Challenge 3

Filip Strömbäck, Ahmed Rezine

February 3, 2026

Instructions

You have to do research on the parts you do not understand. When you solve problems, do not hesitate to:

- State your starting point. What do you know from start? Why is it important?
- State your goal. What do you need to know to reach it?
- State your assumptions. What are you taking for granted or assuming?
- Show step by step how you can go from what you have to what you want.

You will get one credit for each presentation you claim prepared. You may (at random) be selected to present any of the solutions you prepared. When you are selected for presentation:

- For each step in your solution, explain what you were thinking. How did you come up with this? What gave you the clues?
- Aim for 10 minutes.
- You should be able to present most of your solution without reading from your notes. You are, of course, free to reference your notes for calculations, figures, and important points.
- You are not required to have a correct solution to get credit, but **you are required to solve all parts of a problem, to make a serious attempt and to "believe" in your solution**.
- If you are not prepared you loose all credit for the seminar in question. As long as you have solved the problem yourself, and understood all steps in the solution, this is very unlikely to happen.

As audience you should think of how difficult it is to clearly present a solution. Be humble and supportive. There are multiple ways to solve most problems, so you are encouraged to ask questions. Compare the presentation to your solution and ask questions like "I did it in a different way, is there a difference?" This fosters discussions that are beneficial to everyone in the seminar.[1]

You are of course welcome to take notes.

[1] Do not hesitate to ask questions, even if you fear the presenter might not be able to answer. As long as they are able to answer questions about *their approach* and the concepts *they have used*, they run no risk of loosing points for not answering questions from the audience correctly.

Problem 1

Assume that a processor has a 32-bit address space. The system memory address bus is also 32 bits wide. When answering the three questions below, aim for a clear figure and for 3–5 sentences per question.

1. Suppse only segmentation is used for translating logical addresses to physical ones. Assume 8 bits are used to identify the segment and the remaining bits are used as an offset. You can assume segment table entries are 8 bytes each. Describe what information should such a segment table entry contain and explain how a logical address is translated into a physical one.
2. Suppose only paging is used. Assume pages are 4KiB and a two-level paging scheme is adopted. You can assume page entries are 4 bytes each. Describe, at each level, a possible content of a page entry and explain how a logical address is translated into a physical one.
3. Explain how segmentation and paging can be combined, using the same propeties as in the questions above: 8 bits for segment identification, two-level paging, 4 KiB pages, etc. (hint, check how Intel IA32 can combine segmentation and paging).

Problem 2

Assume that your laptop has a 64 bit address space and 64 KiB page size. It supports 4 GiB of physical memory and each page table entry takes up 4 bytes.

(a) How many entries will you need in a single-level page table?

(b) Now assume that you are using a memory-constrained system (*e.g.* small embedded systems or handheld devices), which merely contains 16 MiB physical memory.

The system executes applications, which do not support sharing of code/data. If you want to implement virtual memory in such a system, with the same parameters as in question (a) (64 bit address space and 64 KiB page size), which memory management scheme would you like to choose (hierarchical, inverted, hashed, ...)? Explain your answer (aim for 4–10 lines in your answer).

Problem 3

Consider the problem of implementing a paging scheme on a small handheld device with 32 bit logical address space, 1 KiB page size and 2 MiB of physical memory (DRAM). The device targets applications which may heavily share proprietary code/data (the paging scheme should make sharing easy). Changes to such proprietary code/data are not allowed (you will therefore assume 4 bytes per page table entry). Besides, the device contains a sufficiently large NAND flash memory, which facilitates data exchange with the DRAM.

Do you think an implementation of a one level paging scheme is possible in such a system? How many levels are needed? Explain the paging system and give its advantages and disadvantages. Aim for a clear figure and about 5 lines in your answer.

Hint: You can assume that the NAND flash memory acts as a faster disk for the device.

Problem 4

Answer the following questions related to memory allocation.

(a) Show an example (including total size of the memory and a sequence of requests for memory "chunks") where the `best-fit` allocator performs better than the `first-fit` allocator. Here, better means that `best-fit` results in less fragmentation than `first-fit` and hence can satisfy all memory requests of the sequence.

(b) Your friend claims that the `best-fit` allocator can *never* perform worse compared to the `first-fit` allocator. He is wrong. Provide a counter-example, meaning explain a scenario where the `best-fit` allocator leads to more fragmentation compared to the the `first-fit` allocator.

(c) Some systems may use *compaction* to make all free spaces in the memory to be contiguous. Compaction can be carried out periodically, for instance, when no available block of memory is big enough to hold a process. Can you think of any possible reason why compaction is usually a bad idea in practice?

(d) Nevertheless, assume a system that implements compaction and consider the following two variants of the compaction scheme:

1. Compaction is carried out when no available block of memory is big enough to hold a process.
2. Compaction is carried out when a process exits.

