Operating Systems 2024
Assignment 2: Memory Management Techniques

Deadline: Friday, April 19, 2024, 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. Please refer to the appendices (separate file) for more in-depth information on this 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. More details on this can be found in the assignment appendix. 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.
  - The addition of unit tests to test the different aspects of this page table implementation, see the `tests/` subdirectory.
  - Optimization: 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. In case you implement this optimization, show this in your report.
- 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 several unit tests for your TLB implementation, such that it is thoroughly tested.
  - 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 thoroughly 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 and Assessment

You may work in teams of at most *two* persons. The **deadline** is Friday, April 19, 2024, 18:00. Submit your assignments according to the instructions below.

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

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

For *code quality* the following is considered: good structure, consistent indentation, presence and quality of Makefile, quality of unit tests, comments where these are required. Error handling and correct memory handling are considered for each of the different components.

The following needs to be submitted:

- The source code of the framework with your modifications.
- The report in `PDF` format.

What does *not* have to be submitted: any object files, binaries and in particular memory traces. Please **remove** these from your source code directory before handing in, to keep the tar files as small as possible.

Make sure all files contain your names and student IDs. Give your files the following names:

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

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs.
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!*

Finally, please note the following:

- All submitted source code and reports will be subject to (automatic) **plagiarism checks** using Turnitin, MOSS, or similar. Suspicions of fraud and plagiarism will be reported to the Board of Examiners.
- The use of text or code generated by ChatGPT or other AI tools is **not allowed**. You are required to implement the requested source code **yourself**, and to write the report **yourself**.
- We may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.
- In case you reuse your own work from a previous year, mention this in a README file or within the assignment source code.
- 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. If you use Git, make sure your repository is **private**.
- Test on the university Linux computers before handing in. In the case of disputes, the university Linux installation is used as reference.