NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Solution 4

Paging, Page Tables & Translation Lookaside Buffers

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman

Handout: 28.02.2023

Introduction

In exercise four, we focus on connecting virtual memory with its physical counterpart through page tables, their "caches" the translation lookaside buffer (TLB), and the MMU.

1 Paging

For all the exercises below, assume the page tables and page directory described in Tables 1-4

Index	20 ...	12
0x034	0xDEAD0	flags
0x300	0xBEEF0	flags
0x3FB	0x2BAD2	flags

Table 1: Page Directory stored at 0xBADCAFE

	20 ...	12
0x000	0x00000	flags
0x06F	0x11111	flags
0x0CC	0x22222	flags

Table 2: Page Table stored at 0xDEAD0000

	20 ...	12
0x000	0x33333	flags
0x001	0x66666	flags
0x10C	0xCCCCC	flags

Table 3: Page Table stored at 0xBEEF0000

	20 ...	12
0x1CE	0x99998	flags
0x2D2	0x33330	flags
0x2DC	0x66660	flags

Table 4: Page Table stored at 0x2BAD2000

1. Can you motivate why we use nested structures instead of storing the mapping in a single table? What are the advantages and disadvantages of a multi-level page table solution? How can we address the issues that arise from the multilevel approach?

   Answer: *If we would try to keep the mapping in one huge table, that would eat up a large share of the memory. We could use a linked-list like structure that keeps all mappings. That however would be very slow, as we would need to linearly search through all entries in the list if we want to translate an address. Since this is a quite common operation, we need to optimize that. For that we choose a solution in between the two extremes, by having multiple levels of tables. We only need to allocate the storage for a lower level table if the parent entry is mapped and thus save lots of space, since many indexes will not be mapped at any point in time. However, we still need to chase pointers, which can be very time consuming (and since the CPU might be waiting on that data, we might actually stall execution until we finished translating that address). Thus, we make use of the translation lookaside buffer (TLB), which somewhat acts as a cache for our translation. Whenever we do a a translation we insert the mapping into the TLB. That also means that if there is no space left in the TLB we need to throw some other mapping out, thus we need a replacement strategy for the TLB.*

2. In the Page Directory we only use 20 bits for storing the address of the respective Page Tables, the rest is used to store flags. How can we still resolve, where the page tables are stored?

   Answer: *The tables are always placed aligned to 4K, as we just zero extend the 20 bits we already got. The same holds true for pages themselves, as the Page Table entries also only store the most significant 20 bits.*

3. Assuming that both, Page Directory and Page Tables contain 1K entries and that the physical

page is of size 4KB. Which physical addresses do the following virtual addresses map to?

- `0x0D06F00D` = ...

  Answer: *Looking at the first ten bits (We need 10 bits to index into a 1K Page Directory, as* $2^{10} = 1K$*) gives us the index for the Page Directory. The first (most significant) ten bits equal to* $0x34$ *(=* $0b0000110100$*). The entry at index* $0x34$ *in the Page Directory points to the Page Table stored at* $0xDEAD0000$*. Looking at the "middle" 10 bits, we see that those equal to* $0x6F$*, which we use as the index for table 2. Looking up that entry, we get the upper 20 bits for the physical address (=* $0x11111$*). To complete the entire address, we now add the 12 page offset bits (=* $0xD$*) from the virtual address to the physical address which gives us the final address of* $0x1111100D$

- `0xFEEDC0DE` = ...

  Answer: *In similar fashion, we translate the virtual address* $0xFFEDC0DE$ *to the physical address* $0x666600DE$

- `0xC00010FF` = ...

  Answer: *In similar fashion, we translate the virtual address* $0xC00010FF$ *to the physical address* $0x666660FF$

4. Assuming that both, Page Directory and Page Tables contain 1K entries and that the physical page is of size 4KB. Which virtual address does the physical address $0x99998765$ translate to?

