be inaccessible (invalid) since many mappings are not defined.

In L4 there is a recursive construction of address spaces. A thread can *map* a part of its address space into the address space of another thread. This operation allows two (or more) threads to share data located in this part of memory. The mapper retains full control over a mapped region and mappings can be revoked at any time.

Also, an address space can be *granted* to another thread. After this action the granter relinquishes all rights over a granted part of address space. The grantee gets full control over the granted address space, but there is also the possibility to perform a read-only grant with loss of write access.

[2]This chapter is written using [3]

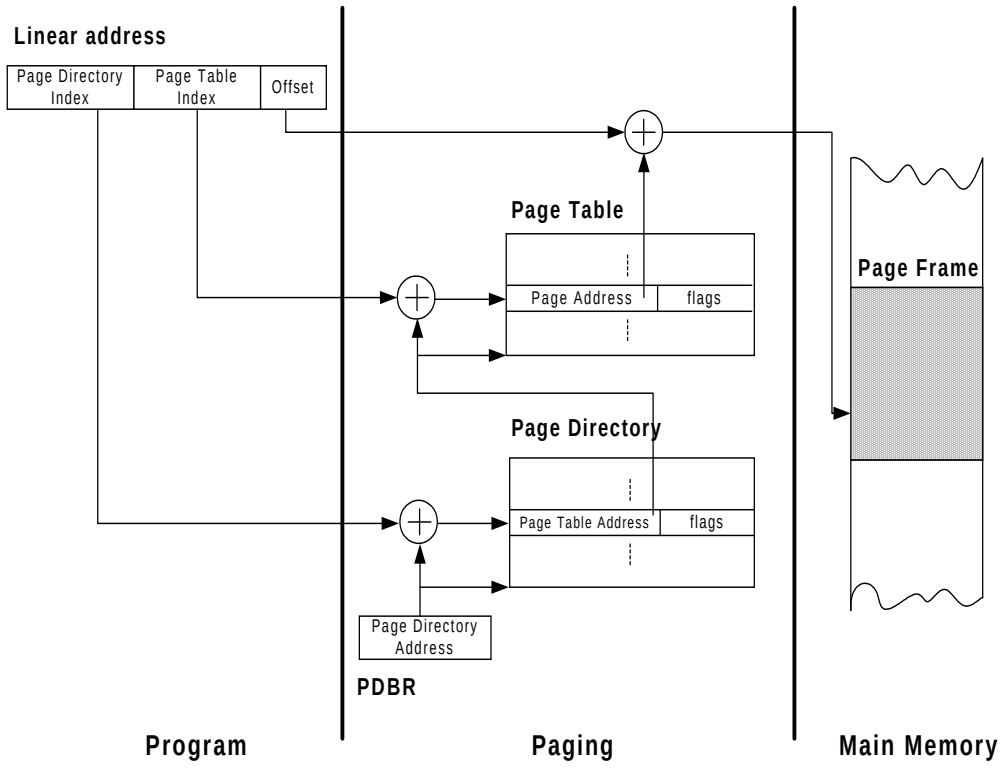

Figure 4: The address translation in 2-level paging

3.2 Flex-pages

Mapping and granting only operate on page tables, these operations don't copy any actual data. Mapping and granting are performed using IPC (inter-process communication) operations, as they require an agreement between the involved threads. Data transfers through mapping (granting) are described by *flex-pages* (or *fpages*).

A flex-page is a continuous part of virtual address space with the following properties:

- A fpage has size 2^s bytes and the smallest possible fpage size defined by hardware page size (4 KB).
- The base address is aligned by a border of 2^s.
- An fpage describes only valid pages.

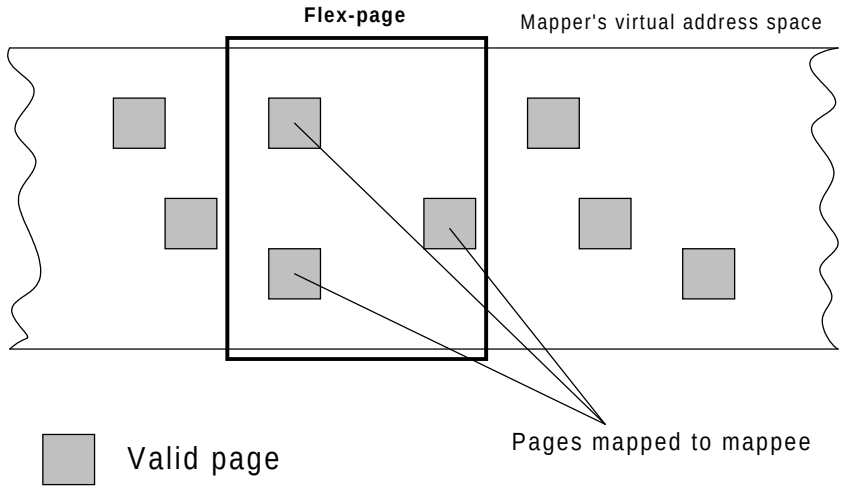

Figure 5: Flex-page

A mapper specifies the fpage which is to be received by the mappee in an message via IPC (or granter and grantee correspondingly). After the fpage is processed by a special procedure of L4, all valid pages within the fpage become accessible for the mappee (Figure 5).

3.2.1 Fpage parameters

An fpage is defined by two basic parameters: b and s. The mapped part of the address space is presented as $[b \times 2^s, (b+1) \times 2^s]$ (see properties).

If sender and receiver specify fpages of different size, one additional parameter h, *a hot-spot* is required. In this case the hot-spot defines how the mapping between fpages of different sizes happens. Consider 2^s is the size of sender's fpage and 2^r is the size of receiver's fpage. There are two cases (see Figure 6):

- $s < r$
  The address space starting from address $bs \times 2^s$ is mapped to the space starting from address $br_{[31,r]}h_{[r-1,s]}0_{[s-1,0]}$, where bs and br are the base addresses of sender and receiver correspondingly and h is the hot-spot address in the fpage of the receiver. Notation $a_{[s,t]}$ means taking the bits of vector a from position s to position t (inclusive).

- $s > r$
  The address space starting from address $bs_{[31,s]}h_{[s-1,r]}0_{[r-1,0]}$ is mapped

to the space starting from address $br \times 2^r$, where h is the hot-spot address in the fpage of the sender.

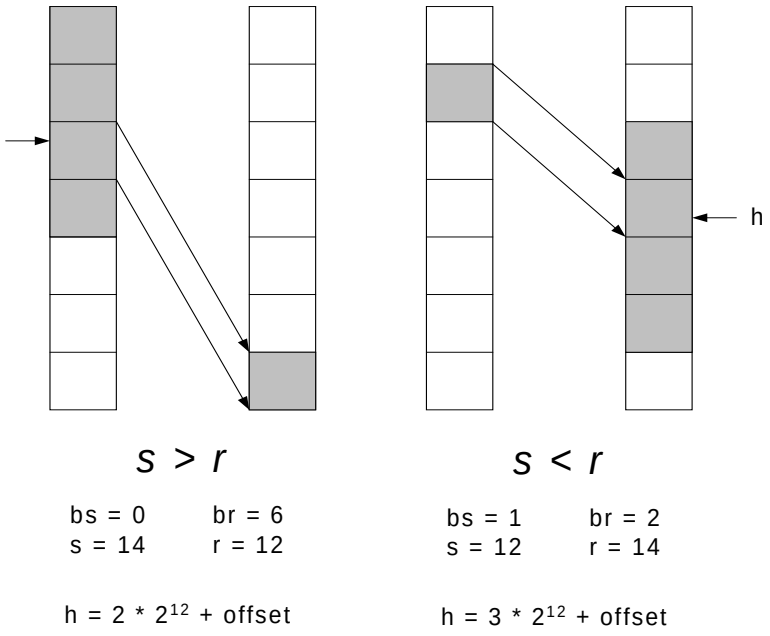

Figure 6: Mapping with fpages of different size

In other words, the address space described by the fpage of larger size is subdivided into several parts (fpages) of smaller fpage size and the hot-spot address shows the fpage which will actually be mapped (or granted), i.e. the fpage containing this address.

3.2.2 Fpage format

As it is illustrated at Figure 7 a fpage structure occupies 32 bits and has the following fields (the order from 0 to 31 bit):

- map/grant flag (1 bit)
- write/read-only flag (1 bit)
- size (6 bits)
  Size exponent to compute fpage size.
- zero 4 bits
  Bits are not used.

- base (20 bits)
  The base address of the fpage can be given in form $b/4096$ rather than just b because any fpage needs to be aligned to the hardware page size 4 KB (2^{12} bytes).

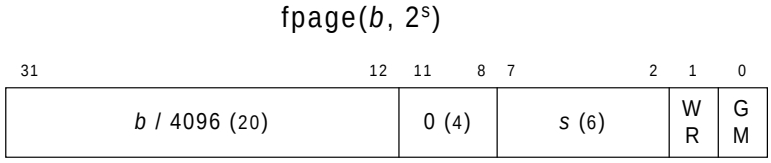

Figure 7: Fpage format

3.3 Mapping revoking

Mapping can be revoked by using the special system call `fpage_unmap`. The sender (owner of the address space) defines a fpage which should be unmapped from the address spaces into which it has been mapped before. The unmapping can be done completely or only as a change of rights. The way the unmap is performed is defined by the parameter of system call *map_mask*. This mask has four flags defining the behaviour of the unmap operation:

- The unmap operation
  `L4_FP_REMAP_PAGE` - set fpage to read-only
  `L4_FP_FLUSH_PAGE` - unmap fpage completely

- The unmap extent
  `L4_FP_OTHER_SPACES` - apply the unmap operation to all address spaces to which fpage has been mapped except for the original flexpage
  `L4_FP_ALL_SPACES` - perform the unmap operation for all address spaces including the original fpage

Table 1 shows all possible combinations of flags that are valid for unmapping (combining by logical OR).

