Lab: page tables

In this lab you will explore page tables and modify them to speed up certain system calls and to detect which pages have been accessed.

> Before you start coding, read Chapter 3 of the xv6 book, and related files:
>
> - `kernel/memlayout.h`, which captures the layout of memory.
> - `kernel/vm.c`, which contains most virtual memory (VM) code.
> - `kernel/kalloc.c`, which contains code for allocating and freeing physical memory.
>
> It may also help to consult the RISC-V privileged architecture manual.

To start the lab, switch to the pgtbl branch:

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

Speed up system calls (easy)

Some operating systems (e.g., Linux) speed up certain system calls by sharing data in a read-only region between userspace and the kernel. This eliminates the need for kernel crossings when performing these system calls. To help you learn how to insert mappings into a page table, your first task is to implement this optimization for the `getpid()` system call in xv6.

> When each process is created, map one read-only page at USYSCALL (a virtual address defined in `memlayout.h`). At the start of this page, store a `struct usyscall` (also defined in `memlayout.h`), and initialize it to store the PID of the current process. For this lab, `ugetpid()` has been provided on the userspace side and will automatically use the USYSCALL mapping. You will receive full credit for this part of the lab if the `ugetpid` test case passes when running `pgtbltest`.

Some hints:

- You can perform the mapping in `proc_pagetable()` in `kernel/proc.c`.
- Choose permission bits that allow userspace to only read the page.
- You may find that `mappages()` is a useful utility.
- Don't forget to allocate and initialize the page in `allocproc()`.
- Make sure to free the page in `freeproc()`.

> Which other xv6 system call(s) could be made faster using this shared page? Explain how.

Print a page table (easy)

To help you visualize RISC-V page tables, and perhaps to aid future debugging, your second 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 part of the lab 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 0x0000000087f6b000
 ..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
 .. ..0: pte 0x0000000021fd9801 pa 0x0000000087f66000
 .. .. ..0: pte 0x0000000021fda01b pa 0x0000000087f68000
 .. .. ..1: pte 0x0000000021fd9417 pa 0x0000000087f65000
 .. .. ..2: pte 0x0000000021fd9007 pa 0x0000000087f64000
 .. .. ..3: pte 0x0000000021fd8c17 pa 0x0000000087f63000
 ..255: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. ..511: pte 0x0000000021fda401 pa 0x0000000087f69000
 .. .. ..509: pte 0x0000000021fdcc13 pa 0x0000000087f73000
 .. .. ..510: pte 0x0000000021fdd007 pa 0x0000000087f74000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

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? What does the third to last page contain?

Detect which pages have been accessed (hard)

Some garbage collectors (a form of automatic memory management) can benefit from information about which pages have been accessed (read or write). In this part of the lab, you will add a new feature to xv6

that detects and reports this information to userspace by inspecting the access bits in the RISC-V page table. The RISC-V hardware page walker marks these bits in the PTE whenever it resolves a TLB miss.

> Your job is to implement `pgaccess()`, a system call that reports which pages have been accessed. The system call takes three arguments. First, it takes the starting virtual address of the first user page to check. Second, it takes the number of pages to check. Finally, it takes a user address to a buffer to store the results into a bitmask (a datastructure that uses one bit per page and where the first page corresponds to the least significant bit). You will receive full credit for this part of the lab if the `pgaccess` test case passes when running `pgtbltest`.

Some hints:

- Read `pgaccess_test()` in `user/pgtlbtest.c` to see how `pgaccess` is used.
- Start by implementing `sys_pgaccess()` in `kernel/sysproc.c`.
- You'll need to parse arguments using `argaddr()` and `argint()`.
- For the output bitmask, it's easier to store a temporary buffer in the kernel and copy it to the user (via `copyout()`) after filling it with the right bits.
- It's okay to set an upper limit on the number of pages that can be scanned.
- `walk()` in `kernel/vm.c` is very useful for finding the right PTEs.
- You'll need to define `PTE_A`, the access bit, in `kernel/riscv.h`. Consult the RISC-V privileged architecture manual to determine its value.
- Be sure to clear `PTE_A` after checking if it is set. Otherwise, it won't be possible to determine if the page was accessed since the last time `pgaccess()` was called (i.e., the bit will be set forever).
- `vmprint()` may come in handy to debug page tables.

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/2022/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/2022/handin.py/

Optional challenge exercises

- Use super-pages to reduce the number of PTEs in page tables.
- Unmap the first page of a user process so that dereferencing a null pointer will result in a fault. You will have to change `user.ld` to start the user text segment at, for example, 4096, instead of 0.
- Add a system call that reports dirty pages (modified pages) using `PTE_D`.