Lab: page tables

In this lab you will explore page tables and modify them to simplify the functions that copy data from user space to kernel space.

> Before you start coding, read Chapter 3 of the [xv6 book](), and related files:
>
> - `kern/memlayout.h`, which captures the layout of memory.
> - `kern/vm.c`, which contains most virtual memory (VM) code.
> - `kernel/kalloc.c`, which contains code for allocating and freeing physical memory.

To start the lab, switch to the pgtbl branch:

```
$ git fetch
$ git checkout pgtbl
$ make clean
```

Print a page table (easy)

To help you learn about RISC-V page tables, and perhaps to aid future debugging, your first task is to write a function that prints the contents of a page table.

> Define a function called `vmprint()`. It should take a `pagetable_t` argument, and print that pagetable in the format described below. Insert `if(p->pid==1) vmprint(p->pagetable)` in exec.c just before the `return argc`, to print the first process's page table. You receive full credit for this assignment if you pass the `pte printout` test of `make grade`.

Now when you start xv6 it should print output like this, describing the page table of the first process at the point when it has just finished `exec()`ing `init`:

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

The first line displays the argument to `vmprint`. After that there is a line for each PTE, including PTEs that refer to page-table pages deeper in the tree. Each PTE line is indented by a number of `" .."` that indicates its depth in the tree. Each PTE line shows the PTE index in its page-table page, the pte bits, and the physical address extracted from the PTE. Don't print PTEs that are not valid. In the above example, the top-level page-table page has mappings for entries 0 and 255. The next level down for entry 0 has only index 0 mapped, and the bottom-level for that index 0 has entries 0, 1, and 2 mapped.

Your code might emit different physical addresses than those shown above. The number of entries and the virtual addresses should be the same.

Some hints:

- You can put `vmprint()` in `kernel/vm.c`.
- Use the macros at the end of the file kernel/riscv.h.
- The function `freewalk` may be inspirational.
- Define the prototype for `vmprint` in kernel/defs.h so that you can call it from exec.c.
- Use `%p` in your printf calls to print out full 64-bit hex PTEs and addresses as shown in the example.

> Explain the output of `vmprint` in terms of Fig 3-4 from the text. What does page 0 contain? What is in page 2? When running in user mode, could the process read/write the memory mapped by page 1?

A kernel page table per process (hard)

Xv6 has a single kernel page table that's used whenever it executes in the kernel. The kernel page table is a direct mapping to physical addresses, so that kernel virtual address *x* maps to physical address *x*. Xv6 also has a separate page table for each process's user address space, containing only mappings for that process's user memory, starting at virtual address zero. Because the kernel page table doesn't contain these mappings, user addresses are not valid in the kernel. Thus, when the kernel needs to use a user pointer passed in a system call (e.g., the buffer pointer passed to `write()`), the kernel must first translate the pointer to a physical address. The goal of this section and the next is to allow the kernel to directly dereference user pointers.

> Your first job is to modify the kernel so that every process uses its own copy of the kernel page table when executing in the kernel. Modify `struct proc` to maintain a kernel page table for each process, and modify the scheduler to switch kernel page tables when switching processes. For this step, each per-process kernel page table should be identical to the existing global kernel page table. You pass this part of the lab if `usertests` runs correctly.

Read the book chapter and code mentioned at the start of this assignment; it will be easier to modify the virtual memory code correctly with an understanding of how it works. Bugs in page table setup can cause traps due to missing mappings, can cause loads and stores to affect unexpected pages of physical memory, and can cause execution of instructions from incorrect pages of memory.