3.4 The root pager

L4 allocates all resources on a first-come-first-served base, i.e. the first task requesting a certain resource will be granted while all following request

map_mask	Description
L4_FP_REMAP_PAGE | L4_FP_OTHER_SPACES	Map fpage read-only in all other address spaces in which the fpage has been mapped
L4_FP_FLUSH_PAGE | L4_FP_OTHER_SPACES	Completely unmap the fpage in all other address spaces in which the fpage has been mapped
L4_FP_REMAP_PAGE | L4_FP_ALL_SPACES	Map the fpage read-only in all address spaces
L4_FP_FLUSH_PAGE | L4_FP_ALL_SPACES	Completely unmap the fpage in all address spaces

Table 1: *Map_mask* attributes' combinations

to the same resources will fail.

There is an initial address space, called σ_0 (Sigma 0) which is created after the system was booted. σ_0 gets all the physical memory except for some kernel areas. σ_0 is idempotent, i.e. all virtual addresses are the same as the corresponding physical addresses. This address space is used to allocate memory for threads upon request via mapping. σ_0 maps frames (with write permission) to the first task requesting it. Any further requests for already mapped frames will be ignored.

4 Source code analysis

In the L4 microkernel the memory management code consists of several basic parts. These are:

1. Physical memory allocation and deallocation by the kernel.
2. Virtual memory organisation.
3. Mapping virtual memory to physical memory.
4. Memory allocation and deallocation for special data structures.

In this thesis I will analyse the realisation of last two items. The algorithm of the memory mapping is realised in the `map_fpage` and `fpage_unmap` functions of the source code (file `memory.c`). The `map_fpage` function performs the mapping/granting of a flexpage from one thread to another. The `fpage_unmap` function performs the unmapping from the page table of a thread in one of four variants (see section 3.3).

There exists a general structure which is called a *mapping data base* where

data about all current memory allocations is stored. The mapping and unmapping system calls prepare all the necessary data, change the local information in the page tables of the threads involved in the mapping and invoke the functions which change the information of the current memory state in the general mapping data base.

The mapping data base is a set of many data structures organised a three structure. The detailed organisation and updating algorithm of the mapping data base is presented in the master thesis of E. Filonenko. The updating of the data base also requires the memory allocation and deallocation in order to store data structures and is also a job for the L4 microkernel.

The memory allocation and deallocation for these data structures are implemented in the `mdb_alloc_buffer` and `mdb_free_buffer` functions (source code file `mapping_alloc.c`). The `mdb_alloc_buffer` function allocates a memory buffer of a requested size. The `mdb_free_buffer` function deallocates a buffer of a specified size and at a specified address.

4.1 Types and Data structures for mapping

We need some types and data structures to present abstractions of memory organisation and L4 mechanisms. I will describe only small part of all data structures that is used in analysed functions.

- `dword_t (types.h)`
  It is the same as `unsigned int`, this type has a size of one double word (32 bits).

- `ptr_t (types.h)`
  A pointer to `dword_t`, i.e. the same as `dword_t*`, it is used as a pointer to addresses and so on.

- `tcb_t (thread.h)`
  This structure is the implementation of a thread. It contains a lot of fields, but I will use only one of them.

  ```
  typedef struct tcb_t {
  ...
  ptr_t page_dir;
  ...
  } tcb_t;
  ```

  `page_dir` points to the page directory of a thread.

- `fpage_t (ipc.h)`
  It is a flexpage type. The structure `fpage_struct_t` implements the format of a flexpage.

```
typedef struct {
    unsigned grant:1;
    unsigned write:1;
    unsigned size:6;
    unsigned zero:4;
    unsigned page:20;
} fpage_struct_t;
```

The union with a variable of `dword_t` type allows to access a fpage as the whole one. It is necessary in order to work with masks.

```
typedef union {
    dword_t raw;
    fpage_struct_t fpage;
} fpage_t;
```

- `mdb_pgshifts[] (mapping.c)`
  This is an architecture specific array defining the page sizes which must be supported. Actually, it defines the shift numbers rather than the page sizes. The last entry in the array must define the whole address space. For Intel x86 it is:

```
dword_t mdb_pgshifts[3] = {
    12,// 4KB
    22,// 4MB
    32 // 4GB (whole address space)
};
```

  So, pages of size 4KB (smallest hardware page), pages of size 4MB (a set of 4KB pages mapped to one page table of second level) and the page of 4 GB (the size of the whole address space) are supported.
  Also, there is the possibility to map 4 MB as one (virtual, not hardware) page to one entry in the page directory without using a page table of second level. There is a flag in the entry format to indicate such a usage of an entry. Such a page is called a *large* or a *super page*.

- `pgent_t (memory.h)`
  It is a structure describing an entry of a page table. `pgent_t` is implemented as a class. There are a structure implementing the format of an entry and several functions to work with that structure. I emphasise functions which I will use in the description.

  - `next`
    It admits as parameter a number of entries to add them as a shift to the current entry and computes an address of a resulting entry.

- `is_valid`
  Checks the presence bit of the entry (bit 0) and returns *true* if the page is valid and *false* if it is not.
- `is_subtree`
  Checks the current entry in the page directory either does it point to a page table – *true* or does it point to a mapping of a large page – *false*. The bit 7 in the entry format is used to indicate this information. For entries in any page table of second level it always returns *false*.
- `subtree`
  Returns the pointer to the page table of the entry.
- `make_subtree`
  The function creates a new page table pointed by this entry.
- `clear`
  Sets the entry value to 0.
- `address`
  Extracts an address of a page specified by entry.
- `is_writable`
  Returns a value of the write permission flag.
- `set_writable`
  Sets the write permission flag to *true*.
- `set_entry`
  Takes an address, a write flag and a current level value as arguments. It packs the address and flags into the entry format.

4.2 Function `map_fpage`

This function provides the mapping (or granting) of a flexpage from one thread to another. The goals of the function are:

- to find all valid pages in the sender's address space described by the flexpage of the sender, then find corresponding places in the receiver's address space (limited by the receiver's flexpage) to be mapped;
- to call the mapping function in order to change the memory data base (`mdb_map` from `mapping.c`) for each of the page pairs found.

There are several cases depending on the sender and the receiver fpage sizes:

- The size of the sender's and receiver's fpages are the same and equal 4 KB. The task of the function is to find corresponding entries in the sender's page table and the receiver's page table and invoke the function which changes the mapping data base.
- The size of the sender's and receiver's fpages are the same and equal 4 MB. In this case it is necessary to find corresponding entries in the page directories of both the sender and the receiver and map all entries from the page table pointed to by that entry at the upper level.
- If the sizes of sender's and receiver's fpages are different then mapping is performed using the hot-spot address (see 3.2.1) specified as an input parameter (this realisation of the function allows only the situation when the fpage size of sender is less or equal than fpage size of the receiver).
- All other cases are built on the base of two first situations (more detailed in the section 4.2.2).

4.2.1 Input parameters

The function takes as parameters the following data:

- `tcb_t *from` - pointer to the sending thread
- `tcb_t *to` - pointer to the receiving thread
- `dword_t base` - the hot-spot address of the receiver's flexpage
- `fpage_t snd_fp` - a flexpage of the sender
- `fpage_t rcv_fp` - a flexpage of the receiver

In order to perform a correct mapping, some conditions must hold:

1. The fpage size of the sender must be less or equal than the fpage size of the receiver.
2. Any fpage size cannot be less than the minimal hardware page size (4 KB).
3. The mapping cannot be performed into a special address space area starting from an address defined by `TCB_AREA` constant, except for if the sending thread is σ_0. This area is used to store information about threads in the system.

4.2.2 Description

The realisation of this function is very complex and is not clear at first sight. To gain insight of the algorithm it is helpful to divide the whole code into smaller blocks (see Figure 8). The explanation of transition conditions are given in the blocks' descriptions.

Now let us provide a description of the actions performed in each block.

Block 0

This block performs the necessary processing of the input variables.

1. Fixing the size of the sender fpage. The value given is the logarithm of the page size, i.e. 12 for a 4 KB page.

   ```
   f_num = snd_fp.fpage.size;
   ```

2. Getting an offset (the hot-spot address) to dispose an address space specified by the sender's fpage inside an address space specified by the receiver's fpage. This is performed by aligning `base` using a mask.

   ```
   base &= base_mask(rcv_fp, f_num);
   ```

   The work of this function is presented in Figure 9. If the sender's fpage size is 2^s and the receiver's fpage size is 2^r, then the resulting mask is $0^{32-r}1^{r-s}0^s$.

3. Getting the aligned address of the beginning of the sender's address space to be mapped by applying the mask $1^{32-s}0^s$. This action is performed by the function `address`. It takes two arguments: a fpage structure to get an address to be aligned and a size of a border. For convenience I will use "from" address for the current sender's address inside the fpage and "to" address for the receiver's respectively.

   ```
   f_addr = address(snd_fp, f_num);
   ```

4. The same function is used in the computation of a starting address inside the receiver's address space. An address is aligned by the sender's fpage size and the middle part of an address replaced by `base` (the hot-spot).

