Operating Systems 2023
Assignment 2: Memory Management Techniques

Deadline: Friday, April 21, 2023, 18:00

1 Introduction

Many modern computer systems support "virtual memory", which allows virtual address spaces to be created that are mapped onto the physical memory. Implementation of virtual memory is typically an interplay between the hardware platform and OS implementation. The hardware provides a Memory Management Unit (MMU) that has the ability to perform translations from virtual addresses, generated by instructions, to physical addresses. The information that is necessary to perform this translation is stored in a data structure called a "page table". The most commonly used page table structure is the hierarchical, or multi-level, page table.

As the virtual address spaces are set up and managed by the operating system, the operating system is responsible for filling the page tables with correct information. The format and layout of a page table is architecture-specific and is dictated by the MMU. Operating systems need specific support for the MMUs of different computer architectures, one could say different MMU drivers are required.

The operating system also has to respond to failed translations. When the MMU is unable to perform a translation, it raises an exception known as a page fault. In response to a page fault, the operating system must determine whether the (virtual) fault address is valid at all and if so add the missing mapping to the page table. Such a mapping must be backed by a physical page. The OS must maintain information on which physical pages in the system are in use and which are available. This is the responsibility of the physical memory manager. An available page can be chosen (allocated) to serve as the backing for this mapping.

Because repeatedly visiting hierarchical page tables is quite an expensive endeavor, MMUs often implement a Translation Lookaside Buffer (TLB). This is a simple associative cache that stores successfully completed translations. Its capacity is typically limited: for example 32 entries. When a new memory access comes in, the MMU should check the TLB first. On a TLB miss, the MMU moves on to perform the actual page table walk.

In this assignment, we will study the above described memory management techniques using a simple framework which simulates components of an operating system kernel and that is able to read memory traces (consisting of virtual addresses). We will study page tables for the 64-bit ARM architecture (AArch64, 16 KiB granule) from both the MMU and OS kernel perspective. This requires the implementation of an MMU subclass that performs address translation according to the ARM specification. When this translation fails, the framework will invoke the page fault handler of the OS kernel. The OS kernel must now allocate a physical page and request the MMU driver to add a mapping to the page table. Also this MMU driver needs to be developed, which is capable of initializing and modifying AArch64 page tables.

Within this assignment, we will also rewrite the physical page manager to use a different algorithm and implement the tracking of physical pages that have been allocated to processes. Upon process termination, the corresponding pages need to be correctly released. Finally, we will extend the MMU class with TLB functionalities. The implementation of the TLB should be generic with regard to the number of entries, such that this number can be easily changed to carry out experiments with different configurations (a command line option to configure this has already been added for you). As replacement algorithm we will consider Least Recently Used (LRU). Moreover, by implementing support for storing address space identifiers (ASID) with each TLB entry, experiments can be carried out to evaluate the effectiveness of ASID for multi-process configurations.

As you might have already noticed, this assignment builds upon a lot of terminology and concepts. If any of the above mentioned concepts is unclear, it is *paramount* that you study the material in the lecture slides and textbook *first*, before starting work on this assignment.

2 Requirements

This assignment concerns the implementation of a number of memory management techniques and a validation that the implementation works. These techniques must be implemented in the provided

framework. Do *not* modify the public API of this framework, as we will be using our own set of unit tests (written against this API) to assess your submission. The following tasks need to be completed:

