NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Lab 3

Memory Management & Paging

Handout: 04.03.2025
Deadline: 21.03.2025, 23:59

Introduction

Hello, and welcome to the third lab. In the labs you have the opportunity to practically try out things you learned in the lectures and the theoretical exercises. The labs are compulsory coursework, you need to get **three out of four labs** approved to sit in the exam. The labs are also subject to NTNU's plagiarism rules. More on that in Section 3.

If you haven't done yet, please sign up for one of the lab sessions under https://s.ntnu.no/labsessions25. You need to be logged in with your NTNU account in order to access the file. The lab sessions will help you get started

1 Task Description

This week we are examining virtual memory management in XV6. While some basic virtual memory implementation is already available in the handout code, we will look at what further functionality and speed-up we can achieve using the abstraction of virtual memory. While virtual memory requires the system to do more work for each lookup, it also enables performance-enhancing optimisations. First, we will implement a helper tool, which can come in handy to debug issues that might arise in the subsequent optimisation part of the lab. The helper enables us to manually examine the corresponding physical address to a given process's virtual address.

We will implement an optimisation called a **VFORK**. You already learned about the fork+exec paradigm to create a new process, where we create a copy of the first process and replace the executable by calling `exec`. Copying the entire process, with all its data and code, to throw it away afterwards seems to be an awfully inefficient way of creating a new process. However, this is not how it is implemented in most Unix/Linux-like systems. They use vfork instead of fork, which does not copy user data at all but only copies the kernel data about the process (the data that would need to be allocated anyway). The user data is then only mapped into the virtual address space of the newly created process read-only (and also changed to read-only in the parent process). Now, as long as neither process writes to memory, we are good and do not need to copy any data, thus increasing efficiency. When one of the processes, child or parent, tries to write any location, we need to handle that, as we do not necessarily want the changes to be visible to the counterpart. Since we set the mapping to be read-only for both, trying to write to a read-only page will trap on a pagefault, which means the OS can now fix the problem by now actually copying the accessed page and updating the pagetable mapping to the copied page and set it to be writeable. After that, the trapping process can continue, and the instruction causing the trap will be executed again.

We kept the tools from previous labs in the handout code, so if you want to migrate your scheduler from the last lab to this lab, feel free to do so. Further, the `time` utility is also available, so you can test if the implementation of `vfork` impacts the performance of either the benchmark or some other executable.

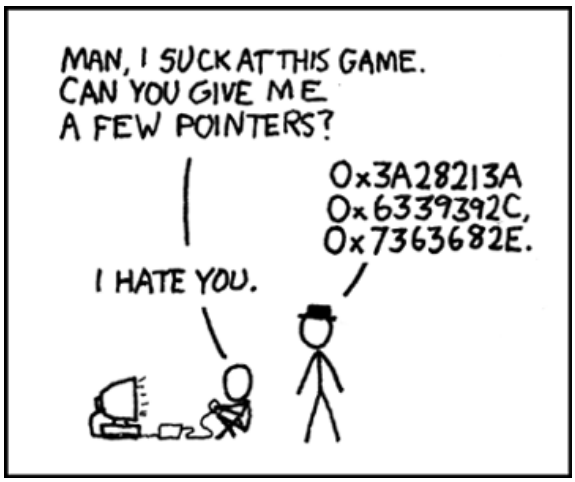
1.1 Task 1: Virtual Address to Physical Address Helper Tool

In order to help with debugging later on, we want to implement a syscall that allows us to translate the virtual address in a given process to the physical address. To test the syscall, we will also add a small userspace program, which we are going to call `vatopa` (Virtual Address TO Physical Address), which takes a virtual address and an optional process id (PID) as arguments and prints out the physical address as a hexadecimal value if such a mapping exists. If either we don't find a process with the given PID or there is no such virtual address for that process, the syscall should return 0. If no PID is supplied to the user-space `vatopa` tool, it should use its own PID for the syscall. Please note in the sample output below that the address changes for subsequent invocations of `vatopa` when we resolve the address for its own PID, as it maps a different page, but when we call `vatopa` with the same address and the same PID, the returned physical address should stay constant (as long as the mapping does not change).

