Operating Systems 2016 Assignment 3: Virtual Memory

Deadline: Friday, April 29 before 23:59 hours.

1 Introduction

The Virtual Memory subsystem is a critical subsystem of modern operating system kernels. 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 operating system kernel must keep track of the virtual memory address space of each process and is responsible for setting up "page tables" that will be used by the Memory Management Unit (MMU) of the CPU to perform the address translations.

The operating system kernel that we use in this lab has been written for an ARM CPU. On this CPU, hierarchical page tables are implemented that comprise two levels, see Figure 1. The `TTBR` register points to the level 1 page table. This page table can have a size up to 16 KB, containing 4096 entries. In our case each of these entries (also called a page table descriptor (PTD)) may point to a level 2 page table[2]. The level 2 page table contains 256 entries and is 1 KB in size. Each entry in the level 2 page table points to a single page of 4096 and is called a page table entry (PTE). We leave it as an exercise to the reader to verify that a full 4 GB address space can be described by this page table setup.

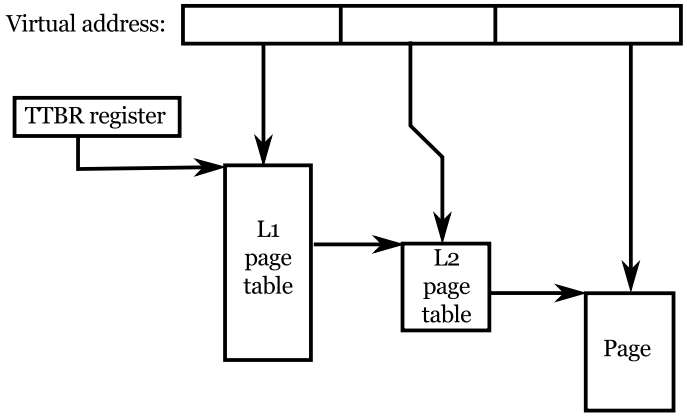

Figure 1: A simplified overview of two-level page tables as used on ARM architectures.

In the kernel the address range `0x80000000 - 0xffffffff` (comprising 2 GB) has been reserved for the kernel. The range `0x0 - 0x7fffffff`, also 2 GB in size, is reserved for user space pages. So, a user-space process is given an address space of 2 GB. In fact, the page tables are split in two and two `TTBR` registers are used: `TTBR0` points to the level 1 page table for user-space processes (so the bottom range) and `TTBR1` to the level 1 page table for kernel-space. On context switch, only `TTBR0` is changed and as a consequence the kernel-space pages are always mapped in.

For reasons that will soon become apparent, currently more than 2 MB of memory must be allocated for each process in the system. In this assignment, it will be your task to improve this by significantly decreasing the amount of memory that must be allocated for each process. The main reason why this amount of memory is currently allocated is because all L2 page tables are pre-allocated for each process. You will improve this by allocating L2 page tables on demand. For each process also a stack is allocated, consisting of 32 pages. As a further improvement, you will modify the implementation to start with a stack of a single page and allocate additional stack pages on demand.

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

[2] The CPU also has support for large pages and sections, but we will not consider these in this assignment.

2 Requirements

We expect the following to be achieved:

- Implement on-demand allocation of L2 page tables for user-space processes:
  - Initially only the L1 table is allocated, all L2 page tables are allocated on demand.
  - L2 tables are released (and invalidated in the L1 table) when no longer necessary. Detection whether L2 tables are still in use is done in an efficient manner.
  - All bytes of an allocated 4096 byte physical page are used to allocate L2 page tables.
  - It is ascertained all L2 tables belonging to a process' page table are released when a process is unmapped.
- Implement lazy allocation of stack pages for user-space processes:
  - Initially, only a single page is allocated as stack area.
  - Page faults for user-space processes are checked to see whether an memory accesses was attempted within the stack area. If so, the allocated stack area must be extended.
  - The stack size remains maximized at 32 pages. Processes attempting to use stack space beyond 32 pages must be terminated as they are now.
- Your submission should include a concise report (not more than two pages A4) which details the data structures that have been designed and reports the number of processes that could be launched at a time using `proclaunch` before and after the modifications. In other words: were the modifications that have been made successful?

3 Submission and Grading

You may work in teams of at most 2 persons. Make sure that all files that you have modified contain your names and student IDs. Please send us a "unified diff" that contains all modifications compared to the starting point. Have an unmodified copy of the starting point extracted as `assignment3.orig` and run:

```
diff -upr assignment3.orig/ assignment3/ > assignment3-sYYYYYYY-sXXXXXXX.diff
```

Check whether the *diff*-file contains *all* of the changes you have made and want to submit (open the *diff*-file in an editor). Submit the *diff*-file together with your PDF report by e-mail to *os2016 (at) handin (dot) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2016 Assignment 3". Include your names and student IDs in the e-mail.

Deadline: We expect your submissions *before* Friday, April 29 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 (1 / 10), Report (1.5 / 10), L2 page table allocation (4.5 / 10), Lazy stack allocation (3 / 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 consists of two parts: (1) implement on-demand allocation of level 2 (L2) page tables for user-space processes, (2) implement lazy allocation of user-space stack pages. In the following two subsections we briefly detail what must be done and where modifications should be made.

5.1 On-demand Allocation of L2 Page Tables

As has been mentioned in the introduction, the size of the user-space address space is 2 GB. So, only half of the page tables are necessary to map this space: 8 KB (out of 16 KB) for the L1 table and 2048 L2 tables, a total size of 8 KB + 2 MB. To test your understanding: verify this and also verify 2 GB of memory can be mapped using this page table.