- A 4-level page table must be implemented according to the AArch64 architecture, 16 KiB granule. Below more details on this can be found. This "page table implementation" consists of:
  - Implementation of the hardware MMU part: address translation.
  - An MMU driver (the OS part): page table initialization, ability to add virtual to physical mappings to the page table, release page table memory when requested, reading referenced and dirty bits from page table entries. Essentially, this requires implementation of the `MMUDriver` interface.
  - Unit tests, see the `tests/` subdirectory.
- The MMU class is to be extended with TLB functionality:
  - The implementation should be generic such that the number of entries stored in the TLB can be easily changed to perform the different experiments. Already present in the framework are a TLB stub class and a command line option to set the number of entries. You must implement the methods of this TLB class, you are *not* allowed to change the method signatures. You are of course allowed to add additional methods and member fields. You can choose yourself in which file to implement the TLB methods, this can also be a new file. Further, do not forget to initialize the `mmu` field in the initializer list of the `TLB` constructor and to add a `TLB` member to `class MMU` which must also be initialized correctly from the MMU constructor.
  - Implement LRU as replacement algorithm.
  - Implement the `getTLBStatistics` method in the `MMU` class, such that statistics are correctly reported upon program termination.
  - Implement support for storing address space identifiers (ASID) for each TLB entry. This will also require an extension to the `MMU` class to allow the `OSKernel` to set the current ASID, which can subsequently be retrieved by the TLB.
  - It is fine to make use of suitable STL data structures in the implementation, but you are restricted to the C++ standard library and basic Boost library features.
  - Note that only the MMU class needs to be modified to implement the TLB, such that all page table implementations can make use of this.
  - Write unit tests for your TLB implementation.
  - The `OSKernel` class must flush the TLB on context switches if ASID is not enabled.
- The physical memory manager (`PhysMemManager`) must be refactored to maintain a list of free memory areas (known as a hole list). Memory allocation must be performed using the first-fit algorithm. Note that when memory is released, you need to find out if an existing hole needs to be enlarged or merged with another hole. In order words, the list of holes must contain as less holes as possible at any point in time.

  Although the framework will only allocate single pages, your implementation **must** work for allocation and release requests of **arbitrary numbers** of pages. Make sure to write unit tests (see the `tests` subdirectory of the framework) to test this.
- The `OSKernel` class must be modified to track all physical pages that have been allocated to processes. These are only these pages that serve as backing for virtual addresses, so the pages containing page tables are excluded. Track the physical pages using the `PhysPage` structure. All pages allocated to a process must be released in the `OSKernel::terminateProcess` method.
- Compose a brief report in `PDF` format that reports on the following:
  - Effectiveness of the AArch64 page table implementation compared to the "simple" page table. For single processes *and* combinations of processes, investigate how much memory is saved using an hierarchical page table, *and* investigate the influence on the number of page faults.
  - Effectiveness of the TLB. Consider different TLB configurations (varying amounts of entries) for at least four different configurations of processes (single and multi process).
  - Effectiveness of TLB ASID. Evaluate the TLB performance without and with ASID enabled for at least three sets of multiple processes. (Note that evaluating ASID for single-process configurations is meaningless).

2.1 Unit tests

This assignment requires you to write unit tests, which can be added to the `tests/` subdirectory. As unit test framework Boost Test is used. An example unit test has been provided. Verifying correct operation and debugging using the memory traces only is difficult. Therefore you are strongly recommended to write unit tests. We expect that unit tests are written and handed in for both the features of the TLB and hole list management of the physical memory manager.

3 Submission

You may work in teams of at most 2 persons. Your submission should consist of the source code of the framework with your modifications and the report in `PDF` format. *Make sure all files contain your names and student IDs.* Remove any object files, binaries and in particular memory traces from your source code directory, to keep the tar files as small as possible. Give your files the following names:

```
sXXXXXXX-sYYYYYYY-lab2.tar.gz
sXXXXXXX-sYYYYYYY-report.pdf
```

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs. Check the size of the zipped tarball, if it is larger than 1 MiB you have most probably included binaries or memory traces, please remove these before submitting. Submit the files through the BrightSpace submission site. Also note your names and student IDs in the text box in the submission website. *Please, ensure only one team member submits the assignment, such that there is a single submission per team!*

Deadline: Friday, April 21, 2023, 18:00.

