Operating Systems 2015 Assignment 3: Virtual Memory

Deadline: Sunday, May 3 before 23:59 hours.

1 Introduction

Each process has its own virtual memory address space. The pages allocated in this virtual memory area are either backed by a page in physical memory (RAM) or by a page stored on secondary storage (such as a hard disk)[1]. The virtual memory (VM) subsystem of an operating system kernel keeps track of the virtual memory address space of each process and where the physical pages (frames) are located. Furthermore, it handles allocation and deallocation of pages.

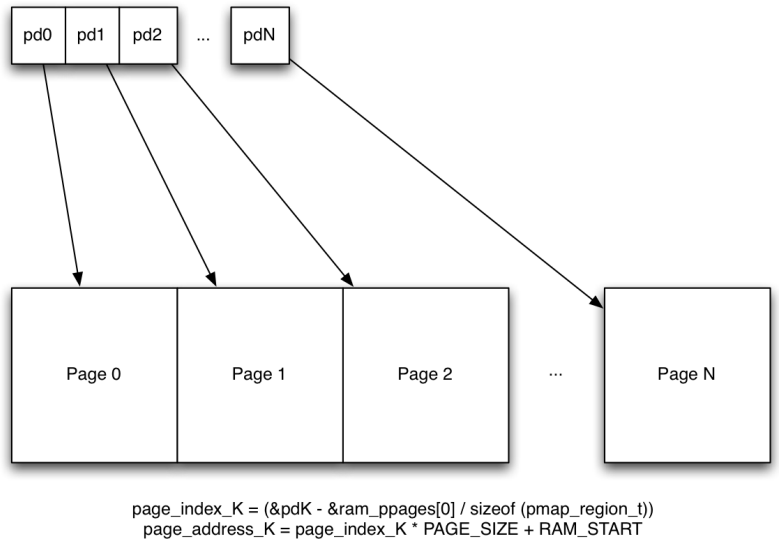

Figure 1: Illustration of current VM system. The page descriptors (type `pmap_region_t` in the array `ram_pages` (shown here as `pd0`...) are mapped using simple equations between the position of the page descriptor in the `ram_pages` array and the physical address of the physical page.

The VM implementation you will be provided with is far from optimal, for example, the physical page tracking system (that tracks which physical pages are allocated and which are not) is based on one linked list of free and one linked list of used pages. The information about these pages is stored in page descriptors. There is one page descriptor object per RAM page (which are 4KiB each on ARM), and there is a simple linear translation between the address of the page descriptor and the page that it represents. That is, one page descriptor object is needed per physical page available in the system. This is suboptimal for several reasons. Primarily, the method at the moment requires a lot of memory (about 1 MiB) to store all the page descriptors. Secondly, allocating new contiguous segments may be very slow in case of memory fragmentation as the list of free pages needs to be scanned for a contiguous sequence of pages. It also slows down allocation of non-contiguous memory as the system needs to traverse and reserve the same number of page descriptors that are needed to be allocated. There are better ways to do this! And this will be your task for this assignment. In short, you will improve the physical page allocation system by (partly) implementing the basic algorithm used in Linux.

VM programming is about memory allocation, it is in many cases necessary to carry out unsafe casts and data reinterpretation. This is the main reason that operating systems are written in a low-level language like C, that supports type coercion. Java, for instance, does not support this

[1] However, note that the kernel we use for this lab does not support paging to secondary storage.

kind of type coercion. An interesting problem when writing VM code is that the VM needs to allocate data structures to bookkeep allocated and available pages. A chicken and egg problem, which we will also have to solve in this assignment.