   ```
   t_addr = (address(rcv_fp, f_num) & ~base_mask(rcv_fp, f_num)) + base;
   ```

5. Setting bit 0 in the "from address" to indicate a granting operation for further use.

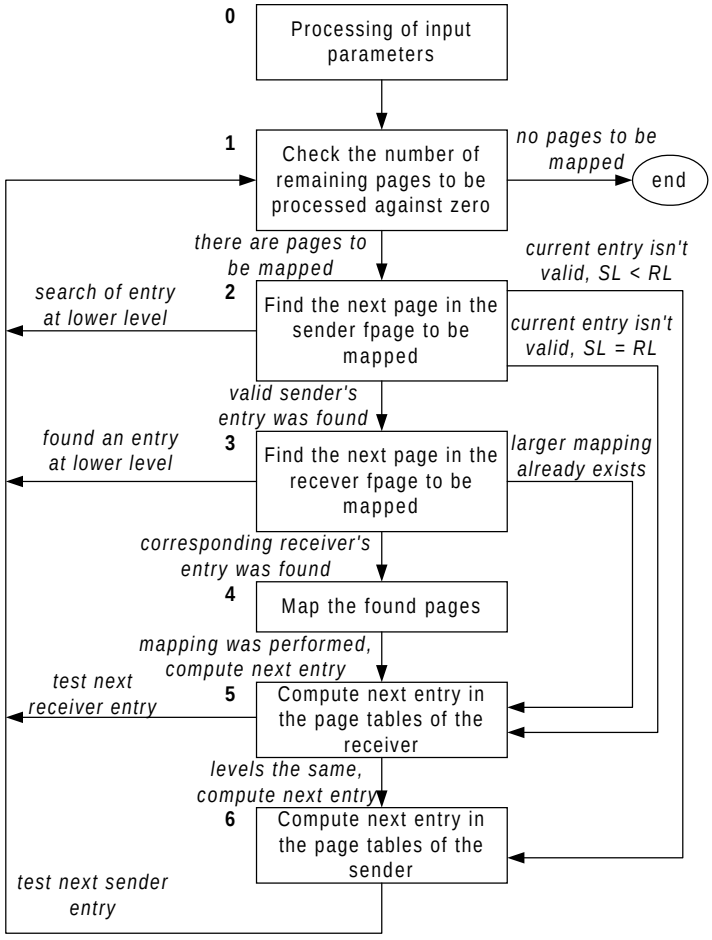

Figure 8: The program execution block-scheme

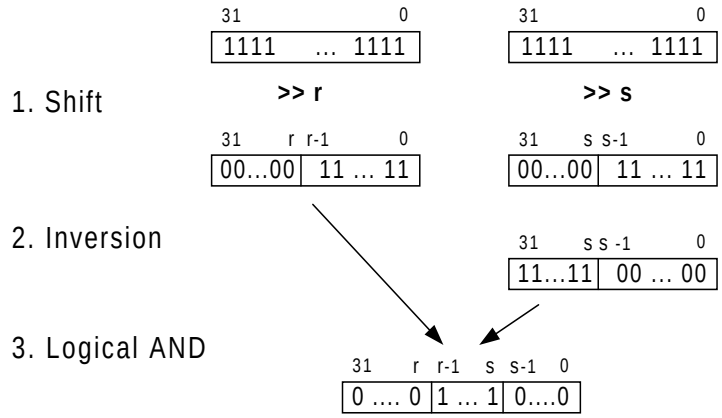

Figure 9: The `base_mask` function

```
if ( snd_fp.fpage.grant )  f_addr |= 0x1;
```

6. Finding a *pagesize* to use. As a *pagesize* I define one of possible sizes from `mdb_shifts` array. This line of the code finds the closest *pagesize* which is less or equal than the sender's fpage (except for the whole address space size).

   ```
   for(pgsize=NUM_PAGESIZES-1; mdb_pgshifts[pgsize]>f_num; pgsize--){}
   ```

   The constant `NUM_PAGESIZES` specifies a number of supported *pagesizes* and is defined as 2 (also except for 4 GB). It also can be considered as a number of page table levels. Obviously, if a fpage is more or equal than 4 MB large then `pgsize`=1, and if a fpage is more or equal than 4 KB large (and less than 4 MB) then `pgsize`=0.

7. Compute the number of memory chunks of size *pagesize* to be mapped.

   ```
   f_num = t_num = 1 << (f_num - mdb_pgshifts[pgsize]);
   ```

   Variables `f_num` and `t_num` contain a number of pages of the found *pagesize* to be mapped. These variables define the number of mapping algorithm iterations.

8. The current page table level (or index defining current page size from `mdb_pgshifts`) is 1, i.e. we start to work with the page directories of the threads.

```
f_size = t_size = NUM_PAGESIZES-1;
```

9. Computing the virtual address of the page directory of the sending thread (by adding the size of the area occupied by the kernel which is defined by the `KERNEL_OFFSET` constant), i.e. now we have an address of entry 0 in this table.

```
fpg = (pgent_t *) phys_to_virt(from->page_dir);
```

10. Find the first necessary entry in the page directory by use of the `table_index` function and the function `next` from class `pgent_t` (see Section 4.1). The `table_index`(a,n) function computes the index in the page table of level n from virtual address a. In this case it computes the index in the page directory since `f_size`=1.

```
fpg = fpg->next(table_index(f_addr, f_size), f_size);
```

11. The same actions are performed for the page directory of the receiver.

```
tpg = (pgent_t *) phys_to_virt(to->page_dir);
tpg = tpg->next(table_index(t_addr, t_size), t_size);
```

That is all that is performed in the Block 0. Now we have the start addresses in the address spaces of the sender and the receiver, the start page entries in the page directories and the number of pages of *pagesize* to be mapped.

Block 1

Block 1 performs only the comparison of the number of pages to be mapped against 0 and ends the work of the function if `f_num` or `t_num` is equal to zero.

```
while (f_num > 0 || t_num > 0) { ... }
```

Block 2

The task of this block is to find the next valid mapping in the address space defined by the fpage of the sender. The scheme of Block 2 is presented at Figure 10.

Before describing this block let us to introduce some definitions. The location of the current found entry in page tables defines the current level of the sender (SL) or the receiver (RL), i.e. if the sender's current entry is in the page directory or in a page table then SL=1 or SL=0 respectively. The *pagesize* found in the Block 0 defines the level which is needed to work with (NL).

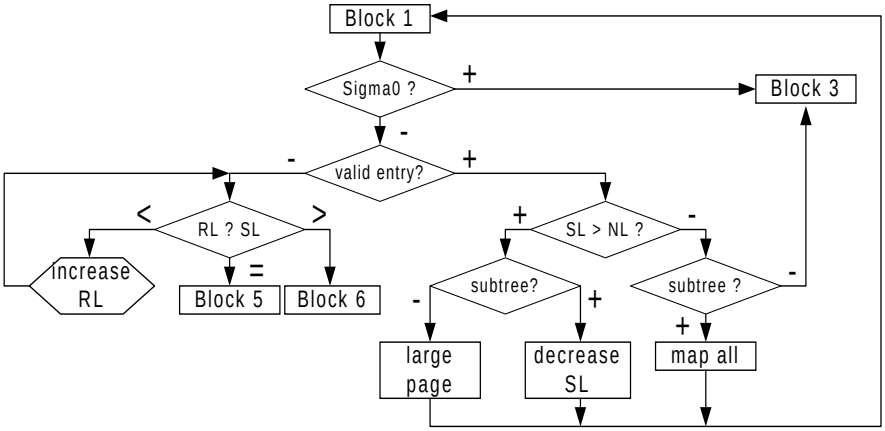

Figure 10: Block 2

1. If the sending thread is the σ_0 thread then it is not necessary to find entries in page tables, since σ_0 is idempotent. Fix the necessary level as the current level of the sender (SL=NL) and bypass the sender's page tables lookup. Go to Block 3.

```
if ( from == sigma0 )   {f_size = pgsize;}
```

2. The validity of a page pointed by a current page directory (or page table) entry is tested. If the page is not valid we need to find the next valid page in the page table of the sender.

```
if ( !fpg->is_valid(f_size) ) { ... }
```

   In both cases we need to perform some tests.

   (a) **entry is valid**
   Case 1.
   SL > NL (i.e. we have the current entry in the page directory and need to map several 4 KB pages with a general sum of sizes less than 4 MB) and the current entry points to a page table of second level. In this case we need to find the next necessary entry in that page table to decrease the current sender's level to 0 and to perform all previous tests of Block 2 for the new found entry.

```
if ( (f_size > pgsize) && fpg->is_subtree(f_size) ) {
     f_size--;
     fpg = fpg->subtree(f_size+1)->next(table_index(f_addr,
f_size), f_size);
     continue; }
```

Case 2.
SL is not less than NL (we need to map 4 MB or more), but the current entry points to a page table of second level. In this case we have to map each entry in that page table. We decrease the current level with saving some parameters (the number of remaining pages of large size and the address of the next processed entry) of upper level to have the possibility to continue the mapping at this level. And also we need an address of the page table beginning and the number of 4 KB pages to be mapped from this table (1024). After that we need to start all tests of Block 2 for the first entry in the table.

```
else if ( fpg->is_subtree(f_size) ) {
     f_size--;
     r_fpg[f_size] = fpg->next(1, f_size+1);
     r_fnum[f_size] = f_num - 1;

     fpg = fpg->subtree(f_size+1);
     f_num = table_size(f_size);
     continue; }
```

Case 3.
SL > NL, but the current entry points to the mapping of a large page, i.e. we need to map several small pages which are inside of the large page. We change the number of pages to be mapped to 1 (one large page which contains necessary smaller pages to be mapped).

```
else if ( f_size > pgsize ){
     f_num = 1; }
```

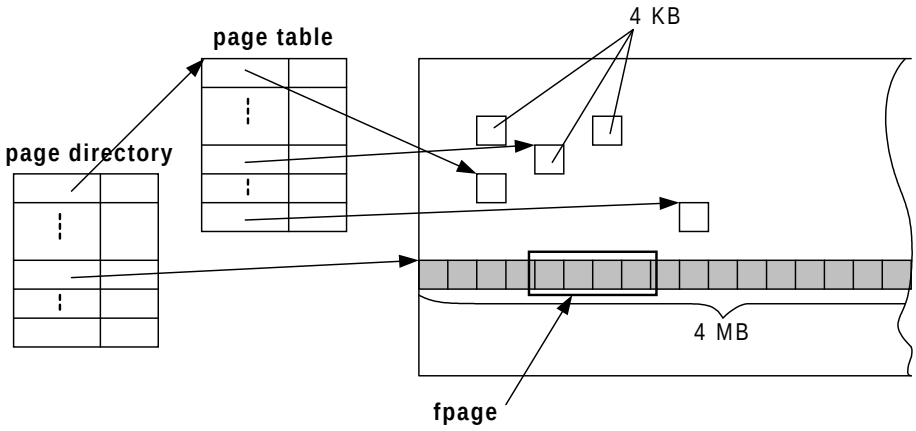

Figure 11: The fpage inside the large mapping

Case 4.
If no one of upper cases is suitable, then we have SL the same as NL and one valid entry of the sender which is ready to be mapped. Now we go to Block 3 to find a corresponding entry in the address space of the receiver.