Some hints:

- Add a field to `struct proc` for the process's kernel page table.
- A reasonable way to produce a kernel page table for a new process is to implement a modified version of `kvminit` that makes a new page table instead of modifying `kernel_pagetable`. You'll want to call this function from `allocproc`.
- Make sure that each process's kernel page table has a mapping for that process's kernel stack. In unmodified xv6, all the kernel stacks are set up in `procinit`. You will need to move some or all of this functionality to `allocproc`.
- Modify `scheduler()` to load the process's kernel page table into the core's `satp` register (see `kvminithart` for inspiration). Don't forget to call `sfence_vma()` after calling `w_satp()`.
- `scheduler()` should use `kernel_pagetable` when no process is running.
- Free a process's kernel page table in `freeproc`.
- You'll need a way to free a page table without also freeing the leaf physical memory pages.
- `vmprint` may come in handy to debug page tables.
- It's OK to modify xv6 functions or add new functions; you'll probably need to do this in at least

kernel/vm.c and kernel/proc.c. (But, don't modify kernel/vmcopyin.c, kernel/stats.c, user/usertests.c, and user/stats.c.)

- A missing page table mapping will likely cause the kernel to encounter a page fault. It will print an error that includes sepc=0x00000000xxxxxxxx. You can find out where the fault occurred by searching for xxxxxxxx in kernel/kernel.asm.

Simplify `copyin/copyinstr` (hard)

The kernel's `copyin` function reads memory pointed to by user pointers. It does this by translating them to physical addresses, which the kernel can directly dereference. It performs this translation by walking the process page-table in software. Your job in this part of the lab is to add user mappings to each process's kernel page table (created in the previous section) that allow `copyin` (and the related string function `copyinstr`) to directly dereference user pointers.

> Replace the body of `copyin` in `kernel/vm.c` with a call to `copyin_new` (defined in `kernel/vmcopyin.c`); do the same for `copyinstr` and `copyinstr_new`. Add mappings for user addresses to each process's kernel page table so that `copyin_new` and `copyinstr_new` work. You pass this assignment if `usertests` runs correctly and all the `make grade` tests pass.

This scheme relies on the user virtual address range not overlapping the range of virtual addresses that the kernel uses for its own instructions and data. Xv6 uses virtual addresses that start at zero for user address spaces, and luckily the kernel's memory starts at higher addresses. However, this scheme does limit the maximum size of a user process to be less than the kernel's lowest virtual address. After the kernel has booted, that address is `0xC000000` in xv6, the address of the PLIC registers; see `kvminit()` in `kernel/vm.c`, `kernel/memlayout.h`, and Figure 3-4 in the text. You'll need to modify xv6 to prevent user processes from growing larger than the PLIC address.

Some hints:

- Replace `copyin()` with a call to `copyin_new` first, and make it work, before moving on to `copyinstr`.
- At each point where the kernel changes a process's user mappings, change the process's kernel page table in the same way. Such points include `fork()`, `exec()`, and `sbrk()`.
- Don't forget that to include the first process's user page table in its kernel page table in `userinit`.
- What permissions do the PTEs for user addresses need in a process's kernel page table? (A page with `PTE_U` set cannot be accessed in kernel mode.)
- Don't forget about the above-mentioned PLIC limit.

Linux uses a technique similar to what you have implemented. Until a few years ago many kernels used the same per-process page table in both user and kernel space, with mappings for both user and kernel addresses, to avoid having to switch page tables when switching between user and kernel space. However, that setup allowed side-channel attacks such as Meltdown and Spectre.

> Explain why the third test `srcva + len < srcva` is necessary in `copyin_new()`: give values for `srcva` and `len` for which the first two test fail (i.e., they will not cause to return -1) but for which the third one is true (resulting in returning -1).

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Submit

You will turn in your assignments using the [submission website](). You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type **`make handin`** to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

`make handin` will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let **`make handin`** generate it again (*myapi.key* must not include newline characters).

If you run **`make handin`** and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with **`??`**. You can cause `git` to track a new file that you create using `git add filename`.

If **`make handin`** does not work properly, try fixing the problem with the curl or Git commands. Or you can run **`make tarball`**. This will make a tar file for you, which you can then upload via our [web interface]().

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Use super-pages to reduce the number of PTEs in page tables.
- Extend your solution to support user programs that are as large as possible; that is, eliminate the

restriction that user programs be smaller than PLIC.

- Unmap the first page of a user process so that dereferencing a null pointer will result in a fault. You will have to start the user text segment at, for example, 4096, instead of 0.