2 Requirements

You have to modify existing operating system source code written in C and therefore you will be writing C code. Your submissions must adhere to the following requirements:

- Submit the source code of the operating system with your functioning Virtual Memory implementation. The requirements are:
  - The VM system should be robust so that `mmaptest` can be repeatedly executed successfully.
  - It must be possible to successfully run `mmaptest2` as the first process from the shell after booting.
  - The code must implement the physical page allocation system as outlined in section 5 below.
  - The code should properly allocate pages for use by the VM system to store administrative data.
  - The code must make use of region splitting.
  - The should use 17 levels of region lists, so the highest level represents all memory on the beagle board (2^{16} pages = 256MiB). Note that level 0 represents single page regions.
  - It should be possible to both allocate and free pages.
- Your submission should include a small README file, which details the data structures you modified and/or designed for the assignment and describes how you solved the problem of allocating new physical pages for storing the physical page descriptors.

3 Submission and Grading

You may work in teams of at most 2 persons. A single tar file should be handed in of the modified operating system containing the improved VM subsystem. To hand in our work, remove any object files and binaries by removing the build directory. *Make sure that the files you have modified contain your names and student IDs.* Create a gzipped tar file of the source code directory:

```
tar -czvf assignment3.tar.gz assignment3/
```

Mail your tar files to *krietvel (at) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2015 Assignment 3". Include your names and student IDs in the e-mail.

Deadline: We expect your submissions *before* by Sunday, May 3 before 23:59. No exceptions; deliveries after the deadline *will not be graded!*.

The grade is determined based on whether the program correctly implements the functionalities listed in the specification above and whether the source code looks adequate: good structure, consistent indentation, error handling, correct memory handling and comments where these are required. Comments are usually required if the code is not immediately obvious, which often means you had to make a deliberate decision or trade-off. Document these decisions, trade-offs and why in the source code. Commenting on the obvious is superfluous and bad style. Note that we may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.

The maximum grade that can be obtained for this assignment is 10. The points are distributed as follows: Code Layout & Quality, README file (1.5 / 10), Correctness (kernel boots and can complete test cases) (1.0 / 10), Initialization (2.0 / 10), Splitting & Allocation (3.0 / 10), Page Stealing (2.5 / 10).

4 Kernel

We will use the same kernel as with the second assignment, however, you will be provided with a new starting point. For more information about the kernel and programming language, please see the text of the second assignment and the additional information on the course website.

5 Assignment

The assignment is to modify the page allocation system in the Virtual Memory (VM) subsystem so that the current big global array of page descriptors and linked lists of used and free pages are no longer needed. There are several potential solutions to this problem. We want you to improve the physical page allocation system by implementing the basic algorithm used in Linux.

In the Linux kernel, the free physical pages are stored in a region list, that stores one page descriptor entry per free page region. The page regions are in turn aligned and sized at power of 2 sizes, where there is one list per size quanta. For example, one list of page groups of 2^{10} pages, one of groups of 2^9 pages and so on (aligned on their own sizes).

The difficulty in this approach lies with splitting and merging of page groups. For example, if there are no free groups of size 8, but there is a free group of size 9, then the free size 9 group should be split into two size 8 groups. When a group is split in two, a new page descriptor has to be created to point to this new group. This in turn may require the allocation of an additional page of page range descriptors.