(b) **entry is not valid**
Case 1.
SL > RL. We have entry a in the page table of the receiver and entry b in the page directory of the sender which must be used to get an entry in the page table pointed to by entry b. Obviously, if entry b is not valid then we have nothing to map to entry a (Figure 11). We need to increase the receiver's level to the sender's level, to restore receiver's data of the upper level (a number of entries to be mapped and an address of an entry to be processed next) and to find the next receiver and sender entries (see *Case 2*).

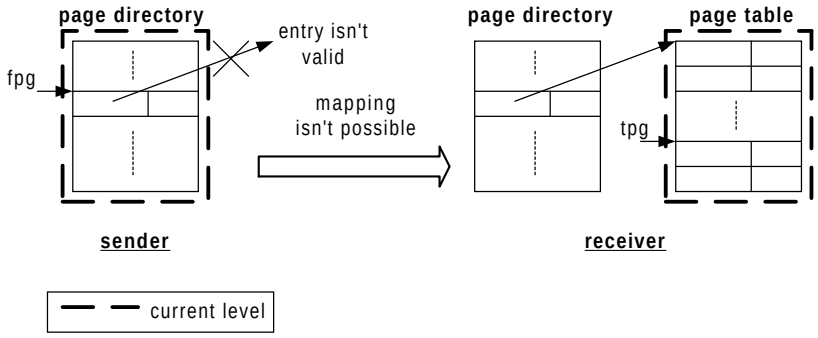

Figure 12: The invalid entry at the sender's upper level

```
while ( t_size < f_size ) {
     tpg = r_tpg[t_size];
     t_num = r_tnum[t_size];
     t_size++; }
```

Case 2.
SL = RL. Levels are the same. In this case we go to Block 5 in order to find the next receiver's entry.

```
if ( t_size == f_size )
     goto Next_receiver_entry;
```

Case 3.
SL < RL. In this case we go to Block 6 to find the next sender's entry to be mapped (and after that to Block 2 in order to test

it). The level of the receiver will be decreased later (i.e. the corresponding receiver's entry will be found later in Block 3).

```
goto Next_sender_entry;
```

Block 3

Block 3 provides the search for an entry in the page table of the receiver which is corresponding to the sender's entry (if we have reached this block we have it). The flowchart of this block is depicted in Figure 13.

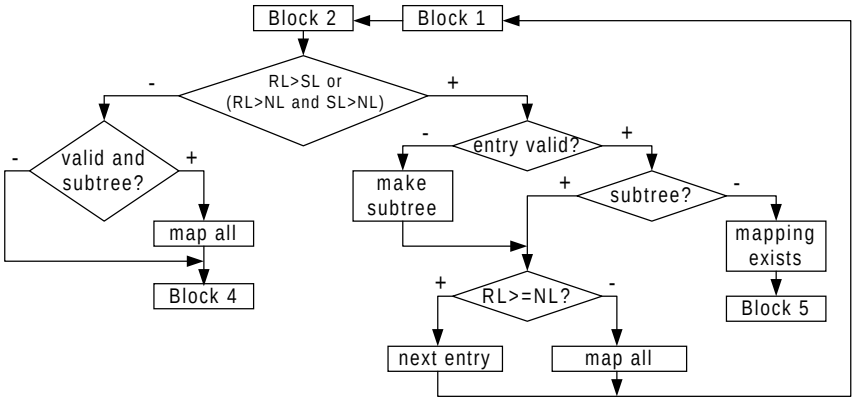

Figure 13: Block 3

1. Testing the condition: RL>SL or (RL>NL and SL>NL). This means either the current receiver's entry is in the page directory and the current sender's entry is in a page table (1st part of condition) or it is *Case 3* in point 2(a) from Block 2 but a corresponding receiver entry in the page directory (2nd part of condition).

   ```
   if ((t_size > f_size) || ((t_size > pgsize) && (f_size > pgsize)))
   ```

   If this expression is true then we need to perform the following actions:

   (a) Decrementing the current receiver's level and storing the information about the upper level: the address of the next entry in the page directory to be processed and the number of remaining page directory entries to be mapped.

   ```
   t_size--;
   r_tpg[t_size] = tpg->next(1, t_size+1);
   r_tnum[t_size] = t_num - 1;
   ```

(b) Testing the validity of page pointed to by the current entry. If a page is not valid then it is needed to create a new page table (see Figure 14), as we need to perform the mapping at the lower level.

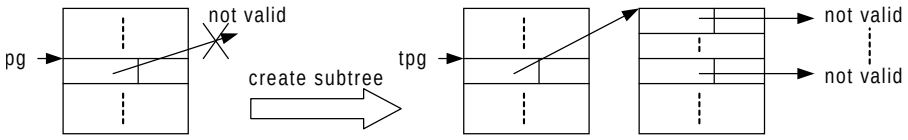

Figure 14: The new page table creation

```
if ( !tpg->is_valid(t_size+1) ) {
     tpg->make_subtree(t_size+1); }
```

If the page is valid then we test this entry for a large mapping. If we have already had a larger mapping here we cannot map and need to compute a next receiver's entry to be mapped.

```
else if ( !tpg->is_subtree(t_size+1) ) {
     goto Next_receiver_entry;  }
```

If the current entry in the page directory is valid and points to a page table and not to an existing mapping of larger size then we perform the next test.

(c) RL >= NL. We have not reached yet the necessary level. Compute the entry in the page table pointed by the current page directory entry using the current "to" address. Now we need to check current entry - return to Block 1, bypass Block 2 (as we have already had a valid sender's entry).

```
if ( t_size >= pgsize ) {
     tpg = tpg->subtree(t_size+1)->next(table_index(t_addr,
                                        t_size),t_size);
     continue; }
```

If the upper condition doesn't hold (i.e. RL < NL) then we need to map several entries from a page table pointed to by the current entry. We extract the address of the first entry and store as the number of entries to be mapped the number of entries in one page table.

```
else {
     tpg = tpg->subtree(t_size+1);
     t_num = table_size(t_size); }
```

Now we correct the next processed receiver's entry and number of entries to be processed according to where source entry is located. Also we compute a next "to" address and go to the Block 1 (hence to Block 3) to test the found entry.

```
tpg = tpg->next(table_index(f_addr, t_size), t_size);
t_num -= table_index(f_addr, t_size);
t_addr += table_index(f_addr, t_size) << mdb_pgshifts[t_size];
continue;
```

So, we have 4 MB divided into two parts: one is pointed to by n entries of the current page table and second is pointed to by 1024-n entries of the next page table (see Figure 15).

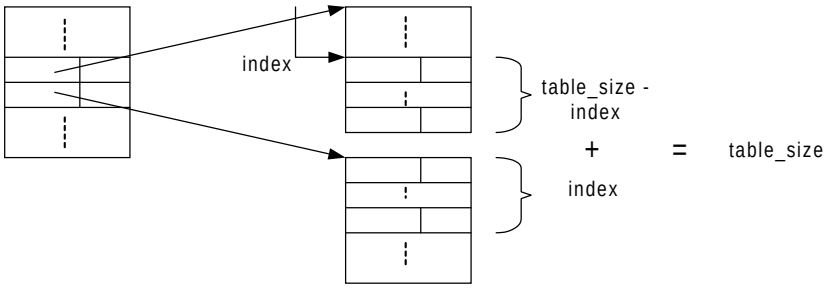

Figure 15: The partial location of a large flexpage

2. If the condition from point 1 is not fullfilled and we have a page table pointed to by the current entry, then we have to perform mapping for all entries in this table. We need to decrease the current receiver level and to store two parameters: the address of the next processed entry and the number of remaining pages to be mapped. After that we compute the address of the first entry in that page table and fix a number of pages to map (1024). We go to next point as we have found the receiver's entry which is ready to map.

```
else if ( tpg->is_valid(t_size) && tpg->is_subtree(t_size) ) {
     t_size--;
     r_tpg[t_size] = tpg->next(1, t_size+1);
     r_tnum[t_size] = t_num - 1;

     tpg = tpg->subtree(t_size+1);
     t_num = table_size(t_size); }
```

3. If none of the upper points is valid, then we have a valid sender's entry and have found a corresponding receiver's entry. Go to Block 4.

Block 4

This block invokes the function to change the mapping data base. There are several case distinctions of its execution. The flowchart is represented at Figure 15.

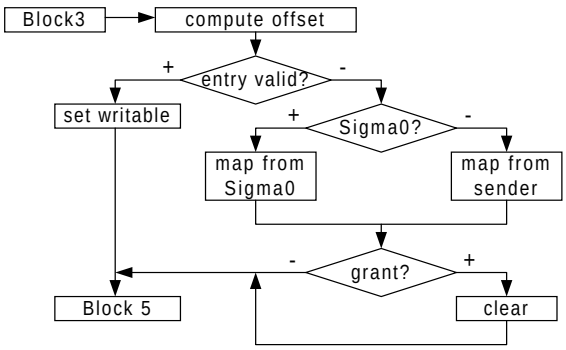

Figure 16: Block 4

1. The first performed action is an offset computation. This is necessary if the sender's small flexpage is in a mapping of large size. We need to point to the small page which will be mapped into the current recever's entry in the page table.