Notes:

- For those who need the weekend to finalize things: BrightSpace submission is open until Sunday, April 23, 23:59. After this time submissions will be considered late. Note that no questions will be answered on the mailing list after Friday 18:00.
- All source code that is submitted will be subjected to automatic plagiarism checks. Cases of plagiarism will be reported to the board of examiners.
- As with all other course work, keep assignment solutions to yourself. Do not post the code on public Git or code snippet repositories where it can be found by other students.
- Test on the university Linux computers before handing in. In the case of disputes, the university Linux installation is used as reference.
- We may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.

4 Assessment

The maximum grade that can be obtained for this assignment is 10. The points are distributed as follows:

- [**1.0 out of 10**] Code layout and quality
- [**2.5 out of 10**] AArch64 page table implementation
- [**2.25 out of 10**] Hole-based physical memory manager; and tracking and releasing physical pages.
- [**2.25 out of 10**] TLB with LRU replacement and ASID support
- [**2.0 out of 10**] Report

For code quality the following is considered: good structure, consistent indentation, presence and quality of Makefile and unit tests, comments where these are required. Error handling and correct memory handling are considered for each of the different components. We will also look at the unit tests that have been written.

5 The framework

This assignment is to be completed by extending a framework that is provided to you through BrightSpace. This is a fully functional framework in which the hardware MMU and OS kernel parts are implemented and separated. The framework "executes" processes by processing a list of memory accesses read from a

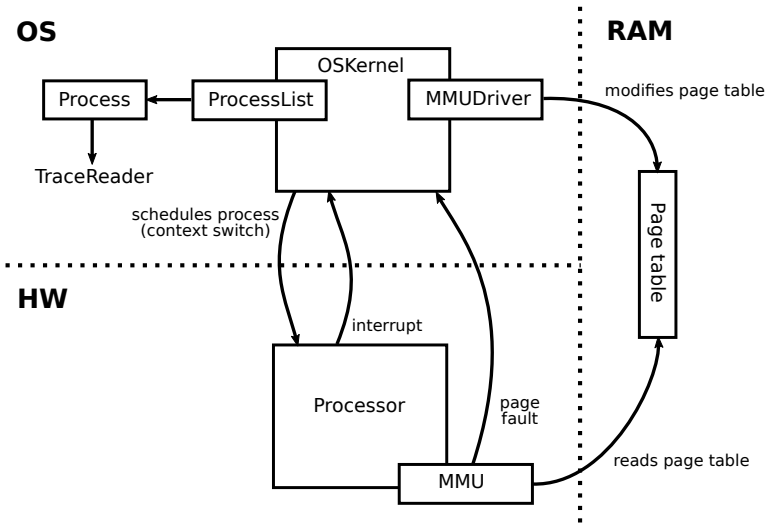


Figure 1: Overview of the different framework components and interactions between these components.

memory trace. It is possible to run multiple processes, in this case multiple memory traces are processed and round-robin scheduling is used to switch between these. We ask you to *not* change the API of the public methods of the `MMU`, `MMUDriver` and `OSKernel` classes, as this will complicate assessing and testing your submission.

Support can be implemented for page tables from different architectures. The page table type to use in the simulation is specified as command line argument, see `main.cc`. An implementation of a very simple, single-level page table is provided as an example. A page size of 64 MiB is used in this case. This page table implementation supports virtual input addresses of 48 bits and physical output addresses of 40 bits (and at most 1 TiB of physical memory as a consequence). In Figure 1 an overview of the framework components and interactions between these components is shown.

One thing to keep an eye on when implementing components in the framework is the difference between page addresses and page numbers. A page address is a full memory address composed of page number and page offset. A page number is just the page number, so with the page offset omitted. `OSKernel` generally works with full addresses. For example, the fault address argument of the `pageFaultHandler` is a full memory address. However, the `MMU` performs translations based on page numbers. This is because the page offset is invariant under address translation. As can be seen in the `getTranslation` method of class `MMU`, the address is transformed to a page number by removing the page offset. The `performTranslation` method operates on page numbers. Also note the difference in variable names: `pAddr` vs. `pPage`.

Before continuing with the actual assignment, make sure to first study and understand the example implementation. Note that some parts of the framework use suboptimal techniques. This is on purpose, such that there are areas that can be improved.