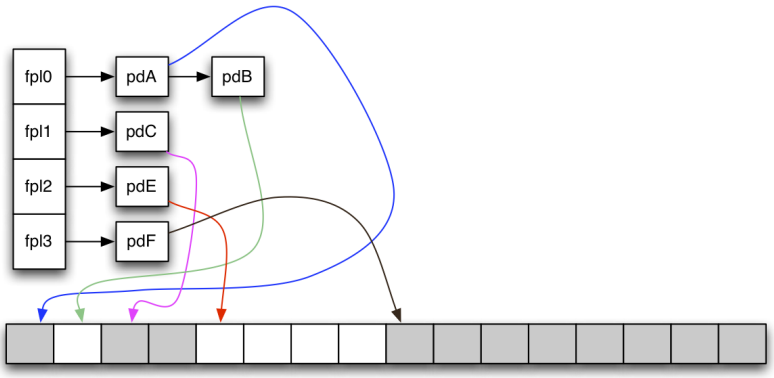

Figure 2: Illustration of VM system you will implement. The page descriptors (type `pmap_region_t`) are allocated as needed. In the illustration you see four page lists, where fpl0 represents 2^0 sized free page blocks, `fpl1` represents 2^1 sized free page blocks, and so on. In this case, a descriptor in `fpl4` (not in the picture) has been split into two `fpl3` descriptors, the first `fpl3` descriptor in turn was split to two `fpl2` descriptors, the first of who was split into two `fpl1` descriptors and in turn split into two `fpl0` descriptors.

After a brief introduction of the VM layer that is implemented in the kernel, we will briefly describe the steps that have to be taken to come to a solution.

6 VM System Overview

The VM system is divided into two layers (with some minor cross layer interactions). The functions and structures are divided into two namespaces, the `vm` namespace and the `pmap` namespace. The `vm` namespace contains the public access functions (that the rest of the kernel uses) and the functions and types required for managing virtual address allocation. The `pmap` namespace contains the functions and structures used for managing the physical RAM pages. All of the platform independent code is available in the file `kernel/src/vm.c`. In addition to this, an important architecture-specific function exists. This is the function `hw_map`, defined in `kernel/src/arch/arm/mmu.c`, which is used to for example map in page tables for processes.

6.1 Types

Essentially there are three very important types in the VM system, the `vm_map_t` type represents a virtual memory address range. For example, the user space (in one application) goes from 0x00001000 to (non inclusive) 0x80000000, this region is represented by one `vm_map_t` object. The map objects contain a list of `vm_region_t` objects, each `vm_region_t` object represents a mapped virtual memory address region within the map object. The `pmap_region_t` object represents a physical memory region (mapped or not). Each `pmap_region_t` object represents one page in the starting point you received, your task will be to ensure that a `pmap_region_t` can point to multiple consecutive pages. The pmap regions contain a few fields, both flags and `ref_count` are unused at present. The `logsize` field defines the size of the region in $\log_2$(pages), since all regions consist of one page in starting point, the `logsize` will be set to 0. In order to complete the assignment, you might have to add additional fields to `pmap_region_t`.

6.2 Bookkeeping

In the starting point, there is an array called `ram_pages`, this array consists of a `pmap_region_t` object per physical page in the system. The elements of this array are then connected in a large list called `free_ppages`. When a region of physical pages is allocated, the page range for that region is removed from the `free_ppages` list and appended to the `used_ppages` list (or the `wired_ppages` list when dealing with special kernel data).

6.3 Initialization

The VM system is initialized using the `vm_init` function. This function initializes the MMU (and enables it with a minimum mapping needed to run the kernel with virtual memory enabled), calculates the amount of physical RAM in the system and calls the `pmap_init` function.

`pmap_init` allocates a few pages starting at the address of the `_free_ram_start` symbol (that is set by the linker script (`kernel/src/arch/arm/bsp/beagle/beagle.ld`)). These pages are then initialized in order to start tracking the free physical pages. Note that one `pmap_region_t` is created for every page. `_free_ram_start` is the first memory address after the kernel code and data from where pages may be allocated. The symbol `_stext` is located at the start of the physical memory in the system.

6.4 Allocation

Allocation of virtual addresses is done with `vm_map`, `vm_map_align` and `vm_map_physical`.

Allocation of physical pages is done with `pmap_alloc` and `pmap_alloc_contiguous`. The first function being simple, just allocating free physical pages, and the second one allocating contiguous physical pages.

6.5 Other Functions

For a comprehensive description of other functions in the VM layer, refer to Appendix C.

7 Working Towards A Solution

In order to complete this lab you need to modify the `pmap_init`, `pmap_get_pa`, `vm_get_pmap`, `pmap_alloc_contiguous`, `pmap_alloc`, `pmap_alloc_wired` and `pmap_free` functions. You will have to add a function called `pmap_steal_region_desc`. In this section, we describe in which order the functions are best modified and give a few hints on what has to be changed. Because the VM system is an integral part of the system, it is not possible to test intermediate versions of your code, one of the other interesting properties of VM programming. You will have to complete most of the implementation before you can commence testing.