   ```
   offset1 = f_addr & page_mask(f_size) & ~page_mask(t_size);
   ```

2. Testing the validity of the receiver's entry. If it is valid it can be the extension of the current access rights. We can perform the right extension if:

   - It is permitted to write in this memory area or the sending thread is σ_0.
   - The sender maps this page with the write permission.

   ```
   if ( tpg->is_valid(t_size) ){
        if ((fpg->is_writable(t_size) || from == sigma0) && snd_fp.fpage.write)
        tpg->set_writable(t_size); }
   ```

   Go to Block 5 to compute next entry.

3. If the current receiver's entry **is not valid** we invoke the `mdb_map` function to change the mapping database. There are two variants, which have differences in data used as functions parameters:

   - Sending thread is σ_0

     ```
     if  ( from == sigma0 ) {
          mdb_map(f_addr + offset1, t_size, 1, SIGMA0_PGDIR_ID,
                  f_addr + offset1, (dword_t) to->page_dir, t_addr);

          tpg->set_entry(f_addr + offset1, snd_fp.fpage.write, t_size); }
     ```

   - Sending thread is not σ_0

     ```
     else {
          mdb_map(fpg->address(f_size)+offset1, t_size, 1,
                 (dword_t) from->page_dir, f_addr,(dword_t) to->page_dir,t_addr);

          tpg->set_entry(fpg->address(f_size) + offset1,
                       fpg->is_writable(f_size) && snd_fp.fpage.write, t_size);}
     ```

   In both cases we change the mapping data base and pack the address of the mapped page in the main memory and flags in the entry format.

4. Also we need to check the granting bit which can be set in Block 0. If a page was granted we clear all information about it form the sender's entry and delete this mapping from the mapping data base.

   ```
   if ( f_addr & 0x1 ) {
        fpg->clear(f_size);
        flush_tlbent((ptr_t) f_addr); }
   ```

5. The mapping of the found pair of entries was performed. Now we need to compute the addresses of the next possible pages to be mapped.

Block 5

This block computes the next entry of the receiver. The flowchart of the algorithm is shown in Figure 16. Tasks of this block are:

1. Computing the next "to" address and the number of remaining receiver's pages.

   ```
   Next_receiver_entry:
        t_addr += page_size(t_size);
        t_num--;
   ```

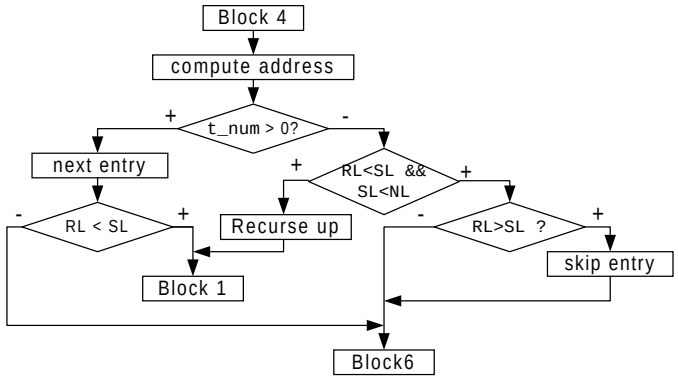

Figure 17: Block 5

2. Checking the number of remaining pages against zero. If we have pages to be mapped then go to the next entry in the corresponding table. If RL<SL we need Block 1 (and hence Block 2) to find the next valid sender's entry, else we go to Block 6 to compute the address of the next sender's page to be mapped.

```
if ( t_num > 0 )  {
     tpg = tpg->next(1, t_size);
     if ( t_size < f_size )
          continue; }
```

3. If the number of remaining pages is zero it may be that we have mapped all pages from one page table and need to check the number of entries at the upper level (in the page directory) to be mapped. We need to restore the data that was stored before: the number of entries at the upper level to be processed and the address of the next processed entry.

```
else if ( (t_size < f_size) && (f_size < pgsize) ) {
     tpg = r_tpg[t_size];
     t_num = r_tnum[t_size];
     t_size++;
     continue; }
```

4. If RL>SL, then we have a mapping of the large size in the receiver's flexpage, but the corresponding area of the sender is in the small mappings - we need to skip the next sender's entry.

```
else if ( t_size > f_size ) {
     f_addr += page_size(t_size) - page_size(f_size);
     f_num = 1; }
```

5. After point 1. (if RL<SL is not true) and point 3. we go to Block 6. For all other variants we go to the start of cycle to Block 1.

Block 6

Block 6 performs similar operations as in Block 5 but for the sender. The block-scheme of this block is at Figure 16.

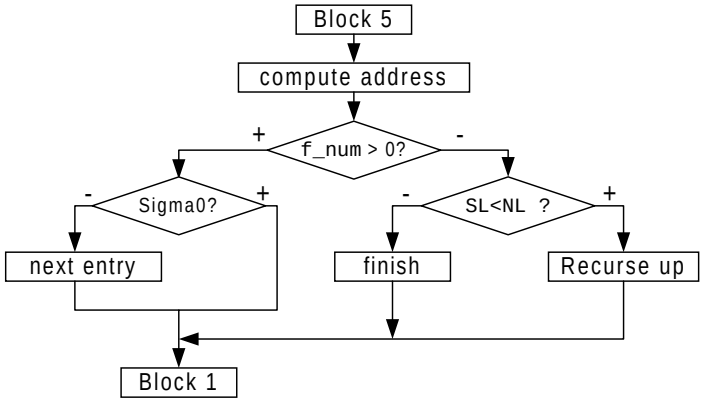

Figure 18: Block 6

1. First, we compute the next address in the sender address space (by adding the size of a corresponding page) and decrement the number of pages to be mapped.

```
Next_sender_entry:
     f_addr += page_size(f_size);
     f_num--;
```

2. If we have pages to map we get the next sender's entry and go to Block 1. If the sender is σ_0, then we don't need to compute the next entry since σ_0 is idempotent.

```
if ( f_num > 0 )  {
     if ( from != sigma0 )
          fpg = fpg->next(1, f_size);
          continue; }
```

3. If the number of pages to be mapped is zero and SL<NL then we have mapped all pages from one page table. Now we need to check if we have to map something else. The number of pages which we need to map from the upper level and the address of the next processed entry is restored from the variables where they were stored before.

```
else if ( f_size < pgsize ) {
     fpg = r_fpg[f_size];
     f_num = r_fnum[f_size];
     f_size++; }
```

4. If none of these situation is suitable we have finished to perform mapping for the given flexpages. Set the number of pages to be processed to zero and go to Block1 and, hence, to the end.

```
else { t_num = 0;}
```

4.3 Function `fpage_unmap`

This function is an inverse function to `map_fpage` and provides the unmapping of the address space specified by a flexpage. The function takes as input parameters:

- `pgdir` - the address of the page directory where the flexpage can be found
- `fpage` - the flexpage to be unmapped
- `mapmask` - a mapping mask (see section 3.3)

The main goal of the function is to find all valid pages in the address space described by the fpage and delete the information about them from the page tables of this thread and from the mapping data base. The work of the function is presented in Figure 19.

The unmapping algorithm works in the following way:

- First, the initialisation of parameters and the information extraction from input data are performed.
- Then we set the counter of the processed pages to zero. Check the validity of each entry until the first invalid one. Increment the counter for each valid entry found.
- After detection of the first invalid page (or page table) we flush all valid pages which were found before. The changing of the mapping data base is performed by invocation of the `mdb_flush` function.
- Invalid pages are passed, the counter is set to 0 and the cycle is continued until all pages are unmapped.

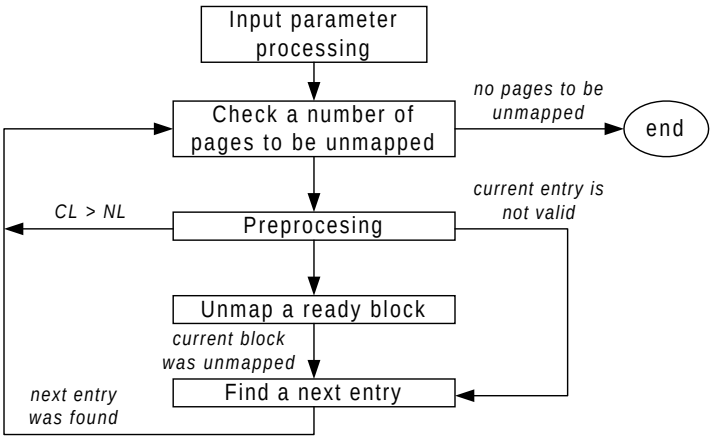

Figure 19: The `fpage_unmap` execution

The function works correctly if the fpage size is more or equal than the hardware page size.

Block 0

Actions performed in this block:

1. Getting the fpage size from the input parameter and the aligned address of the fpage beginning.

```
num = fpage.fpage.size;
vaddr = address(fpage, num);
```

2. Searching the minimum *pagesize* supported in the system to represent the flexpage as a finite number of pages of the particular size (either 4 KB or 4 MB).

```
for (pgsize = NUM_PAGESIZES-1; mdb_pgshifts[pgsize]>num; pgsize--) {}
```

3. Compute the number of pages of found *pagesize*.

```
num = 1 << (num - mdb_pgshifts[pgsize]);
```

4. Set the upper level in the page table structure as the current one.

```
size = NUM_PAGESIZES-1;
```

5. Find the necessary entry in the page directory using the address of the fpage beginning.

```
opg = pg = ((pgent_t *) phys_to_virt(pgdir))->
                          next(table_index(vaddr, size), size);
```

6. Initialize the counter of processed pages as zero.

```
cnt = 0;
```

7. Go to Block 1.

Block 1

Block 1 performs only the comparison of the current number of pages to be unmapped against zero. If all pages are unmapped (`num`=0) the function execution is finished.

