Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2023

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 11: Cache Coherence & Page Table Programming

Assigned on: **13th December 2023**
Due by: **19th December 2023, 23:59**

Part I: Pen & Paper Exercises

Question 1

Consider the system in the figure below, where memory write operations are performed in the write-back mode.

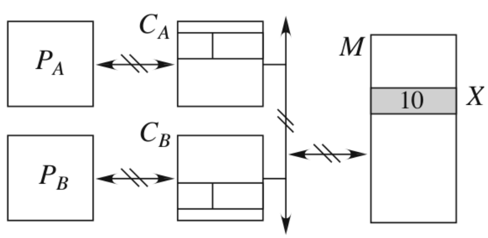

Figure 1: Model of a multi-core architecture with two processors P_A and P_B, their respective caches C_A and C_B, connected to main memory M over a system bus.

At first, all of the cache-lines are marked "invalid". The value 10 is stored in memory at the address X. The two processors execute two sequences of instructions described in the following table.

n	P_A	P_B	Comment
1	mov (X),r1		P_A: r1 := (X)
2		mov (X),r4	P_B: r4 := (X)
3	mov \$0,(X)		P_A: (X) := 0
4		mov (X),r5	P_B: r5 := (X)
5		mov \$20,(X)	P_B: (X) := 20

Table 1: Operations executed by processor P_A and P_B.

a) In the table below, describe the sequence of operations that ensure cache coherence according to the MSI protocol. In each step, assume the instruction has already been executed. For each phase specify the state of the cache lines, the value of X they are holding, and the value of X in main memory.

n	State C_A	Value C_A	State C_B	Value C_B	Value M
1	S	10	I	–	10
2	S	10	S	10	10
3	M	0	I	–	10
4	S	0	S	0	0
5	I	–	M	20	0

b) Now describe the same sequence but assume the architecture uses the MESI protocol.

n	State C_A	Value C_A	State C_B	Value C_B	Value M
1	E	10	I	–	10
2	S	10	S	10	10
3	M	0	I	–	10
4	S	0	S	0	0
5	I	–	M	20	0

Question 2

Consider three cores equipped with a direct mapped, write-back cache of 128 bytes capacity using a cache-line size of 8 bytes. These machines executes "load-store" sequences in the order described in the table below. The size of loads and stores is two bytes. n is the time when instructions are executed. X contains address $0xA0C0$.

n	P_1	P_2	P_3
1	`ld r1,[X]`	...	...
2	`add r1,1`	...	...
3	`st r1,[X]`	...	...
4	...	`ld r2,[X+2]`	...
5	...	`and r2,0FH`	...
6	...	`st r2,[X+2]`	...
7	...	...	`ld r3,[X+6]`
8	...	...	`sub r3,r1,r5`
9	...	...	`st r3,[X+6]`

C = 1100

1000 cache index

⇒ set 8

16 sets, 4 bits, 3 bits block

a) Which cache-line will be used by the three processors? all map to the same cache block

b) Describe MSI protocol transition for the caches of each processor in every step, after the instruction has been executed. Assume all cache-lines are marked as invalid in the beginning.

n	State C_1	State C_2	State C_3
1	S	I	I
2	S	I	I
3	M	I	I
4	S	S	I
5	S	S	I
6	I	M	I
7	I	S	S
8	I	S	S
9	I	I	M

Hand In Instructions for Part I

This part is a paper exercise. If you want your solution to be revised, scan it and either push it to your git repository or send it directly per email, but either way be sure to ping your assistant via Moodle/email.

Part II: Programming

In this assignment you will implement basic virtual memory functionality in C based on the structure of a x86_64 page-table. x86_64 has a four-level page table and these are referred to as the page map level 4, page directory pointer table, page directory and page table respectively.

The following figure shows the bit layout for the different levels of paging structures. (In our case we consider `MAXPHYADDR`, the width of the physical addresses, to be 48 bits.)