5.1 Limitations

To keep things simple, the framework is not a exact replica of the real-world situation. For your understanding of real-world systems, it is important to be aware of the limitations:

- All page mappings are added to the page table on demand. In a typical situation a large number of pages (in particular these containing the instructions and stack) are already setup before the program starts to run. While running, an application will request memory mappings (`mmap` system call), which are not handled explicitly. Instead, we wait until the application starts to access this area and add mappings on demand.
- Because explicit memory allocation requests are not handled, we also do not handle explicit memory release requests. So, the memory used by a process continues to grow until it is terminated.
- In actual implementations page faults are also used to implement swapping, we do not deal with this in this assignment. Because we do not support swapping, we cannot replace dirty pages.

- We only support a single page size, while actual systems support multiple.
- The memory traces only concern *user-mode* addresses. We do not consider kernel memory.
- For round-robin scheduling a time quantum setting is necessary (as we will later see in the lecture on scheduling). We do not maintain a notion of time in this simulation, but instead approximate the time by counting the number of memory references. The time quantum is thus expressed as the number of memory references instead of a passing of time. Note that the time quantum can be configured using the command line option `-q`, it defaults to 1000.

5.2 Usage

The `pagetables` executable reads the files specified on the command line. It accepts both plain and gzipped files as input. In order to use the program, you need to download a memory trace from the course website. There is no need to unzip the trace as the program is capable of reading these directly:

```
./pagetables -s simple.txt.gz
```

The `-s` option selects the provided "simple" page table to use when processing the memory trace. When working on the implementation of your page table it is strongly recommended to make a small example trace (or excerpt of a larger trace) of at most 25 to 30 memory accesses. This will greatly simplify debugging. You can also have every memory access printed to `stderr` by adding the `-l` command-line argument before the filenames.

If you would like, you can also generate your own memory traces. The traces that are provided on the course website were created using valgrind with the "lackey" tool. To create your own trace, take the following command

```
valgrind --tool=lackey --trace-mem=yes --log-file=mytrace.txt  ls -al /
```

and replace `ls -al /` with a command of your liking.

6 Some notes on AArch64 page tables

In this assignment you will have to implement 4-level page tables according to the AArch64 specification, 16 KiB granule. This means that the pages are 16 KiB in size. A 64-bit address space is incredibly large. Therefore, current implementations only support 48-bit virtual addresses. The upper 16 bits are not used and must be all 0s or all 1s. So, a 48-bit input (virtual) address is translated to a 48-bit output (physical) address.

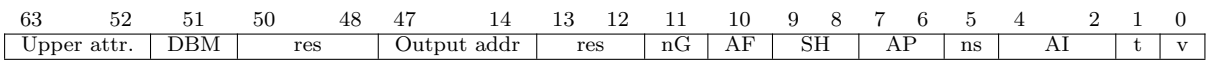
With the 16 KiB granule active, AArch64 uses a 4-level page table to perform this translation. This is depicted in Figure 2. The first level is pointed to by the page table register (`TTBR`, Translation Table Base Register). In this case, the first level is indexed with a single bit and therefore contains two entries. To start with, keep things simple and allocate a full 16 KiB page for the first level (and take the internal fragmentation for granted).

The remaining three levels are indexed with 11 bits and contain 2048 entries each. Page table entries contain 34-bit next-level table or output addresses, which can be turned into a 48-bit physical address using the shift operator and by clearing the upper bits.

Once your program is working, you can optimize the allocation of first-level page tables. According to the ARM documentation, a page table must be aligned to the size of that table. As an optimization, implement the allocation of multiple first-level page tables from a single 16 KiB page. This should reduce the amount of fragmentation and the amount of memory allocated for page tables. Show this in your report, in case you implement this optimization.