Note: Since we use the underlying syscall also for the testing program for the next task, a syscall stub already exists (the syscall is called `va2pa`*). So you only have to add the functionality to the syscall handler and potentially other functions. If no PID is provided to vatopa, use pid 0 as the argument for the va2pa syscall, as no process has pid 0.*

```
$ vatopa
Usage: vatopa virtual_address [pid]
$ vatopa 0
0x87F44000
$ vatopa 0
0x87F4F000
$ vatopa 0 1
0x87F69000
$ vatopa 0 1
0x87F69000
$ vatopa 3735928559 1
0x0
```

Listing 1: Sample in/output for `vatopa`



"Every computer, at the unreachable memory address 0x-1, stores a secret. I found it, and it is that all humans ar– SEGMENTATION FAULT."
Source: Pointers by Randall Munroe / CC BY-NC 2.5 DEED

1.2 Task 2: Switching from fork to vfork

The current fork implementation of xv6 is very simplistic and does not use the functionality the virtual memory implementation provides. Thus, when we fork, it allocates space for a new process (for the proc struct), and then proceeds to allocate user memory of the same size as the parent's memory to copy over any data from the parent to the child. While this is simple and works for small application footprints, it breaks when we have a very data-intensive workload, processing data that is bigger than half the size of the physically available memory. XV6 currently fails when we fork such a process, as XV6 cannot allocate all the required memory for the child process. However, the child process might not operate on the entire dataset but only on parts of it; thus, we don't want to copy all the data but only the one the child wants to modify.

As we don't know which pages the child (and also the parent) will write to, we need a way to determine which pages the two processes want to write to. Again, we use virtual memory implementation and the processes' pagetables. We store a bunch of flags in the pagetables for the respective processes. We can use those flags to indicate that, currently, they are not to be written (since both point to the same hardware page). A write to any address within that range will fail and trap into the operating system. The OS regains control and can now sort things out by checking if that page is a page that was shared due to a vfork. If it is one of the vfork pages, we can now make a copy of that page and replace the pagetable entry for the writing process with one pointing to the copy and allow writing to that section. We call this behaviour "Copy-on-Write" (COW), as we only make a copy once a process attempts to write to a specific address.

To implement the vfork behaviour we need the following parts to be working:

- The fork syscall: Instead of copying everything eagerly, the fork syscall should only map the user-space memory read-only into both virtual memory spaces, the one of the parent and the one of the child. As a single physical page is now mapped into multiple processes memory, you must ensure that you only deallocate a page once no process has it mapped in their pagetable. Also, make sure that no process can write to the pages that are shared. Files that might be interesting for that part are:
  - `kernel/vm.c`: Implements a lot of helper functions for virtual memory, including `uvmcopy`, which currently eagerly copies memory from one process to another.
  - `kernel/kalloc.c`: Implement memory allocation functionality for the kernel, including `kalloc` and `kfree`, which are the functions to (de-)allocate physical pages.
  - `kernel/proc.c`: Implements the fork functionality.
- The trap handling: A process that tries to write to a page mapped as read-only will trap into the kernel with `scause` 15. The OS regains control and has to handle that trap by making sure we now copy the content of the page and map it into the writing process's memory space. Please ensure that pages no longer mapped in any pagetable are freed and that no page still mapped in some process is freed. For more details on how trapping works, see the RISC-V privileged ISA documentation[1], sections 12.1.* and 12.3, as well as the XV6 book[2]. Files that might be interesting for this part are:
  - `kernel/trap.c`: The trap handler of XV6. Add handlers for whatever traps you think are necessary to be handled and find out how to get the relevant information. Page faults in RISC-V set the `scause` register to the value 0xf (=15) and put the faulting address in the `stval` register.
  - `kernel/vm.c`: Implements a lot of helper functions with respect to virtual memory.
  - `kernel/kalloc.c`: Implement memory allocation functionality for the kernel.
- Other useful definitions or declarations can be found in the following files:

[1] Available on Blackboard under Labs/Useful Resources/RISC-V Privileged ISA Manual
[2] Also available on Blackboard under Labs/Useful Resources/XV6 Book

- `riscv.h`: Contains a few helper definitions to work with physical and virtual addresses and the corresponding pagetable entries. Additionally, there is a bunch of constants that might prove to be useful.

Feel free to modify any methods and functions to work as intended. As you might want to roll back certain changes, using a version control system such as GIT[3] is highly encouraged.

If you find starting the task challenging, see the easier and similar difficult optional tasks. Some might help you break down the problem into more manageable pieces. If you want to dive deeper, please tackle the harder optional tasks.

1.2.1 Optional Tasks

EASIER: As we now start sharing pages between multiple processes, we need a way of tracking how many processes reference this page. A relatively simple approach is to introduce some form of reference counting. How could you implement a reference counter for all pages? How many pages do we need to track at the maximum? When do we deallocate a page? Implement the reference counter and adjust the functions in `kalloc.c` and possibly other files that use the functions from `kalloc.c` to use that reference counter in order to decide when to deallocate a page.