Block 2

In this block the following actions are performed:

1. Compare the current processing address inside the fpage with a special kernel area address defined by the `TCB_AREA` constant. If we try to unmap in this region the function execution is stopped.

```
if ( vaddr >= TCB_AREA )
     break;
```

2. Compare the current level (CL) with the necessary level (NL) to be worked with defined by `pgsize`. If we have not reached the necessary level then go to point (a), else go to point (b).

```
if ( size > pgsize )
```

   (a) CL > NL

   1. If the current entry in the page directory is not valid, we end the function execution since we don't have pages to be unmapped in the memory.

```
if ( !pg->is_valid(size) )
     break;
```

   2. If the entry is valid but it points to a mapping of the large size (we need small pages) then we need to split this mapping into 1024 small mappings and to create a new page table in order to address these mappings (Figure 20).

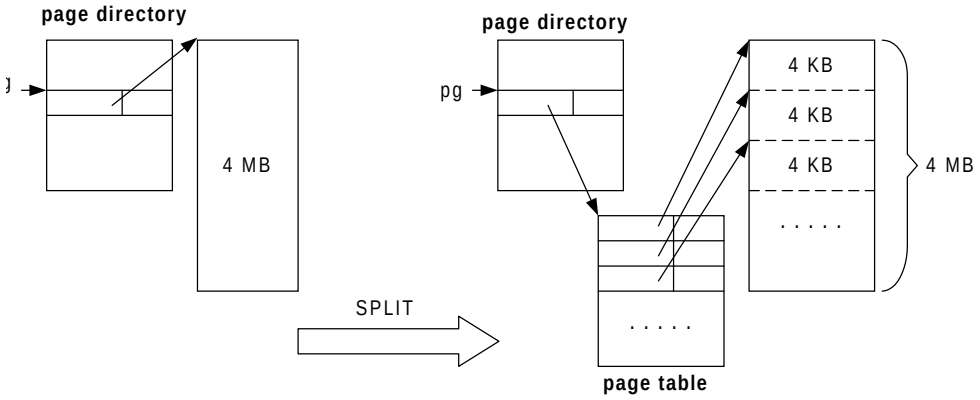

Figure 20: The large mapping splitting

```
else if ( !pg->is_subtree(size) )
      pg->split(size);
```

3. No matter if we have got here directly or after point 2, we need to decrement the current level value and to find the entry to be processed in the found page table.

```
size--;
pg = pg->subtree(size+1)->next(table_index(vaddr, size), size);
```

4. Assign `opg` the pointer to beginning of the valid pages block to the current entry. Go to Block 1 in order to test the current entry.

```
opg = pg;
continue;
```

(b) CL = NL

We have reached the necessary level. Now we need:

1. Initialise the continue flag with *false*.

```
is_cont = 0;
```

2. If the entry starting at the valid pages block is not valid or it points to a page table then we skip the initial absent mappings. Reset the counter of found contiguous valid pages and go to Block 4 to find the next entry to be mapped.

```
if ( !opg->is_valid(size) || opg->is_subtree(size) ) {
     opg = pg;
     cnt = 0;
     goto Next_entry; }
```

3. If the entry in the beginning of the valid pages block and the current entry are the same we need to find the next entry to be unmapped. This can be the case when we have skipped a found invalid page.

```
else if ( opg == pg )
     goto Next_entry;
```

4. If we have neither the situation from point 2 nor the situation from point 3 then we have the valid mapping to start. Now we go to Block 3.

Block 3

If we get here, we can start to create a block of valid pages to be unmapped.

1. Increment the counter of processed pages.

   ```
   cnt++;
   ```

2. Compute the new state of the continue flag.

   ```
   is_cont = pg->is_valid(size) && !pg->is_subtree(size) &&
             pg->address(size) == (opg->address(size) + cnt*page_size(size));
   ```

   The flag value is 1 (we can continue) if the following conditions hold:

   - The current entry points to a valid page.
   - The current processed entry doesn't point to a page table.
   - The address of the current entry equals the address of the starting entry plus the total size of found valid pages which are contiguous to each other.

3. Now we check the continue flag state. If either it is *false* (this is the case when we have encountered an invalid page or the current mapping is not physically contiguous with the previous mapping) or it is *true* and we have the last page to be unmapped then we need to interrupt the searching of valid pages and to unmap the collected block.

   ```
   if ( !is_cont || (num == 1 && is_cont) )
   ```

   We need:
   1. Add the size of one page to the size of the current block (by increasing the address) and increment the counter of pages in the block if it was the last entry to be processed.

```
if ( num == 1 && is_cont ) {
     vaddr += page_size(size);
     cnt++; }
```

2. For all situations we need to invoke the `mdb_flush` function to unmap the found block of valid pages. We transfer as parameters the starting address, the current level, the number of pages in the block, the page directory address, the total size of unmapped address space in this block and the unmapping mask (see section 3.3).

```
mdb_flush(opg->address(size), size, cnt, pgdir,
            vaddr - cnt * page_size(size), mapmask);
```

3. Now we redefine the block starting entry (for the next block) and reset the counter of pages in the block.

```
opg = pg;
cnt = 0;
```

4. If it was the last page fix the address of its beginning.

```
if ( num == 1 && is_cont )
     vaddr -= page_size(size);
```

5. Go to Block 4.

Block 4

This block computes the next entry to be processed. We can be here if we need to start the new block, or compute the next entry in the current block. Also we check some conditions in this block and perform actions depending on the results.

1. We need to be sure that last page was unmapped. If we have a valid mapping which is not a page table but the continue flag is zero then it may be the case that the last entry to be unmapped is located in another page table (see Figure 21). We unmap it with `mdb_flush` function.

```
if ( num == 1 && !is_cont && pg->is_valid(size) &&
        !pg->is_subtree(size) ) {
     mdb_flush(pg->address(size), size, 1, pgdir, vaddr, mapmask);}
```

2. If the current entry is valid and it points to a page table then we need to unmap each page pointed by the entries from this table.

```
if ( pg->is_valid(size) && pg->is_subtree(size) ) {
```

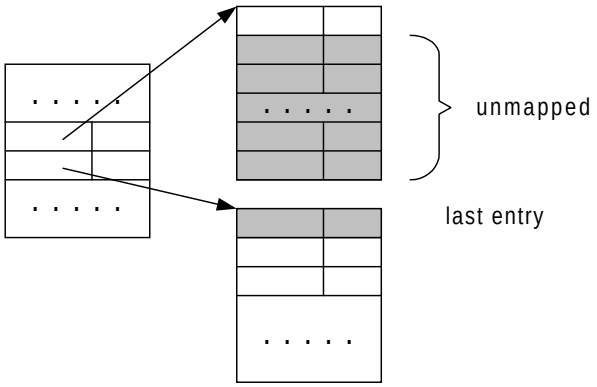

Figure 21: Last entry in the another page table

In this case we perform the following actions:

- Decrement the current level value.

  ```
  size--;
  ```

- Store some parameters from upper level (current processed entry and a remained number of pages to be processed).

  ```
  r_pg[size] = pg;
  r_num[size] = num - 1;
  ```

- Get the address of the first entry in the page table. The number of pages to be mapped is the maximal number of entries in a page table (1024). Return to Block 1 to perform all necessary tests for the found entry.

  ```
  opg = pg = pg->subtree(size+1);
  num = table_size(size);
  continue;
  ```

3. If upper cases are not suitable then we compute the next entry parameters: pointer to the next entry, increase the current processed virtual address and decrement the number of pages to be unmapped.

   ```
   pg = pg->next(1, size);
   vaddr += page_size(size);
   num--;
   ```

4. If the number of pages to be processed is zero and the current level is less than the necessary level then we have unmapped all pages from the whole page table.

```
if ( num == 0 && size < pgsize )
```

Now we need to:

- Restore the upper level parameters which were stored before.

  ```
  pg = r_pg[size];
  num = r_num[size];
  size++;
  ```

- If flags `MDB_ALL_SPACES` and `MDB_FLUSH` are set in the map mask we need to delete the processed page table.

  ```
  if ( (mapmask & MDB_ALL_SPACES) && (mapmask & MDB_FLUSH) )
        pg->remove_subtree(size);
  ```

- Set the next entry in the upper level as a starting entry of a new block.

  ```
  opg = pg = pg->next(1, size);
  ```

- Reset the counter of pages in the block.

  ```
  cnt = 0;
  ```

5. The cycle ends and we go to Block 1 to test for the last found entry.

4.4 Buffer allocation and deallocation

As it was said earlier there exists the mapping data base which stores all information about mappings in the system. This data base consists of several different data structures. In order to dynamically change the mapping data base we need to allocate these structures somewhere.

The buffers for these data structures are allocated by the `mdb_alloc_buffer` function and deallocated by the `mdb_free_buffer` function. There are buffers of different sizes. The number of sizes and their values depend on the processor architecture. Buffers of the same size are located in a common page (4 KB) which has a managing structure at the beginning. The pages with buffers of the same size form a list. In order to allocate a buffer the function performs a search in a particular list.

4.4.1 Data structures for the buffer allocation

- `mdb_buflist_t` (file `mapping.h`)
  This structure is the starting element of a list with buffers of particular size.

```
typedef struct {
    dword_t size;
    ptr_t list_of_lists;
    dword_t max_free;
} mdb_buflist_t;
```

`size` - the size of buffers in a list starting from this structure;
`list_of_lists` - a pointer to the first element (page with buffers) of the list;
`max_free` - the maximal number of buffers of the specified size in one page;

- `mdb_buflists[]` (file `x86\mapping.c`)
The `mdb_buflists[]` array defines the buffer sizes that will be needed by the mapping data base, these sizes depend on the page sizes to be supported, and also on the size of the structures involved in the mapping data base. For efficiency reasons, the first entry should define the buffer size that will be allocated most frequently. The array should be terminated by defining a zero buffer size.

```
mdb_buflist_t mdb_buflists[4] = {
    { 12 },      // Mapping nodes
    { 8 },       // ptrnode_t
    { 4096 },    // 4KB array
    { 0 }
};
```

The array initialisation creates the starting elements for each list of all sizes declared in the system. During the initialisation only the first field (`size`) is defined, the other fields are defined during the allocating.