Operating system code that sets up and modifies page tables is CPU-specific code. The code that manages ARM page tables can be found in `kernel/src/arch/arm/mmu.c`. A number of important functions can be found in this file:

- `hw_alloc_page_table` – This function is called upon creation of a new process to allocate the page tables.
- `hw_map` – Adds an address mapping (virtual, physical address pair) to the page tables.
- `get_l1_table` – Determines the pointer to the L1 page table for a given memory map and virtual address.
- `get_l2_table` – Determines the pointer to the L2 page table for a given memory map and virtual address.
- `hw_unmap` – Removes an address mapping from the page tables.

The memory map is a structure of the type `vm_map_t`, which represents a virtual memory address range (or address space). A `vm_map_t` is associated with each process. 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), so one or more consecutive physical pages. The platform independent code to manage `vm_map_t` types can be found in `kernel/src/vm.c`.

Before you start to implement the necessary changes we very strongly recommended you to first study and understand the current implementation! Start in `hw_alloc_page_table` where the page tables are allocated. Then study the `get_l2_table` function where the L2 page tables are accessed. In particular, understand how the `pagetable_pa` and `pagetable_va` pointers are computed and why this is correct given the initial allocation in `hw_alloc_page_table`. Finally, study how the `get_l2_table` function is used in `hw_map` to create page table mappings.

The assignment is to remove the pre-allocation of all L2 page tables and to allocate these on demand instead. Note: this only has to be done for user-space processes! All code dealing with page tables for kernel-space can be left as it is. To start the implementation: modify the function `hw_alloc_proc_table` to only allocate the L1 table and modify `get_l2_table` to allocate L2 tables on demand. Pages can be allocated using the `vm_map` function, see also the Appendix, and keep in mind that `vm_map` gives you pointers to *virtual* addresses. Page table entries must use *physical* addresses!

Remember that the L2 table size is 1024 bytes, so one page fits 4 L2 tables. You will need to write a number of supporting functions and you might want to introduce a "L2 table descriptor" structure which can be used to point to L2 tables that are available (free) and that are in use. You probably want to maintain a list of L2 tables that are in use in the `vm_map_t` data structure, next to the page table pointers that are already there (see `kernel/include/vm.h`). We do not specify what this data structure should look like: this is entirely up to you. You are free to use the linked list type that is available, see the Appendix for an overview of the macros. But this is not required at all.

You do not need to implement any special handling for out-of-memory situations. If the system is out of memory, simply assert or call `panic`.

In `hw_unmap` you need to free L2 tables that are no longer used. If you maintain a list of available L2 tables, you can put the L2 table that is no longer in use on this list. Do not forget to invalidate the correct entry in the L1 table (simply by setting the value to zero). Design a method to efficiently determine whether an L2 table is in use. You can for example bookkeep some data in a "L2 table descriptor".

5.2 Lazy Allocation of Stack Pages

For the second part of the assignment, we will improve the mechanism by which the stack of a process is allocated. Currently, the entire stack consisting of 32 pages is allocated all at once. This is done in the function `hw_proc_create_user_stack` in `kernel/src/arch/arm/process.c`. `STACK_SIZE` is defined to be 32 times the page size at the top of this source code file. This is quite a waste, as many processes will only use a small fraction of this stack space.

The idea is to only allocate a single page for the stack to start with and to allocate additional pages when this is necessary. When is additional stack necessary? This is the case when memory outside the currently allocated stack, but within the valid stack range, is accessed. What happens when memory outside the currently allocated stack is accessed? This area is not mapped in the process' address space, so a *page fault* will occur. On ARM systems, the page fault handler is called the "data abort handler". You can find this handler as the function `mmu_data_abort` in the file `kernel/arch/arm/mmu.c`. Note that this function is called for all page faults, so for both kernel-space and user-space code.

Your task is to write a function that will extend the allocated stack area. Put this function in `kernel/src/arch/arm/process.c` and call this function from the data abort handler. Determine whether it is valid to continue to grow the stack, we will continue to enforce a strict limit of 32 pages (if a process needs more stack space, run the usual data abort handler code to terminate the process). If you need to add data members to the process structure, do so in `cpu_helper_t` which can be found in `kernel/include/arch/arm/arch-process.h` and can be accessed using `p->cpu_helper.` with `p` of type `proc_t*`.

6 Test programs

We have provided a number of test programs to help test your modifications:

- `mmaptest` - performs a number of memory allocations of varying sizes. Should cause various L2 tables to be allocated.
- `proclaunch <n>` – takes an integer argument n and launches this amount of processes. So, it causes n processes to be allocated and be active at the same time. Given the unmodified code approximately 55 processes can be launched before running out of memory. To test your implementation, can you try significantly larger numbers: 200, 300, etc.
- `fibo <n>` – takes an integer argument n and computes the n-th Fibonacci number using recursive function calls. $n = 20$ and higher are guaranteed to need more than a single page of stack space.

7 Report

Together with the code you will hand in, we expect you to hand in a concise report, not more than two pages A4. Describe the data structure you have designed and implemented to point to L2 page tables, to maintain sequences of L2 page tables and to determine whether a L2 table is still in use or can be released.

Furthermore, we would like you to describe how many processes could be started prior to modification and after implementing the first and second parts. Why (or why not) does this number correspond with what you would expect? Does implementation of the lazy stack allocation give additional benefits? You could also perform a quick test with only the second modification enabled and the first part disabled.

A Linked List API

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` structures, 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`.

B VM Memory Allocation Functions

You will notice that the following functions are used to perform memory allocations in the source code you need to modify:

- `vm_get_kernel_map` returns a `vm_map_t` 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_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.