Explain at least one advantage and one disadvantage for one of the variant over the other.

Problem 5

Answer the following questions, which deal with page replacement algorithms.

(a) Which replacement policy is more expensive to implement in hardware – FIFO or LRU? Explain your reasoning.

(b) Find a sequence of memory accesses where FIFO replacement policy performs better (fewer page faults) compared to LRU replacement policy?

(c) Find sequence of memory accesses where LRU replacement performs better (fewer page faults) compared to FIFO replacement policy?

(d) Consider a sequence of memory-page accesses $\mathcal{S} \equiv \langle s_1, s_2, \ldots, s_{n-1}, s_n \rangle$. Assume that the memory (DRAM) is not big enough to hold all distinct pages and the initial state of the DRAM (*i.e.* the set of pages contained in the DRAM) is unknown to us. Your friend claims that the worst-case performance of LRU policy, on the sequence $\mathcal{S}$, could be obtained by assuming that the DRAM was initially *empty*. Your friend is correct. Explain in 5–10 lines why that is the case.

(e) This does not hold for a FIFO replacement policy. Give a scenario where an initially empty DRAM would not correspond to the worst-case performance.

Note: The more page fault that occurs, the worse the performance is. The counter-example can be provided by giving a concrete sequence $\mathcal{S}$ (where each s_i in the sequence $\mathcal{S}$ captures a page number) and the initial state of the DRAM.

Problem 6

You are designing the paging unit for a 32-bit CPU. The CPU produces 32-bit logical addresses, limiting user-space processes to 4 GiB memory. However, physical addresses are 48-bit wide, meaning that the system supports up to 256 TiB of physical memory. Page table entries are 32-bits wide (i.e., 4 bytes). Each entry consists of a frame number and 4 additional bits: *valid*, *user/kernel*, *read-only*, *no-execute*.

(a) Describe how you should design the paging unit for the CPU. That is, what page size should be used, and how many levels are suitable? How are the bits in the logical address divided into page number and offset? How is the information in page table entries used to construct a physical address? And finally, how does address translation work in the end. Remember that physical addresses are 48 bits, but each page table entry can only hold $32 - 4$ bits for a page number.

In your explanation, aim for a clear figure and 2–5 written sentences (your oral explanation can of course be longer).

(b) How is it possible to use 256 TiB of memory, even though the CPU is only able to address 4 GiB of memory at a time?

Aim for 1–3 sentences.

Problem 7

Assume a demand paging system with a DRAM and a mechanical disk for permanent storage. The disk has an average access and transfer time of 10 milliseconds. The logical address is translated via a two-level page table. The system also incorporates a TLB to make the address translation faster. Assume that 75 percent of all memory accesses result in a TLB hit and that the memory access time is in the nanosecond order of magnitude (you can assume 100ns).

(a) How many memory requests are made for a load instruction of the form "`load rx, [addr]`"? Answer for both best and worst case.
Note: The instruction `load` requests one byte from address `addr` to be loaded into a register `rx`.

(b) Now consider an application, which triggers 10% page faults on average, over *all possible requests that might access the memory*. Compute the average execution time of the application if it executes one billion `load` instructions. Assume that the system does not have cache memory and that any overhead is negligible, except the overhead caused by the paging system.

Problem 8

Assume that we have a system with 4 KiB page size and it has a total of 256 KiB physical memory. We want to implement a demand paging algorithm with LRU replacement policy. Now, consider the following program fragment:

```
double a[256][256], b[256][256], x[256][256], y[256][256];
...
for (i = 0; i < 256; i++) {
  for (j = 0; j < 256; j++) {
     a[i][j] = 2.0 * x[i][j] + y[i][j];
  }
}
for (i = 0; i < 256; i++) {
  for (j = 0; j < 256; j++) {
     b[i][j] = 3.0 * a[i][j] * y[i][j];
  }
}
```

Note: Differences in the two loops are underlined for clarity.

Assume that each element of the array consumes 8 bytes. Also assume that program arrays are laid out in memory in row-major form, meaning that for the array `a[256][256]`, the elements of the array a appear in memory consecutively in the following order:

$$a[0,0], a[0,1], \ldots, a[0,255], a[1,0], a[1,1], \ldots, a[1,255], a[2,0], \ldots, a[255,255]$$

Answer the following questions:

(a) How many page faults will occur when executing the given program fragment?

(b) Can you slightly modify the program to reduce the number of page faults? Note that your modification must not affect the semantics of the program, meaning that the end result (values computed in array a and array b) must be exactly the same as obtained via the original program fragment.

(c) A thoughtless programmer may inadvertently write the program in a slightly modified version that considerably worsen the performance. Can you, just as in (b), provide an example?