- `mdb_mng_t` (file `mapping_alloc.c`)
This structure is used to manage the context of one page where buffers are located.

```
struct mdb_mng_t {
    mdb_link_t    *freelist;
    dword_t        num_free;
    mdb_mng_t     *next_freelist;
    mdb_mng_t     *prev_freelist;
    mdb_buflist_t *bl;
};
```

`freelist` - a pointer to the next free buffer at the current page;
`num_free` - the number of free buffers at the page;
`next_freelist` - a pointer to the next page with buffers of the same

size;
`prev_freelist` - a pointer to the previous page;
`bl` - a pointer to the structure at the beginning of a list.

- `mdb_link_t` (file `mapping_alloc.c`)
  This structure is used for the connection of free buffers in one page with each other.

```
typedef struct mdb_link_t mdb_link_t;
struct mdb_link_t {
    mdb_link_t *next;
};
```

Field `next` points to the next free buffer in the page.

4.4.2 Function `mdb_alloc_buffer`

The function provides the allocation of a new buffer with a size specified as input parameter. The correct execution only is possible when the required buffer size is supported in the system. In order to allocate a new buffer it is necessary to perform the following actions:

1. Find the starting element of the list with the needed buffer size.

```
for ( bl = mdb_buflists; bl->size != size; bl++ )
```

2. If the requested buffer size is more than the value specified by the `MDB_ALLOC_CHUNKSZ` constant (4096 for x86) then the allocation of the memory is performed by another kernel function.

```
if ( size >= MDB_ALLOC_CHUNKSZ )
     return kmem_alloc(size);
```

3. Get a pointer to the managing structure of the first page in the list.

```
mng = (mdb_mng_t *) bl->list_of_lists;
```

4. Now we have two cases:

   - `mng == NULL`
     We allocate the first buffer at all or all buffers were deallocated before. We must create a new page for buffers of this size and initialise it. After that we must connect the list starting element with the created page.

     First page creation

     1. Allocate one page from memory using `kmem_alloc` function.

```
mng = (mdb_mng_t *) kmem_alloc(MDB_ALLOC_CHUNKSZ);
```

2. Set the `freelist` pointer to the first free buffer in this page (Figure 22).

```
mng->freelist = (mdb_link_t*)((dword_t) mng + MDB_ALLOC_CHUNKSZ
                                            - bl->max_free*size);
```

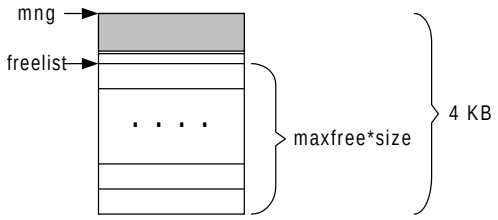

Figure 22: New list element creation

3. Initialise the number of free buffers in this page with the maximal number of buffers defined for this buffer size, which is defined as `max_free`=$\lfloor \frac{\text{4 KB - sizeof(mng\_t)}}{\text{size}} \rfloor$. Force the absence of a previous page.

```
mng->num_free = bl->max_free;
mng->prev_freelist = (mdb_mng_t *) NULL;
```

4. Connect the managing structure of the page with the corresponding list starting element.

```
mng->bl = bl;
```

Initialisation of buffers (see Figure 23)

1. Get the address of the last buffer at the page.

```
ed = (mdb_link_t *) ((dword_t) mng + MDB_ALLOC_CHUNKSZ - size);
```

2. Set the `next` pointer in each linking structure to the next free buffer.

```
for ( b = mng->freelist; b <= ed; b = n ) {
     n = (mdb_link_t *) ((dword_t) b + size);
     b->next = n; }
```

3. The linking structure of the last buffer doesn't have a buffer to point to, hence set it to `NULL`.

```
p = (mdb_link_t *) ((dword_t) b - size);
p->next = (mdb_link_t *) NULL;
```

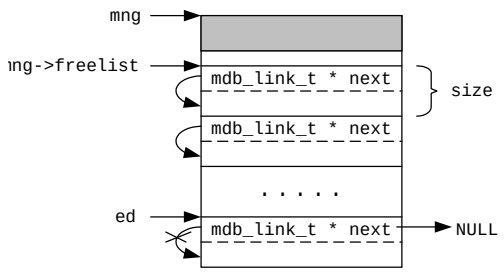

Figure 23: Buffer initialisation

Update buffer list

1. We have created a new page and this page is sole. Hence, we don't have a next page with buffers. The next line is equivalent to `mng->next_freelist=NULL`.

```
mng->next_freelist = (mdb_mng_t *) bl->list_of_lists;
```

2. Connecting the starting element of the list with the created page.

```
bl->list_of_lists = (ptr_t) mng;
```

3. Set the pointer to the previous page of the next page in the list. This condition is redundant as we don't have a next page.

```
if ( mng->next_freelist ){
    mng->next_freelist->prev_freelist = mng; }
```

Now we have a page with maximal number of free buffers and can allocate them. The allocation is depicted in Figure 24.

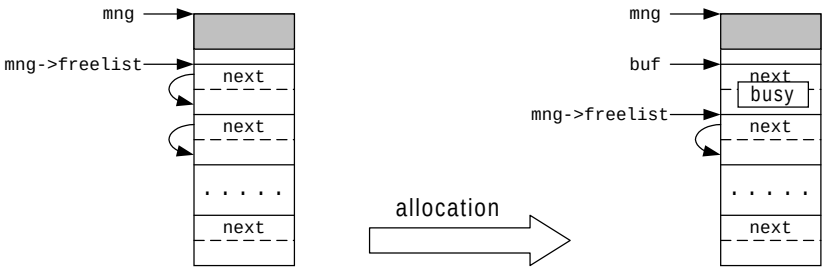

Figure 24: The buffer allocation

- `mng != NULL`
  We have a page with free buffers and need to allocate one of them.

  1. We need to find a buffer that will be allocated and delete it from the free buffers list.

  Define the first free buffer in the page as return buffer. Set the `freelist` pointer to the next buffer and decrement a number of free buffers in the page.

```
buf = mng->freelist;
mng->freelist = buf->next;
mng->num_free--;
```

  2. It is possible we have allocated the last free buffer from this page. In this case we need to remove this page from the list of pages since it doesn't have any free buffers left (Figure 25):

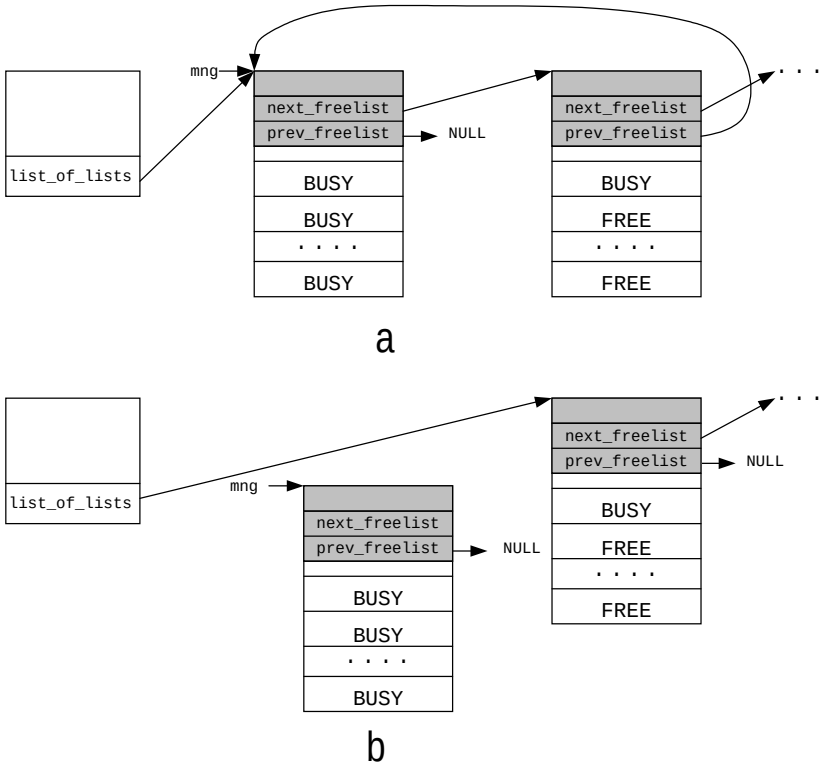

Figure 25: Removing of the completely allocated page. (a) list before deletion, (b) list after deletion

Check the number of free buffers remained in the page. If it is

zero, we need to delete this page. Take the next page from the list as new first list element. If this page exists then we need to correct the pointer to the previous page in its managing structure.

```
if ( mng->num_free == 0 ) {
     bl->list_of_lists = (ptr_t) mng->next_freelist;

     if ( bl->list_of_lists != NULL )
          ((mdb_mng_t *) bl->list_of_lists)->prev_freelist =
                                        (mdb_mng_t *) NULL; }
```

3. Return the address of the allocated buffer.

```
return (ptr_t) buf;
```

4.4.3 Function `mdb_free_buffer`

This function performs the deallocation of a memory buffer. The buffer size and the address where it is located are specified as input parameters. The function is used for mapping data base updates. The algorithm of the deallocation is as follows:

1. If we need to free more than 4 KB (`MDB_ALLOC_CHUNKSZ` constant), then deallocation is performed by another kernel function.

```
if ( size >= MDB_ALLOC_CHUNKSZ ) {
     kmem_free(addr, size);
     return;  }
```

2. Fix the address of the buffer to be deallocated.

```
buf = (mdb_link_t *) addr;
```

3. Get the aligned address of the page where this buffer is located. The `MDB_ALLOC_CHUNKSZ` constant is used as a mask. This address is also the address of the managing structure of the page.

```
mng = (mdb_mng_t *) ((dword_t) addr & ~(MDB_ALLOC_CHUNKSZ-1));
```

4. Get the starting element of the list where the page containing the buffer is included.

```
bl = mng->bl;
```

5. Restore the buffer as free in the page. We make the deallocated buffer the new first free buffer in the page. The buffer which was first before becomes second (see Figure 26). Also we need to increment the number of free buffers.