7.1 Initialization

We strongly recommend you to start with modifying `pmap_init`. Start with thoroughly understanding what this function is currently doing. After modification, the function has to map a single memory page and initialize as many region descriptors (`pmap_region_t`) that will fit on this single page. These initialized region descriptors should be added to a linked list of free (as in unused) region descriptors. You are responsible for adding this linked list.

It is very important to understand the difference between a free region descriptor and a free region. A free region descriptor refers to a region descriptor which is currently unused, that is, the descriptor is not pointing at any physical memory. A free region, is a region descriptor pointing at memory which is currently free. In this latter case, the region descriptor *is used*, but the physical memory it is pointing at is available to be mapped.

Secondly, a region descriptor has to be created that covers all system memory. Obtain a region descriptor from the list of free region descriptors you have just created and properly initialize the descriptor. This descriptor has to be inserted into the region lists (you have to add these region lists). You can assume that the value of the parameter `nrampages` always is a power of two. Finally, you compute the number of pages that are taken by kernel data (plus the page you allocated at the beginning of the function!). Allocate this amount of pages using `pmap_alloc`, so that pages which are taken by the kernel are no longer listed as free pages. Initializing the VM system this way makes things easier, because you do not have to write tedious code which will insert only non-kernel pages into the region lists taking into account power of two sizes *and alignments*.

A logical next step is to implement a function to get a free region descriptor from the linked list of free region descriptors. Note that when the linked list of free region descriptors is empty, another page to place region descriptors has to be allocated (see further down this section). In conjunction with this, implement a function which inserts a `pmap_region_t` into the region lists.

NOTE: Use 17 levels of `pmap_region_t` lists so the highest level represents all memory on the beagle board (2^{16} pages = 256MiB).

7.2 Allocation

Next, you likely want to rework the allocation functions `pmap_alloc` and `pmap_alloc_contiguous`. These functions have to allocate the requested number of pages. The return value for `pmap_alloc` should be the first entry in a linked list of `pmap_region_t` objects referring to the allocated page regions. `pmap_alloc_contiguous` must return a consecutive range of pages. You may assume the number of pages requested from `pmap_alloc_contiguous` is always a power of 2. Thus, returning a single `pmap_region_t` is sufficient. Calls to `pmap_alloc_wired` can for now simply be forwarded to `pmap_alloc`. Do not forget to rework `pmap_free` as well.

To satisfy the allocation requests, you will have to split regions. Region splitting can be done in a very simple way, essentially you need a list of free `pmap_region_t` objects. Splitting a descriptor on level N is then done by removing the descriptor from the free page list, and by taking one descriptor out of the free region object list (an unused region descriptor), you then set the fields in the two descriptors properly and append the descriptors to level $N - 1$ free page list. Now,

if there are no free descriptors you need to call the `pmap_steal_region_desc` function to make some.

We expect that both allocation and freeing of pages will work, however, you do not need to implement region merging (the inverse of region splitting).

When allocating physical pages, try to do the following: floor the page count you need to allocate to the closest power of 2 (see `bittools.h`) and then compute the 2-logarithm of this, since it is a power of two you can compute the logarithm with the count trailing zeroes routine. This logarithm can then be used to index the array that contains the list of free page regions. Obviously, if this list is empty, you need to split larger regions (or if there are no larger regions, try with a smaller region).

When you have allocated one region, you may still need to allocate more pages. You have to keep track on how many pages you have allocated so far and do the floor to power of 2 trick again until you have allocated all the necessary pages.

