Operating Systems 2016/17
Assignment 10

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Submission Deadline: Monday, January 16th, 2017 – 23:59

A new assignment will be published every week, right after the last one was due. It must be completed before its submission deadline.

The assignments must be filled out online in ILIAS. Handwritten solutions are no longer accepted. You will find the online version for each assignment in your tutorial's directory. **P-Questions** are programming assignments. Download the provided template from ILIAS. Do not fiddle with the compiler flags. Submission instructions can be found on the first assignment.

In this assignment you will get familiar with caches and paging-based address translation.

T-Question 10.1: Caches

a. Why do virtually-indexed, physically-tagged caches not suffer from the ambiguity problem? **1 T-pt**

b. Consider the following code fragment. The algorithm has been optimized so that the loop only has to modify every 16th array element ($k = 16$) instead of every element ($k = 1$). However, although the loop now does only 6% of the original work, the execution of the loop still needs 98% of the original time. What might be the reason? Explain your answer! **1 T-pt**

```
#define ARRAY_LENGTH 64 * 1024 * 1024
int a[ARRAY_LENGTH];

// --- some algorithm here ---

for (int i = 0; i < ARRAY_LENGTH; i += k) { a[i]++; }
```

c. What kinds of cache-misses do exist? What can you do to reduce the number of cache misses of each type? **3 T-pt**

T-Question 10.2: Paging

Consider a system that translates virtual addresses to physical addresses using hierarchical page tables. Every page table comprises 512 entries, with each entry having a size of 8 bytes. The size of both the virtual and the physical address spaces is 512 GiB. The page size is 4096 bytes.

a. How many page tables build a full page table hierarchy in the given system? How many levels does the hierarchy have? **2 T-pt**

b. Into what parts would a MMU for the system split the virtual address during address translation? For each part give its length in bits. **2 T-pt**

c. How many page tables exist when using inverted page tables and single level forward page tables in the operating system, respectively? Briefly explain how you derived your answer. **1 T-pt**

P-Question 10.1: MMU for Paging

Download the template **p1** for this assignment from ILIAS. You may only modify and upload the file `page_table.c`.

Consider a 32 bit x86 system that uses a paging-based 32 bit virtual to 32 bit physical address translation with a two-level page table hierarchy. See the lecture (10. Paging) for more information.

In this question you will implement a simplified MMU for x86 paging in software. The simplified PTEs contain the following fields:

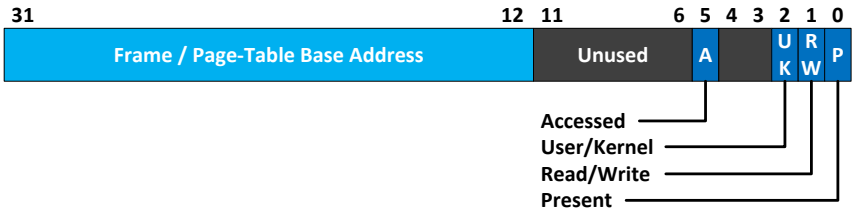
Present Bit Set to 1 if the PTE represents a valid mapping.

Read/Write Bit If set to 1, writes are allowed to the page, otherwise the page is read-only. *If an access is performed from kernel-mode, pages are always writable.*

User/Kernel Bit If set to 0, access from user-mode is *not* allowed to the page. If an access is performed from kernel-mode, both user- and kernel pages may be accessed.

Accessed Bit Indicates whether the MMU has used the PTE for translation.

Frame/Page-Table Base Address Physical frame number (PFN) of the referenced target frame. If the PTE's flag fields are masked, the resulting value is the physical base address of the target frame (i.e., the address of the frame's first byte). For page directory entries (PDEs), the target frame must hold the corresponding second-level page table.



The template already provides bit masks to work with the fields of the PTE as well as functions to retrieve the individual parts of a virtual address (`_getVirtualBase`, `_getPageDirectoryIndex`, ...).

a. Write a function that maps a given virtual page to a given physical frame. The function takes the base address of the page and the target frame as well as the intended access rights as input. Your implementation should fulfill the following requirements: **3 P-pt**

- Uses `cr3` to get the page directory and find the correct page directory entry.
- Allocates a new page table if the PDE is not valid, using `posix_memalign()`. This way the allocated memory for the page table can be aligned to page boundaries, allowing correct addressing via the base address field (i.e., low 12 bits zero). Use `pointerToInt()` to convert the resulting pointer to an integer. Initialize the new page table to all zeros. Do not forget to set the present bit, when updating the PDE.
- Builds the new PTE to establish the desired mapping and updates the page table.
- Returns 0 on success, -1 otherwise (e.g., page table allocation failed).

```
int mapPage(uint32_t virtualBase, uint32_t physicalBase,
    ReadWrite accessMode, PrivilegeLevel privileges);
```

b. Write a function that translates a virtual address to a physical address. The function should fulfill the following requirements: **3 P-pt**

- Uses `cr3` to get the page directory and to find the correct page table and page table entry for the virtual address `*address`.
- Fails if no page table is allocated or no mapping for the virtual page has been established.
- Fails if the desired access is not allowed or the current privilege level of the CPU is not sufficient for the access according to the mapping.
- Performs the address translation and assigns the resulting physical address to `*address`.
- Updates the accessed flag of the PTE.
- Returns 0 on success, -1 otherwise.

```
int translatePageTable(uint32_t *address, ReadWrite accessMode,
    PrivilegeLevel privileges);
```

c. The MMU should be extended with a translation lookaside buffer (TLB), that caches used PTEs, internally using the virtual page base address as key. Complete the `TLBEntry` data structure and write a function that clears a certain entry in the TLB. *Hints:* The `valid` field in the structure indicates if an TLB entry is used and contains a cached PTE. You can access a TLB entry with `_tlb.entries[x]`. **1 P-pt**

```
void invalidateTLBEntry(uint32_t virtualBase);
```

d. Write a function that caches a PTE in the TLB entry specified by `index`. **1 P-pt**

```
void _addToTLBAt(int index, uint32_t virtualBase,
    uint32_t pte);
```

e. Write a function that performs a lookup for the supplied virtual address in the TLB. On success, the function should pass the physical address by updating the value pointed to by `address` and return 0. If a matching translation does not exist, the function should return -1. *Hints:* Remember that the TLB must perform the same access permission checks as `translatePageTable()`. **1 P-pt**

```
int translateTLB(uint32_t *address, ReadWrite accessMode,
    PrivilegeLevel privileges);
```

f. Integrate the TLB into `mapPage()` and `translatePageTable()` by adding new TLB entries or invalidating existing ones as needed. Use the existing functions. **1 P-pt**

Total: 10 T-pt 10 P-pt