For the last page table level, the 34-bit physical page number is combined with the 14-bit page offset to form the final physical address. So, a virtual address consists of 4 page table subscripts and a 14-bit page-offset: $1 + 3 \times 11 + 14 = 48$. Figure 3 shows an overview of the address format. For those who would like to read more background on ARM address translation, the document ARMv8-A Address Translation, Version 1.0, gives a reasonably comprehensive overview in 32 pages.

The AArch64 reference manual describes different page table descriptors for the different levels. For our purposes these can be overlaid to result in the following descriptor that can be used in all levels:



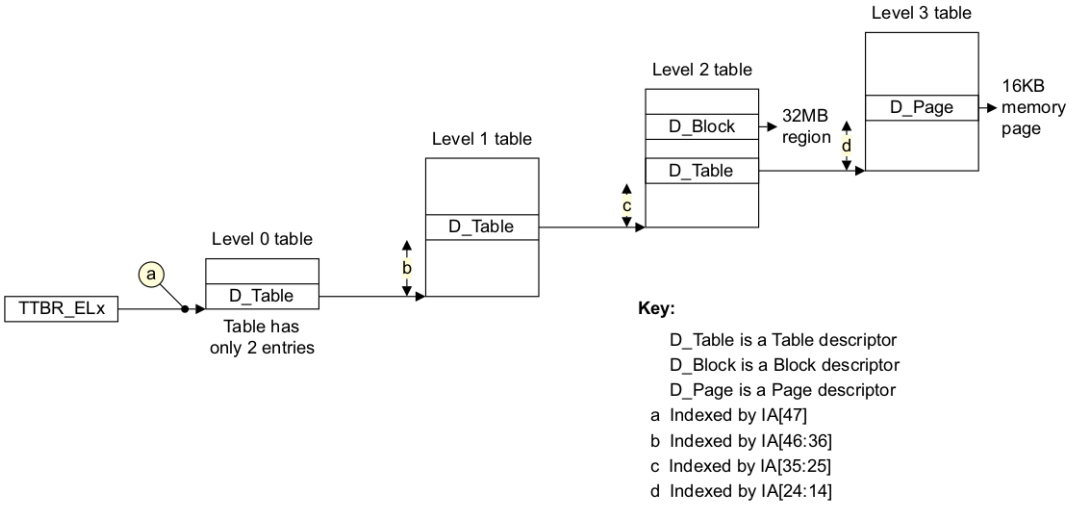


Figure D5-9 General view of VMSAv8-64 stage 1 address translation, 16KB granule

Figure 2: Source: ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile, Figure D5-9.

VA bit [47]	VA bits [46:36]	VA bits [35:25]	VA bits [24:14]	VA bits [13:0]
Level 0 Table Index Each entry contains: Pointer to L1 table (No block entry)	Level 1 Table Index Each entry contains: Pointer to L2 table	Level 2 Table Index Each entry contains: Pointer to L3 table Base address of 32MB block (IPA)	Level 3 Table Index Each entry contains: Base address off 16KB block (IPA)	Block offset and PA [13:0]

Figure 3: Source: ARMv8-A Address Translation. Version 1.0. (ARM 100940_0100_en)

Where the abbreviations have the following meaning:

- **v**: valid, is this entry valid?
- **t**: type, type of entry, for our purposes always set to 1.
- **AI**: AttrIndex
- **ns**: non-secure bit
- **AP**: access permissions
- **SH**: shareability
- **AF**: access flag, was this page accessed?
- **nG**: not global
- **Output addr**: either next-table address or page address.
- **DBM**: dirty bit modifier

Another peculiarity of traditional ARM systems is that the access flag and dirty bit must be managed by software. However, modern ARMv8.1 systems have hardware support to manage these bits. Within this assignment, we simply assume that this hardware support is present and has been enabled, so these two bits are set by the hardware when required.

For those who are interested in all the details, you can refer to the ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile[1] (a 8128-page document) Section D5.3 ("VMSAv8-64 translation table format descriptors"), Figures D5-15, D5-17, Section D5.3.3 subsection "Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors".

[1] https://developer.arm.com/documentation/ddi0487/fb