For a given region, it must be possible to compute the physical (starting) address of the pages described by that region. This is done by the function `pmap_get_pa`. In the current VM implementation, we can essentially decode a pmap region pointer and compute the physical address since there is one pmap region per page. This will not be possible in the new physical page management system you will implement, and you need to rework `pmap_get_pa` to return correct physical addresses for regions corresponding to your implementation.

7.3 Allocating Memory for Page Descriptors

Memory has to be allocated for the page descriptors. It is the task of the VM system to handle allocation and deallocation of pages, so it is obvious that the VM system itself cannot simply allocate memory using for example `slab_alloc()`. (Observe that `slab_alloc()` calls `vm_map()`, resulting in endless recursion). This problem is solved by handling the VM data structure allocation as a special case.

On initialization of the VM system, you have allocated a single page dedicated for storing page descriptors. As more regions are being split during runtime, more page descriptors will be in use. At some point, another page will have to be allocated to provide space for more page descriptors. To handle this special case, a free physical page has to be stolen and the page needs to describe itself. The steal function needs to find a single free physical page from the free region lists and find the first free virtual address in the kernel heap.

A good way to do this is to create a function, let's call it `pmap_steal_region_desc`, that firstly looks up a free virtual address (in the kernel heap), and secondly looks up a free physical page. This physical page may be in any of the free page lists, so take it from the smallest one. Then, map the page (only one page) with `hw_map` and initialize the following on that page: *one* `vm_region_t` descriptor describing the page itself, and the remainder of the page as `pmap_region_t` descriptors which you place in the list of free region descriptors. The extracted `pmap_region_t` entry describing the region where you just mapped in a page, must now be split so the descriptor itself only refers to one page. Do not forget to insert the `vm_region_t` object in the kernel heap.

See also the function `vm_steal_region_descriptor()` for an example of how to do this.

8 Testing

We have provided two applications to test your modified virtual memory subsystem. Firstly, `mmaptest` stresses the VM system by allocating eight pages (using separate calls to `mmap`) and freeing them in a shuffled order in a loop. Since the freeing inserts the entries at the end of the `free_ppages` list, this will reduce the number of entries that are available for contiguous page allocation substantially, because the `pmap_alloc_contiguous` function cannot find any contiguous blocks of the right size. When `mmaptest` is started the next time, there will be so much fragmentation in the `free_ppages` list that the application will fail and in this case also crash the kernel

that we have provided to you. In your adapted kernel, it must be possible to run the `mmaptest` program repeatedly without failing.

Secondly, `mmaptest2` allocates in total 1024 single pages and 512 double pages. This will require a lot of page descriptors to be allocated and will thus force your steal region descriptor code to be executed. Note that running `mmaptest2` a second time does not have effect, because all necessary page descriptors have already been allocated by then.

A The Bittools Header

In the kernel there is header file called `bittools.h`. You should study this header file and get to know it well. You will most likely find the inline functions in it very helpful.

B Linked List API

Managing virtual memory and physical memory pages will more or less force you to use linked lists. The kernel has a header file called `list.h` which contains macros for the definition of linked lists and their operations (insert, remove etc.).

The most important operations here are:

- `LIST_HEAD(T)` which expands to a list head type (a structure with a head and a tail pointer). To declare a list of say `pmap_region_t` objects, write `LIST_HEAD(pmap_region_t) mylist;`. Do not forget to set the head and tail pointers to `NULL`!
- `LIST_ENTRY(T)` which expands to a link type containing a next and previous pointer.
- `LIST_EMPTY(L)` is a predicate that checks if the list `L` is empty or not.
- `LIST_FIRST(L)` returns the first entry in the list `L`.
- `LIST_NEXT(E, LNK)` returns the next list entry after the entry `E`. `LNK` is the name of the link in the type of `E`.
- `LIST_REMOVE(L, E, LNK)` removes entry `E` from the list `L`, where `LNK` is the name of the link in the type of `E`.
- `LIST_APPEND(L, E, LNK)` appends an entry `E` to the list `L`.

