Virtual Memory Lab Assignment

DistriNet, KU Leuven Ghent Technology Campus

April 20, 2024

1 Goals

In this lab, you will implement a form of multi-level paging, as it exists in modern OS kernels. You will have to parse and process an instruction *trace* from a file, which will contain instructions that map/unmap and access memory, and launch/kill processes. To handle these instructions, you will need to build, manage, and walk page tables, as well as visualize the page table state in a GUI.

You work in teams of 2 for this lab, the same groups as the Scheduling lab. Use the provided GitLab repository for collaboration. That GitLab repository is also where we will evaluate your code. You can make this assignment in whichever programming language you prefer.

2 Overview of Simulated Hardware Environment

1. **The RAM**: There is 64 KB of available RAM. Every physical frame is 4 KB large (2^{12}), the **same size as a virtual page**. Hence, there are 16 physical frames.

2. **Virtual Address Space**: The virtual address space is 2^{48} bytes in size, customary for current 64-bit CPUs. This means that every virtual address is in the range $[0, 2^{48} - 1]$, and the page table can contain up to $\frac{2^{48}}{2^{12}} = 2^{36}$ Page Table Entries (PTEs) for every process. The addresses are 64 bits wide, **not** 32 bits, as may have been the example in the lectures.

3. **The Backing Store**: The backing store/swap area is the secondary storage area where paged out memory can reside until it is moved back to RAM. You can consider it infinite in size.

4. **The Page Tables**: You should maintain 1 page table per process. Every Page Table Entry (PTE) contains the following information:

   (a) `accessible` bit: Can memory accesses access this page? Was it previously mapped by the process?

   (b) `present` bit: Is the page resident in physical memory, or is it swapped out?

   (c) Physical Frame Number (PFN): Corresponding physical frame location in RAM. Only holds useful data if `present` is set.

   (d) `accessed` bit: Set to '1' when any memory access accesses the page. Can be cleared at will.

   (e) `modified` bit: Set to '1' when a memory write writes to the page. Can be cleared at will.

   2^{36} PTEs per process is *way* too much to store in a single page table level. You will need a multi-level page table to handle this amount of virtual memory efficiently. We recommend a 4-level page table, similar to most real-world kernels right now, with 512 entries per array of entries, at every level. Entries in higher levels of the page table are simpler, and only contain the following information:

(a) `present` bit: Does this entry point to a lower level of the table or not?

(b) Pointer to array of lower level entries. Only holds useful data if `present` is set. We recommend that you implement this as a pointer to a raw, fixed-size array (not growable!) in your language.

You are allowed to extend page table entries to contain more information, as long as you can justify your extensions during the demo, keeping in mind the additional memory overhead that storing additional data in PTEs incurs. Most realistic CPUs have only a few spare bits left in the PTE format.

3 The Instruction Trace

Your code will process a trace of instructions from a file. We describe the operation of these instructions below:

1. `<pid> START`: starts a new process that will be identified by the number `pid`. You should create a page table for this new process.

2. `<pid> MAP <address> <size>`: allocate new *virtual* memory space at location `address` with size `size` in the page table of the process given by `pid`. The virtual memory range `[address, address+size[` is guaranteed to not overlap with a previously existing mapping. `size` and `address` are guaranteed to be a multiple of the page size (4KB). One or more PTEs should be created in the process' page table for the range of virtual memory mapped by this instruction. You may need to create additional entries in higher levels of the page table on-the-fly to satisfy this request.

3. `<pid> READ <address>`: a read operation of the memory at the given virtual `address`, in the process given by `pid`. Your code should walk the corresponding process' page tables and obtain the physical address that is mapped to this virtual address. If the virtual page is currently not `present` in the page table, allocate a new physical frame for it, evicting other memory as necessary. If the accessed memory was not previously mapped via the `MAP` instruction, the process should terminate (see the `END` instruction) and the GUI should show the faulting address.

   TIP: *Processes that access invalid memory will contain no further instructions in the trace, because they are supposed to have crashed. If you find that a certain process continues trying to execute instructions in the trace after you terminated it, that points to a* ***bug*** *in your code.*

4. `<pid> WRITE <address>`: Mostly similar to a `READ` operation, but also sets the `modified` bit of the PTE corresponding to this virtual address.

5. `<pid> UNMAP <address> <size>`: removes the PTEs associated with the virtual memory range `[address, address+size[` from the process identified by `pid`. The memory range is guaranteed to be currently mapped in the process, e.g., as the result of a previous mapping using `MAP`. `address` and `size` are guaranteed to be multiples of the page size (4KB).

