Operating Systems 2018, Assignment 2: Page tables & TLB

Deadline: Friday, April 27 before 23:59 hours.

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 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 address is valid at all and if so add the missing mapping to the page table.

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 page tables for a 64-bit architecture from both the MMU and the OS kernel perspective. We will do so using a simple framework that is able to read memory traces (consisting of virtual addresses). Within the MMU class a function is implemented that should try to perform the address translation. When this translation fails, it will call a callback function to which the page fault handler method of the OS kernel is connected. The OS kernel class has as responsibilities the correct initial initialization of the page tables and addition of mappings on demand in response to page faults. Additionally, we will extend the MMU class with TLB functionalities. Finally, we will explore the switching between different page tables that correspond to different processes that are running on a system.

The aim of the assignment is to develop a page table implementation from the hardware and software perspective, extend the MMU class with TLB functionality, implement switching of page tables and perform experiments. The page table that is to be implemented is a 4-level page table modeled after what is used on the x86_64 architecture, see below for more details.

The implementation of the TLB should be generic with regard to the number of entries, so it should be possible to easily change the number of entries that can be stored in the TLB. Add a configuration setting and command line argument to change the number of TLB entries, to help you perform the experiments. As replacement algorithms implement both Least Recently Used (LRU) and random replacement and add a command-line option to choose between these. It is fine to make use of suitable STL data structures in the implementation, but you are restricted to the C++ standard library (no Boost). On program termination the TLB should report a number of statistics, such as: the number of lookups, number of hits and number of entries that had to be removed from the TLB. 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.

Much of the code necessary to simulate multiple processes has already been provided to you: the program accepts multiple filenames to read memory traces from and uses round-robin scheduling to switch between these. 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. Therefore, we 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.

Now, the part that has not been implemented and is left for you is switching the page table pointer when a process is switched. As you can see in `pt-simple.h` the `switchProcess` method will be called when a process is switched. A numeric PID is given as argument. You need to modify this method (and the corresponding method in your own 4-level page table implementation) to switch to the current page table for this address space. Note that this also implies that you need to allocate multiple page tables!

The number of processes that will be run is already known when the class is constructed (count the number of items in `processList`) and will not increase. So, in essence, you will allocate a page table for each process and then switch between these in the `switchProcess` method. Also do not forget to flush the TLB on a process switch!

Finally, we expect you to conduct a number of experiments and write a brief report in PDF format (2-4 A4 pages, use LaTeX). Things to consider are:

- Considering a single process (memory trace): performance of the simple page table versus the 4-level page table (bytes of memory used by the page tables, number of page tables) and the effects of the TLB with different numbers of entries and the different TLB replacement algorithms (compare e.g. hit ratio, number of replaced entries).
- Considering multiple processes: effects of different time quantum settings on TLB performance. Consider different TLB sizes and both TLB replacement algorithms.

2 Requirements

The assignment is to implement the 4-level page table implementation from the hardware and software perspective within the provided framework, extend the MMU class with TLB functionality, implement switching of page tables and write a brief report on the performed experiments. We expect the following:

- A 4-level page table must be implemented that is modeled after the page tables found in the x86_64 architecture. Read on below for more details. This “page table implementation” consists of:
  - Implementation of the MMU part: address translation.
  - Initialization of the page table by the OS part.
  - All memory that is allocated for the page tables by the OS should be properly released in the destructor.
  - An implementation of the page fault handler.
- The MMU class is to be extended with TLB functionality:
  - Two replacement algorithms must be implemented: LRU and random replacement. Add a command-line option to choose between these.
  - The implementation should be generic such that the number of entries stored in the TLB can be easily changed to perform the different experiments. Again, add a command line option to set the number of entries.
  - On program termination statistics should be reported, for example: number of lookups, number of hits, number of entry replacements.
- The `OSKernel` subclass of both page table implementations (the original and yours) must be extended to support switching between page tables as part of a process switch:
  - Multiple page tables should be initialized by the constructor, one for each process (memory trace) that will be run.
  - The `switchProcess` method should be extended to set the page table pointer on the MMU to the page table corresponding to the given process (PID) and to flush the TLB.
- A brief report in `PDF` format (2-4 A4 pages) that reports on the experiments that you have performed for the features that you have implemented. See the introduction for inspiration for experiments. You may also add your own metrics as long as they make sense.

3 Submission and Grading

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.* Put all files to deliver in a separate directory (e.g. `lab2`) and remove any object files and binaries. In particular remove any memory traces from your source code directory to keep the tar files as small as possible. Finally create a gzipped tar file of this directory (make sure the filename contains your student IDs):

```
tar -czvf sXXXXXXX-sYYYYYYY-lab2.tar.gz lab2/
```

Mail your tar files to *os2018 (at) handin (dot) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2018 Assignment 2".