C Virtual Memory Layer Overview

C.1 pmap functions

The pmap functions are the following:

- `pmap_get_pa` computes or returns the physical address associated with the page region descriptor.
- `pmap_alloc_contiguous` allocates a number of pages in contiguous physical memory. The region may optionally be aligned at a specified size. You can assume that alignment is always at power of 2 page counts. You do not need to handle wired memory in any special way.
- `pmap_alloc` allocates memory pages, make sure you return the head of a NULL-terminated list of regions.
- `pmap_alloc_wired` allocates memory pages in wired memory (non swappable), but you can essentially just forward the call to `pmap_alloc`.

- `pmap_init` initializes the physical memory tracking system. For example, you should initialize your `pmap_region_t` lists in here. The function also reserves memory for the kernel itself (i.e. makes sure that there are no physical memory descriptors that indicate that the kernel code and the initial page descriptors are free memory).
- `pmap_free` frees the physical pages associated with a specific vm region.

C.2 VM layer functions

The vm layer functions are the following

- `vm_append` appends a vm region descriptor to a virtual memory map.
- `vm_insert` inserts a vm region descriptor after a given position in the vm map.
- `vm_insert_region_descriptor` inserts a region descriptor on its correct location in the vm map.
- `vm_get_kernel_map` returns a vm map object with the given name (see the `kernel_region_type_t` enum), usually you use this to get the vm map for the kernel heap.
- `vm_get_pmap` returns the first pmap region object for the given vm region.
- `vm_get_first_free_kernel_heap_address` returns the first free address in the kernel heap that can be used to store ONE page. You can use this function when stealing pages to query for a virtual address.
- `vm_steal_page` steals a page in the kernel heap, it returns the (virtual) pointer to the stolen page, which includes the `vm_region_t` self descriptor.
- `vm_steal_region_descriptor` looks in the free region descriptor list and returns one entry in it, if the free region descriptor list is empty, the function steals a page and initializes the page contents (except the first entry which describes the page itself) as entries in the free region list and returns the first of these.
- `vm_release_region_descriptor` frees the region descriptor and inserts it in the global free region descriptor list (`free_regions`).
- `vm_get_new_region` allocates a new region descriptor without giving an address. The address will be assigned based on where the vm system can find a free virtual address that would allow *size* bytes to be allocated.
- `vm_get_new_region_at_addr` is the same as `vm_get_new_region`, but will use the user supplied address instead of looking for one. Note that *size* bytes must still fit in the region.
- `vm_init` initializes the VM system by firstly calling `hw_mmu_init` that will enable the MMU and map in the kernel, secondly it computes the total amount of RAM in the system and calls `pmap_init` with the number of pages in the system.
- `vm_is_in_region` is a predicate for checking whether a given virtual address is in a `vm_region_t` object.
- `vm_are_disjoint_regions` is a predicate that verify whether two regions are disjoint (i.e. their address ranges do not overlap).
- `vm_find_region` locates the region for a specific address in a vm map object. If the address does not exist in any of the regions in the map, the function returns NULL.
- `vm_find_region_before` returns the region which is located before the given address.

- `vm_find_region_after` returns the region which is located after the given address.
- `vm_map` is the main memory allocation function for the kernel, it allocates virtual and physical pages (number computed from *len* which is in bytes) using an optional address. The user may supply flags to the function that indicate the access privileges of the allocated pages (i.e. read, write, execute) and other properties such as whether the pages are to be device memory, wired, contiguous or shared.
- `vm_map_align` same as `vm_map` but the memory will be aligned at the user specified alignment.
- `vm_unmap` unmaps and frees the virtual memory segment associated with the given address and map.
- `vm_map_physical` maps in a new virtual memory region to a given physical address, this is predominantly used for mapping in memory mapped device driver registers.
- `vm_probe_physical` returns the physical address for a given virtual address.
- `vm_map_exists` is a predicate for checking whether a virtual address is mapped or not.