6. `<pid> END`: ends the process identified by `pid`, and makes all of its resident memory available for other processes to use.

   TIP: *All processes either terminate by accessing invalid memory, or through an* ***`END`*** *instruction. If you find that a process is still alive at the end of the instruction stream, or when its* ***`pid`*** *is reused in another* ***`START`*** *instruction, that points to a* ***bug*** *in your code.*

4 Simplifications

1. In a real kernel, more than just application-level data needs to be mapped in the page tables. For one, the page tables themselves need to map their own memory. **You can ignore the mapping of non-application data in the page tables for this lab**.

2. The `READ` and `WRITE` instructions do not read or write any *actual* data in this assignment. As such, **your simulator should not reserve any actual memory for the physical frames or the backing store**. During page ins/outs, no actual data transfer should happen, although it can be helpful to mark the appropriate locations where this *would* happen with comments in your code.

5 Memory Management Operations

1. You are responsible for finding free physical frames to map virtual pages to. This requires you to keep track of which physical frames are already in use, and which ones are free to assign to new virtual memory. Make sure that you are able to defend your physical frame allocation mechanism at the demo. The physical memory is small in this assignment to facilitate visualization, but try to also think about larger (128TB RAM) systems, and how your physical memory allocation would scale to those.

2. Given the limited RAM size, it is clear that processes can use far more virtual memory than there is physical memory available. When more physical memory is required than what is available, we need to evict some physical pages. Refer to the theory slides for a great overview of the possible algorithms, including LRU, NRU, and Second Chance. You can choose where to search for eligible pages. Make sure that you are able to motivate your choices at the demo.

3. When evicting memory, it is possible that it was only ever read from, and never written, i.e., `modified` bit is not set. This means that the backing store will contain the same contents for that frame already, and the kernel can avoid an expensive copy of the frame to the backing store. You should keep track of the total number of page outs, i.e., modified pages that get evicted and copied to the backing store, as well as the number of page ins and page evictions.

4. Your implementation should be able to support an arbitrary number of processes, paging memory as necessary.

Much like real kernel design, this assignment leaves room for different implementation choices in many places. Optimal implementations often do not exist, or are subject to ongoing research. Feel free to draw inspiration from actual, modern kernel design: the layout of the virtual address space in this assignment was specifically modelled after the userspace of an x86-64 Linux OS for this reason. The most important thing is that you are able to motivate your choices at the demo, and think of realistic scenarios in which your implementation is beneficial.

6 Description of the GUI

The GUI can be very simple in this assignment, even a text-based Terminal User Interface (TUI) suffices. It only serves to help you debug your code, and demonstrate how it works during the demo. The GUI should at least contain the following elements:

1. A button to simulate the next instruction, after which the GUI updates.

2. A button to simulate the rest of the instruction trace, after which the GUI should have up-to-date values for the statistics you collect: page ins, page evictions, page outs (= evictions of modified pages).

3. A system "clock", that increments every time an instruction is simulated.

4. A view of the 16 physical frames in RAM, showing the `pid` of the process they currently belong to, and the virtual address in that process that is mapped to them.

5. The instruction we have just simulated, as well as the instruction we are about to simulate. For READ and WRITE operations that you have just simulated, show the virtual and physical addresses that were involved.

6. The number of page ins, page evictions, and page outs.

Do not compromise the memory and run-time efficiency of your memory management code to satisfy the requirements of this GUI.

7 Report and Evaluation

You will have to submit a small report together with your code for this lab. Try to avoid discussing the parts of the implementation that are standard/mandatory, and focus on the distinct choices you made to optimize page fault handling, memory efficiency, physical frame allocation, etc. You can choose whether to write in Dutch or English. Reports in LaTeX are preferred. PDF form is mandatory.

Push your report into the GitLab repo before the deadline. We will evaluate the code and report that is present in the repo on the `main` branch at the time of the deadline. Please include instructions on how to run your code in a simple `README.md`. Some time after the submission deadline, we will invite you to come defend this lab, together with the scheduling assignment, in person. Your total grade will be a combination of code and implementation quality, report quality, and your in-person defense. Specific dates for the submission deadline and defense will be provided via Toledo.

Feel free to ask questions or clarifications throughout this assignment. You can contact us via email, at adriaan.jacobs@kuleuven.be, or come find us in the G107 office.