Deadline: We expect your submissions by Friday, April 27, before 23:59. No exceptions; deliveries after the deadline *will not be graded!*. Send e-mail attachments, Google Drive or DropBox links *are not accepted*.

We expect comments in the source code if the code is not immediately obvious, which often means you had to make a deliberate decision or trade-off. Do not add comments on things that are obvious. 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 (1.5 / 10), x86_64 page table (2.0 / 10), TLB (2.0 / 10), Maintaining multiple page tables (2.0 / 10), Report (2.5 / 10).

4 The framework

On the course website the source code of the page table framework is available for download. This is a fully functional framework and we have implemented the MMU and OS kernel parts for a very simple, single-level page table as an example. A page size of 64 MB is used in this case. **Before continuing with the actual assignment, make sure to first study and understand the example implementation.**

The `pagetables` executable reads the files specified on the command line. If none are specified, is reads from standard input. 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 simple.txt.gz
```

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.

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 the containing the instructions and stack) are already setup before the program starts to run.
- In actual implementations page faults are also used to implement swapping, we do not deal with this in this assignment.
- We only support a single page size, while actual systems support multiple.
- The memory traces only concern *user-mode* addresses.
- Usually the page tables have to be stored in physical memory of the system. For this assignment, we simplify this: only for the physical pages fake physical addresses will be allocated using `PhysMemAllocator`. For the page tables we allocate memory within the address space of the simulator application using for example `new` and `delete`. (These memory allocations to have to be specifically aligned however! See also below).
- We do not consider memory mapping requests done by the application (`mmap` system call).

5 Some notes on 64-bit page tables

In this assignment you will have to implement 4-level page tables that are modeled after the x86_64 architecture. 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 are set to the sign extension of the highest bit of the 48-bit address.

The size of the physical address space that is supported is 52 bits. So, a 48-bit virtual address is to be translated to a 52-bit physical address. The x86_64 architecture uses a 4-level page table to do so. A page table at each level is 4 KB in size and contains 512 entries of 8 bytes. As the table contains 512 entries, 9 bits of the virtual address are used to index the table. In the page table entry, among other things, a 40-bit physical page number is found. Using the shift operator this can be turned into a 52-bit physical address.

For the last page table level, the 40-bit physical page number is combined with the 12-bit page offset to form the final physical address. So, a virtual address consists of 4 page table subscripts and a 12-bit page-offset: $4 \times 9 + 12 = 48$. Figure 1 shows a high-level overview of how this page table works. Much more information can for example be found in the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, Chapter 4.5.

Note that only 40 bits are available to store addresses of page tables that are allocated using `new`. This means that the lower 12 bits of the address *must* be zero, or in other words the addresses have to be aligned on 4 KB boundaries. To ensure this you will have to use a special memory allocation function. Because C++ constructions within the C++ standard library to do so are very complicated, we recommend you to use the function `posix_memalign`.

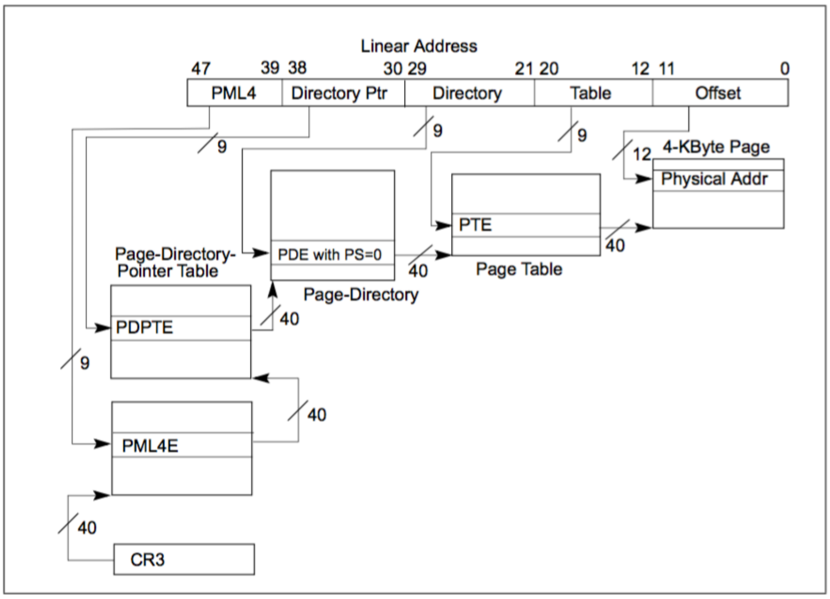

Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Figure 1: Source: Intel Software Developer's Manual, Volume 3A, Part 1