Computer Science 432
Spring 2022
Lab 4: Page Table Management, due next Monday before lab.

Objective. To be able to create, interpret, and modify page table entries.

Discussion. This week we will be looking carefully at the various processes involved in mapping of virtual addresses to locations in physical memory. This mapping is stored as a partial function, encoded in the entries of the *page table*. The structure of the page table is a highly constrained tree of blocks. Each leaf in the tree contains a number of page table entries that translate the addresses associated with a single block in virtual memory.

It is important that this structure be very carefully maintained because the processor depends on being able to quickly interpret its entries. If we deviate from the format it is very likely that the processor will generate *page faults* which, in `xv6`, are handled by simply killing off the offending process. So: let's avoid that.

We'll extend `xv6` in three simple ways: we'll make a simple optimization to how system calls are implemented, we'll work on a means for printing out page table structures, and we'll build in some very simple support for detecting which pages of a process have been accessed. Necessarily, these are the very first steps in supporting more advanced optimizations.

The Assignment. We'll experiment with the mapping of virtual memory by implementing three different kernel extensions. The basis for this lab is found in the `lab4` repository (use your user id rather than `22xyz`):

```
$ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab4.git
```

While this repository does not include any of the improvements that you've implemented in the last lab, we'll assume the techniques you have developed there will help you with implementations this week. If you have questions about prior work, make sure we talk those through before you attempt these tasks.

Assuming you're up to speed on the readings (through Chapter 3), here is a workflow that will get you through this week's tasks:

1. First, we'll implement a faster `getpid()` system call.

   An approach to making certain kernel calls faster is to store some relatively static state information on a page that is readable by both the user and the kernel, but writable by only the kernel. In this particular case, we'll keep track of the process ID.

   When each process is created, you should map one read-only page at `USYSCALL` (a virtual address defined in `kernel/memlayout.h`). At the start of this page, store a `struct usyscall` (also defined in `memlayout.h`), and initialize it to store the process ID of the current process. We have provided a `ugetpid()` library call on the userspace side that will automatically use the `USYSCALL` mapping.

   Here are some things to think about as you go about this task:

   (a) Convince yourself that, if correctly implemented, this "system call" does not, in fact, require access to the kernel. Instead, it simply locates and returns the process ID from the page of memory at `USYSCALL`.

   (b) You will need to allocate and map a per-process block of memory for conveying state information from the kernel. You should perform the mapping in `proc_pagetable()` in `kernel/proc.c`.

   (c) Make sure that you set the permission bits on this page to allow the user to read, but not write, this structure.

   (d) You will find the `mappages()` routine is useful.

   (e) Make sure that `allocproc()` and `freeproc()` allocate and maintain this new page appropriately.

2. Develop support for printing out a page table. It is frequently useful to get a listing of the virtual address space for a process. To aid us with this task, we'll write a function that prints a page table accessed through a `pagetable_t` pointer.

   Define a function called `vmprint()`. It should take a `pagetable_t` argument and print that page table in a format similar to that shown below. As a test, print the page table associated with the very first process. Add the following:

   ```
   // identify and describe the page table of the first process
   if (p->pid == 1) vmprint(p->pagetable);
   ```

   in `exec.c` just before the `return argc` (around line 118).

   Here are some things to think about:

   (a) You will see that we have placed a stub for `vmprint()` at the bottom of `kernel/vm.c`.

   (b) The `PG...` and `PTE...` macros at the end of the file `kernel/riscv.h` will be helpful. Make sure you use these macros; it will help with maintaining the code, later, if the structure of the page table should change.

   (c) You will find the function `freewalk()` useful in thinking about recursive page table traversals.

   (d) Define the prototype for `vmprint()` in the `vm.c` section of `kernel/defs.h` so that you can call your procedure from `exec.c`.

   (e) Use `%p` in your `printf` calls to print out full 64-bit hex page table entries and addresses, shown in the example. You will also find a helper function that will print out the metadata associated with a particular page table entry.