   Answer: *Following the traces back, starting at Table 4, we see that the middle ten bits need to correspond to* $0x1CE$ *(=the index of the entry). Checking for the Page Directory entry that points to that page table, we find that the 10 most significant bits need to correspond to* $0x3FB$ *(again the index of the entry). Now we can shift the bits to the correct position and add them together, including the offset of* $0x765$*. The result of this calculation gives us the virtual address* $0xFEDCE765$

5. Given the following split up of the virtual address, how many Page Directories and Page Tables are there? How big are the respective tables? How big are the physical pages? What is the max amount of memory we can handle like that? Would another split change the amount of memory that we can manage? Assume we have a 48-bit address, that is divided up as follows:

   - Part 1 (most significant bits): 14 bits
   - Part 2: 13 bits
   - Part 3: 12 bits
   - Part 4 (least significant bits): 9 bits

   Answer: *This split leads to 1 Page Directory with 16K entries. Then we have two levels of Page Tables, the first level contains 8K entries and the second level contains 4K entries. The last part of the address is used as the offset into the page. As this one only holds 9 bits, the physical page size is 512B. The overall memory we could handle with that is 256 TB, independent of the way we split the address, as we can at most address* 2^{48} *bytes with a 48-bit address.*

6. How does the use of page table enable us to share memory between multiple processes? Do the processes know that a page is shared between multiple processes? Do they have to be at the same virtual address?

   Answer: *We can map the same physical page into the page tables of multiple processes, so all processes that should be able to access a certain region can translate their respective virtual addresses to the physical address. While it might be useful to know that a certain page is shared (e.g. if it is used for synchronization or other forms of communication), there is*

nothing different from a page that is mapped for a single process from the perspective of the process. The mapping also does not need to resolve to the same virtual address, which means the OS can handle virtual addresses for each process independent of other processes.

2 Swapping/Page Replacement

1. Let's assume we are working with a small machine that has 6 physical frames. Given the access pattern below, how would LRU and the Belady's MIN replacement strategy perform? How many replacements are necessary and when do they occur? How many page faults will occur?

   Access pattern: 1, 3, 6, 8, 4, 2, 4, 7, 5, 5, 6, 8, 9, 4, 1, 3, 8, 8, 2, 5, 1, 3, 4, 8, 1, 3, 6

   Answer: *We represent a pagefault with a small p and a replacement with a small r (a replacement implies a pagefault, so a r also implicitly counts as a fault). A - indicates that nothing happened there*

   LRU:

   ```
   1, 3, 6, 8, 4, 2, 4, 7, 5, 5, 6, 8, 9, 4, 1, 3, 8, 8, 2, 5, 1, 3, 4, 8, 1, 3, 6

   p, p, p, p, p, p, -, r, r, -, -, -, r, -, r, r, -, -, r, r, -, -, -, -, -, -, r
   ```

   Total Pagefaults: 14 (replacements: 8)

   Belady's MIN:

   ```
   1, 3, 6, 8, 4, 2, 4, 7, 5, 5, 6, 8, 9, 4, 1, 3, 8, 8, 2, 5, 1, 3, 4, 8, 1, 3, 6

   p, p, p, p, p, p, -, r, r, -, -, -, r, -, -, -, -, -, r, -, -, -, -, -, -, -, r
   ```

   Total Pagefaults: 11 (replacements: 5)

2. Assuming that a memory access takes 20 ns and an access to the disk (we are assuming an SSD here) takes $5 * 10^5$ ns, how much faster is Belady than LRU? What if we don't count the warmup phase (the pagefaults that can't be avoided) and only look at the steady execution?

   Answer: *We can just multiply together the amount of page faults or replacements in the previous solution with the amount of time it takes. This gives the following timing for all pagefaults: Belady:* $11 * 5 * 10^5 + 16 * 20 = 5.50032$ *ms, LRU:* $14 * 5 * 10^5 + 13 * 20 = 7.00026$ *ms. If we only consider replacements: Belady:* $5 * 5 * 10^5 + 16 * 20 = 2.50032$ *ms, LRU:* $8 * 5 * 10^5 + 13 * 20 = 4.00026$ *ms*