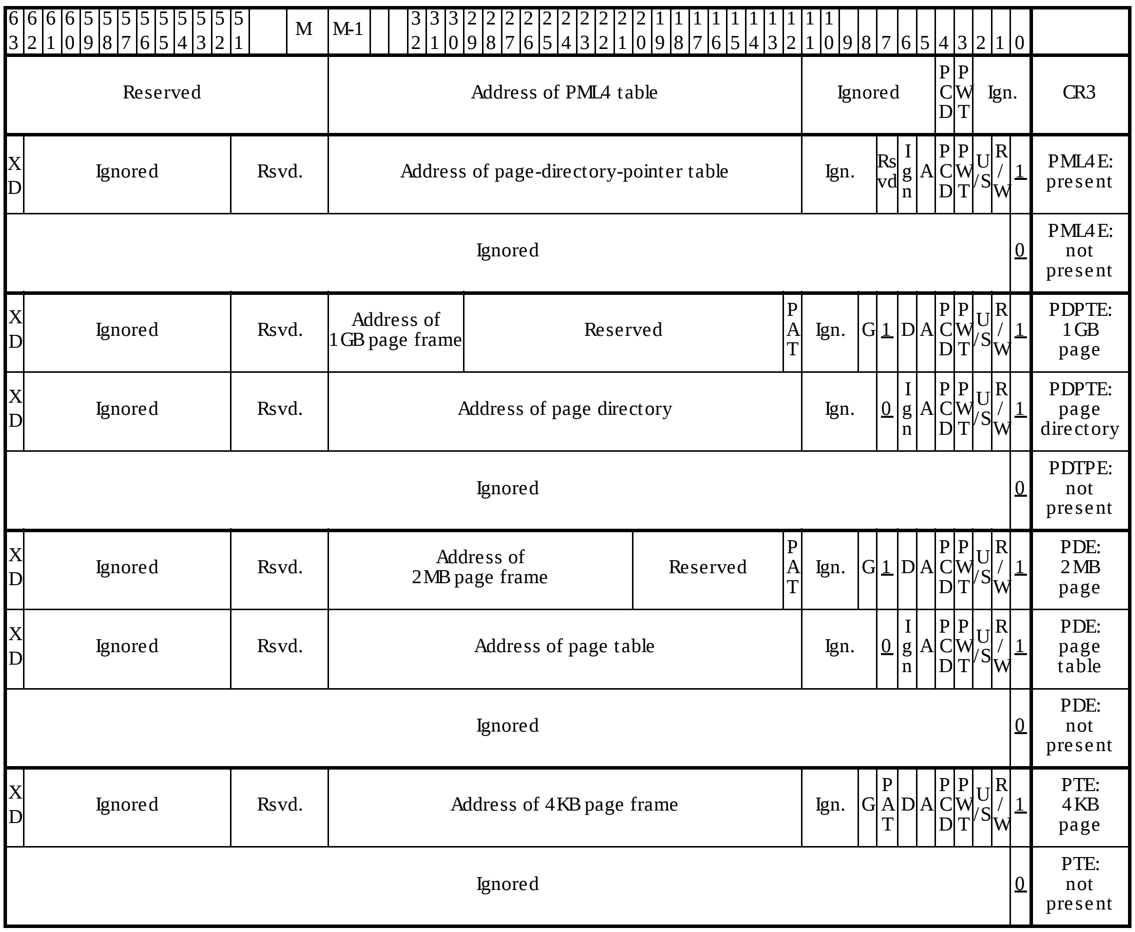

Figure 2: Formats of Paging-Structure Entries[1]

The architecture manuals contain a thorough description of the flags used in figure above and the most pertinent fields are summarized below:

Bit Position	Contents
(M-1):12	Physical address of the page table or page referenced by this entry
7 (PS)	Page Size; must be one if this refers to a large/huge page, otherwise this entry references a page table or regular page
1 (R/W)	Read/Write; if 0, writes may not be allowed
0 (P)	Present; marks whether the entry is present or not

[1]This figure is taken from Figure 4-11, Page 4-28 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1

The structure of a 48-bit virtual address is:

VPN1	VPN2	VPN3	VPN4	VPO	
47	38	29	20	11	0

VPN 1 – 4 are the virtual page numbers used as lookup indices for the four levels of page table, whereas VPO is the virtual page offset which addresses the byte within the page.