   Here is what the output from `vmprint()` should look like:

   ```
   Page table at physaddr 0x0000000087f6e000:
    L2:0: pte 0x0000000021fda801, physaddr 0x0000000087f6a000, meta .........V
     L1:0: pte 0x0000000021fda401, physaddr 0x0000000087f69000, meta .........V
      L0:0: pte 0x0000000021fdac1f, physaddr 0x0000000087f6b000, meta .....UXWRV
      L0:1: pte 0x0000000021fda00f, physaddr 0x0000000087f68000, meta ......XWRV
      L0:2: pte 0x0000000021fd9c1f, physaddr 0x0000000087f67000, meta .....UXWRV
    L2:255: pte 0x0000000021fdb401, physaddr 0x0000000087f6d000, meta .........V
     L1:511: pte 0x0000000021fdb001, physaddr 0x0000000087f6c000, meta .........V
      L0:509: pte 0x0000000021fdd813, physaddr 0x0000000087f76000, meta .....U..RV
      L0:510: pte 0x0000000021fddc07, physaddr 0x0000000087f77000, meta .......WRV
      L0:511: pte 0x0000000020001c0b, physaddr 0x0000000080007000, meta ......X.RV
   ```

   The first line of the output displays the pointer passed to `vmprint`. Each line after that describes a single valid page table entry. The root block of the tree, remember, is level 2, whose valid entries refer to level 1 blocks whose entries refer to level 0 blocks of "leaf" table entries that describe actual translations. Each entry—indented in a way to suggest the tree structure—is described by the decimal level and entry numbers, a hexadecimal representation of the page table entry, a hexadecimal representation of the physical address derived from the entry, and the metadata that describes the access control associated with that page. Thus, in the example above, the root node has two valid entries—entry 0 and entry 255. Entry 0 has, itself, a single entry 0 at level 1 and three leaf nodes representing translations associated with virtual pages 0, 1, and 2.

   When you are finished, the page table for the first process is printed just after the machine boots. The physical addresses may be different, but the structure of the page table and page metadata will be the same.

3. Our final modification to the system will allow us identify which pages have been accessed on the system. Some memory management tasks can benefit from information about recently accessed pages. In this part of the lab, you will add a new feature to `xv6` that inspects the *access* bits in the `RISC-V` page table. The `RISC-V` hardware page table walker marks these bits in the leaf PTE whenever it resolves a TLB miss.

   Your job is to implement `pgaccess()`, a system call that reports which pages have been recently 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 pointer to a buffer to store the results into. This buffer is a bitmask, a data structure that uses one bit per page, where the first page corresponds to the least significant bit.

   Here is an approach to implementing `pgaccess()`:

   (a) We have started the process of declaring the `pgaccess()` system call. We have reserved a call number in `kernel/syscall.h` and developed linkage in `kernel/syscall.c`, and made the appropriate userland declarations.

   (b) Start by implementing `sys_pgaccess()` in `kernel/sysproc.c`. Remember, this routine will need to use `argint()` and `argaddr()` to read its arguments. Develop your bitmask in a temporary buffer. Then call `copyout()` to transfer the results back to the user.

   (c) Your experience with `mex` will likely help.

   (d) You must support the scanning of 32 pages, but you can set an upper bound to any reasonable number, if that is helpful.

   (e) The `walk()` routine in `kernel/vm.c` can help with finding the appropriate page table entries.

   (f) We have added the definition for `PTE_A`, the page table entry bit in the metadata that is set whenever the hardware page table walker encounters the page table entry.

   (g) As you collect the page access bits, you should clear the bits so that you can detect later accesses. If you don't do this, the `PTE_A` bit is set permanently.

   (h) You may find that your `vmprint()` routine will be helpful in debugging `sys_pgaccess()`.

4. When you are have implemented your page table extensions, you will find the `user/pgtabletest` program will be helpful in performing basic tests of your extensions. Make sure you add, commit, and push your changes for review and grading.

Thought Questions. Please think about the following questions in preparation for meeting in small groups:

1. In our manipulation of the `USYSCALL` page we modified `allocproc()` and `freeproc()`.
   Should `fork()` be modified as well?

2. What other system calls might make use of the `USYSCALL` page?

3. Explain, in the output from `vmprint` in terms of the user process layout.
   What does page 0 contain?
   What does page 2 contain?
   What does the third to last page contain?
   Be prepared to explain the permissions associated each of the first process's pages.

4. Bit 7 of the page table entry is the *dirty bit*. It keeps track of those pages that have been modified since they were brought into memory.
   How do you think this bit is set?
   Why might it be useful to know which pages are dirty?

⋆