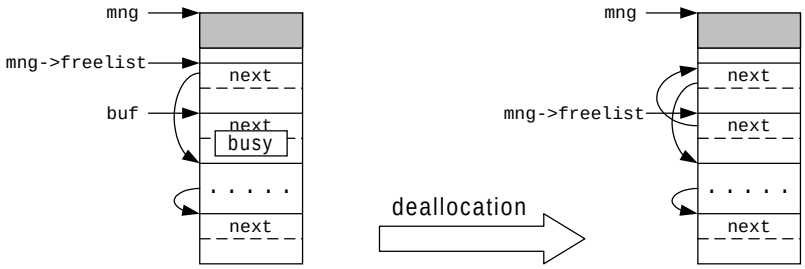

Figure 26: The buffer deallocation

```
buf->next = mng->freelist;
mng->freelist = buf;
mng->num_free++;
```

6. Now we have two special cases besides the normal deallocation.

   - **Deallocated buffer becomes the only free buffer in the page**

     Such the situation is indicated by

     ```
     ( mng->num_free == 1 )
     ```

     . In this case the page becomes non-empty from empty. We need to add this page into the corresponding list (Figure 27).

     1. If the list has already contained a page (or more) then we insert the current page as the first element. The pointer to the previous page in the old first page in the list will then point to the current page.

     ```
     if ( bl->list_of_lists )
          ((mdb_mng_t *) bl->list_of_lists)->prev_freelist = mng;
     ```

     2. Define the page which was first before as second. If we didn't have any page in the list then this field will be `NULL`.

     ```
     mng->next_freelist = (mdb_mng_t *) bl->list_of_lists;
     ```

     3. Set the absence of a previous page for the current page and connect the starting element of the list with the current page.

     ```
     mng->prev_freelist = (mdb_mng_t *) NULL;
     bl->list_of_lists = (ptr_t) mng;
     ```

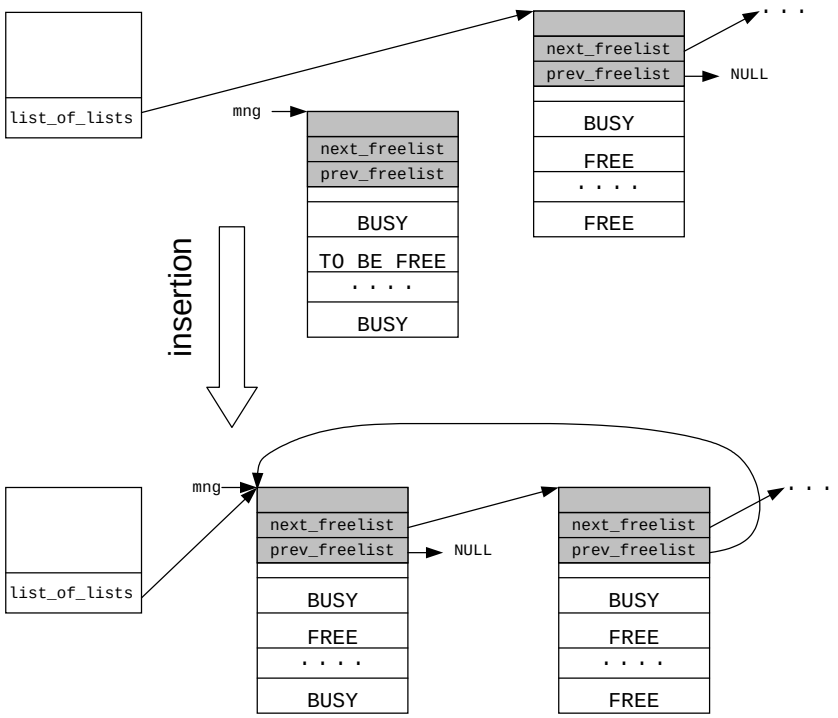

Figure 27: Insertion of the page with one free buffer

- **Deallocated buffer is the last busy buffer in the page**

  The situation is indicated by the statement

  `(mng->num_free == bl->max_free)`.

  In this case we have freed up all buffers from this page and we need to return this page to the memory management. This situation is depicted in Figure 28.

  1. If the released page has a non-null next pointer then we replace the previous-pointer in the next page with the previous-pointer in the current page.

```
if ( mng->next_freelist )
     mng->next_freelist->prev_freelist = mng->prev_freelist;
```

  2. If the current page (with last allocated buffer) was the first element in the list then the next page (for the current page) becomes the new first.

```
if ( bl->list_of_lists == (ptr_t) mng )
```

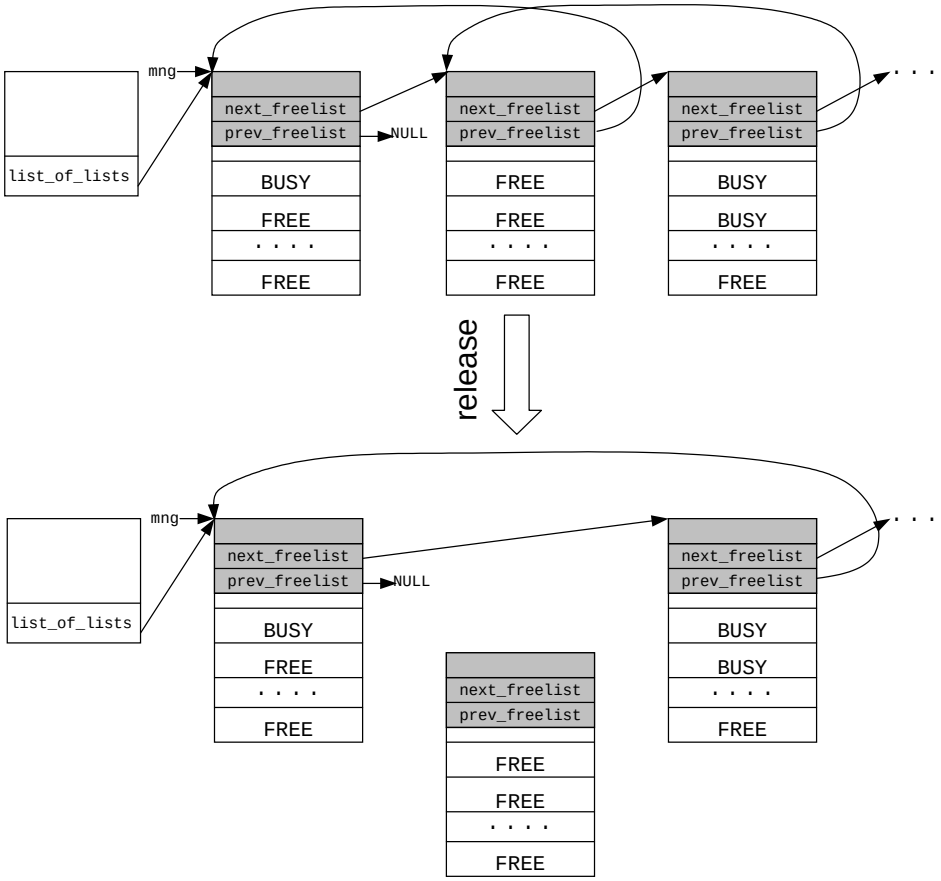

Figure 28: Deletion of the page with all free buffers

```
bl->list_of_lists = (ptr_t) mng->next_freelist;
```

3. If the current page has a previous one then we need to change the pointer to the next page in it.

```
else if ( mng->prev_freelist )
     mng->prev_freelist->next_freelist = mng->next_freelist;
```

4. Now we need to invoke the kernel deallocation function for this page.

```
kmem_free((ptr_t) mng, MDB_ALLOC_CHUNKSZ);
```

5 Conclusion

The given Master thesis concerns memory management organisation in the L4 microkernel. Several aspects of the common memory representation and some L4 memory organisation features are depicted in this paper.

Also, in this thesis I have performed the description of several memory management functions which imply data exchange using flexpages and memory allocation/deallocation to store special structures for the mapping data base.

The data structures and data types which are involved in the functions implementation are also presented in this paper. The source code of the functions is described in many details. The large functions are divided in smaller blocks and the flowcharts for each block as well as the description are given.

This thesis can be useful as the documentation for the analysed part of the source code of the L4 microkernel as well as a base for the further works in this area. The planned further works concern function specification and their correctness proofs. So, the given thesis can be helpful to offer correctness criteria for these algorithms and implementations, to create the functions specification and so on.

Bibliography

1. Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 14th ACM Symposium on OS Principles*, Copper Mountain, CO, USA, December 1995

2. The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor Architecture, Programming, and Interfacing edited by Barry B. Bray, Prentice Hall, 2002

3. Alan Au, Gernot Heiser. L4 User Manual. School of Computer Science and Engineering The University of New South Wales Sydney 2052, Australia, 1999

4. Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 Reference Manual - MIPS R4x00. School of Computer Science and Engineering, University of New South Wales, Sydney 2052, Australia, 1997

5. Gernot Heiser. Inside L4/MIPS - Anatomy of a High-Performance Microkernel, 2001

6. Bjarne Stroustrup. The C++ Programming Languge. Addison-Wesley, 2000

7. Source Code of the L4 microkernel. www.L4ka.de, www.L4ka.org

This Master Thesis has been written on my own without any unpermitted help and using mentioned materials only.

Elena Petrova, June 2003