EASIER: Which function is responsible for copying over the data from one process to another? Have a look at the fork function in `kernel/proc.c` first, and figure out which of the functions called there do copying. Modify that function to instead of copying the user memory just to map it to the child and change the entry in the parent's page table such that it now only is read-only. Feel free to add more flags to the page table entry if you require more meta information to be stored with the entries (hint: the RSW bits in the pagetable entry are reserved to be used by the supervisor or in our case the OS. See the RISC-V privileged ISA documentation 4.3.1 for more information on pagetable entries). You find the existing flags and some helpers defined in `kernel/riscv.h`. See other examples in the code for how those flags are used.

SIMILAR: Now that we share pages between multiple processes, we would expect that some processes try to write to pages that are set to be read-only. In that case, the process will trap[4], and the OS trap handler gets to handle the trap. In case the process trapped on a shared page, that is shared because of a vfork, we should copy that page (compare to the unmodified version of the function you updated in the previous optional task) and map the copy into the calling process now as a writeable page. When we return, the trapping instruction will be retried and succeed, as the page is now mapped as writeable. Please ensure that only pages shared because of a vfork are granted write access, as other pages might be shared read-only intentionally (e.g. reading out information from the kernel). A process page faulting on any other condition should be killed and `SEGFAULT` should be printed to stdout. Add your trap handler to the file `kernel/trap.c` in the function usertrap (as a userprocess traps).

HARDER: As we now already have implemented the capability to share pages, add a syscall to map shared pages that the children then can either read from, write to or use to read & write to exchange information with the parent process or other children. Since we copied the mapping, the address will stay the same for all processes. Remember that if a process now traps on a shared page (e.g. it is read-only for that process), we should not give write access to that process but kill the process (the process does not have the right to write to that page).

HARDER: With the pagetables we can not only map memory into the address space, but we also can map devices or files. Add an mmap syscall to map files into memory and handle the traps to read/write (depending on the permissions) from and to the file underneath. You can decide how much of the default unix mmap interface you want to include in your implementation. Try, however, to achieve the read and write to a file.

[3] https://git-scm.com/

[4] Trapping on different conditions is a functionality provided by hardware; the OS just configures the hardware in a way that behaves as expected. To find out more, have a look at the RISC-V (hardware) documentation and XV6 (software) book (footnote 1 & 2 above)

2 Handing In

Before handing in, we advise you to use the command `make test`, which will test your implementation with a small set of test cases. If you want to see the tests in greater detail, you can find the executed commands and the expected output in the file `test-lab-l3`. The test script also tests for some optional tasks, so you don't need to pass all the tests. We marked the test cases that belong to the mandatory tasks with a tag `[MANDATORY]`. Try to make sure that all the mandatory tests pass before handing in.

After you have tested and potentially debugged and fixed your solution, you can run `make prepare-handin` to create an archive of your solution. The archive will be written in the current directory, and the file is called `lab-l3-handin.tar.gz`. Take that file and upload it to Blackboard on the submission page for lab 3. You can submit multiple solutions; we will grade the last submission we received before the deadline.

Please note that if you submitted your solution before the deadline and did not contact us in case of any problems, we will proceed to grade it. Once we have graded it and released the feedback, you cannot resubmit a new solution.

Files to Hand in

- **lab-l3-handin.tar.gz**: The output of the command `make prepare-handin`. Please do not package up your code manually, as this command runs some tests on your side and ensures that we see if it ran on your machine. Additionally, the format it packages it up is expected by our automated tests.

No other files need to be submitted for this lab. If you have any comments, please feel free to use the comment field for submissions in Blackboard. If you have any questions, please use Ed Discussion or come to the lab sessions. We will not answer questions in the comment field to your solution.

Missing Deadlines

We expect you to start working early on the labs, to ensure submitting before the deadline. The deadlines are fixed and there is little we can do to move them. However, if you encounter a problem and notice that you can't make it in time, please contact us as soon as you notice. If you contact us early we have time to come up with a solution. In general, we expect you to contact us before the deadline if you can't make it. There are only few reasons we consider if you contact us after the deadline.

Last lab: Since we need to hand in a list of people that get to sit in the exam, handing in late for the last lab is not possible. We need a bit of time to grade the labs and if you hand in late, we cannot grade your assignment in time before the list must be handed over to administration.

3 Plagiarism

You must submit your **own work**. You must write your **own code** and **not copy** it from anywhere else, including your classmates, internet, and textbooks[5]. Failure to do so might be considered plagiarism. Detailed guidelines on what constitutes plagiarism can be found at: https://innsida.ntnu.no/wiki/-/wiki/English/Cheating+on+exams.

We check all submitted code for similarities to other submissions and online sources. Plagiarism detection tools have been effective in the past at finding similarities. If we suspect that a student copied code, we will involve administration to follow up on the case.

[5] This is not an exhaustive list. Don't copy code from any source