Task 1

We prepared a skeleton that can be used as a basis for the implementation. You will find it on the course website as a tar archive. Download and extract that file.

As a first step, read through understand the type definitions which can be found in <paging.h>:

Type	Base type	Description
vaddr_t	uint64_t	Virtual address
paddr_t	uint64_t	Physical address
pml4e_t	union	Entry in the page map level 4
ppde_t	union	Entry in the page directory pointer table (level 3)
pde_t	union	Entry in the page directory (level 2)
pte_t	union	Entry in the page table (level 1)

These types define the bit layouts of page directory and page table entries. The terminology we use here is that a "page directory entry" is referring to another page table or page directory while a page table entry is referring to a mapped region of physical memory.

Note that we are also using C bitfields here. Bitfields are structs where we specify the bit widths of different fields using :<bits> after the field name. This way you can access some bits without having to manually shift and mask. Additionally we're wrapping the bitfield and a uint64_t in an union to be able to easily convert between raw values and structured values of page table and page directory entries.

Next, implement functions *parse_virt_addr*, *get_pml4e*, *get_pdpe*, *get_pde*, *get_pte*. The first of these should extract the page table indices from a virtual address; the remainder are utility functions to read a page table entry and walk to the next level of the page table.

Task 2

Have a look at the skeletons for functions *map*, *map_large* and *map_huge*. These functions map physical pages into the virtual address space by inserting entries into the respective levels of the page tables. They also set some attributes in these entries. At this point, you only have to handle the read/write permissions (i.e. if at least one page is writable, the entire path leading to it should also be marked writable).

For regular 4 KiB pages, *map* needs to touch all four levels of the page-table hierarchy, whereas *map_large* (for 2 MiB mappings) and *map_huge* (for 1 GiB mappings) only modify the content of the second- and third-level page tables (page directory and page directory pointer table respectively). Mapping a large or huge page works by pointing the respective page table entry directly to a 2 MiB or 1 GiB chunk of memory. Note that the memory itself needs to be appropriately aligned, as only the uppermost 18/27/36-bits of the physical address can be stored in the different levels of the page table entries. The PS bit in the second and third-level entries is used to distinguish huge, large and regular pages. Implement these three functions.

In case the page-directory or the appropriate page-table does not yet exist, you will have to allocate some space for it. Have a look at and implement the *alloc_table* function. Remember: Both the beginning of the page-table and the page-directory data-structures need to be page-aligned. Do not forget to implement *free_pagetable* to free all allocated memory again.

Task 3

Next, you are going to implement the *unmap* operation. This routine takes a virtual address and removes the respective entries by setting the value to zero (including the present bit). It should work for both, 4 KiB, 2 MiB and 1 GiB pages.

Evaluation

You can use the *./correctness* script in the handout archive to verify your solution. The program will run your page table implementation and compare it to a reference solution. *dropAddresses* is executed on your output first to remove all dynamic addresses from the page table entries. (These are the addresses stored in the page directory for the pages themselves.) Then, the Unix *diff* command is executed to compare the files.

Hints

Run *make* after extracting the archive we provide on the website to compile your page table manipulation program. gcc will link against a static library *libdump.a* required to dump your page table in a format we can parse with the *./correctness* script. Your program will be compiled in 64-bit mode. As always, you should install the *build-essential* package which contains the compiler and build tools if you have your own environment.

Hand In Instructions for Part II

The first part is a pen and paper exercise. If you want your solution to be revised, scan it and either push it to your git repository or send it directly per email, but either way be sure to ping your assistant via Moodle/email.

For part two, you can find the exercise on code expert.