```
6.S081 2020 Lecture 7: Q&A

Plan: answering your questions
  Approach:
    walk through staff solutions
    start with pgtbl lab because it was the hardest
    your questions are at bottom of this file

Pgtbl lab comments
  few lines of code, but difficult-to-debug bugs
  - worst case: qemu/xv6 stops running
  - "best" case: kernel panic
  hard to debug for staff too
  - there are so many possible reasons why
  - you discovered once we hadn't seen yet
  likely to be the most challenging lab
  - historically the first VM lab is hard
  - this year too, even though we made a new lab to provide a gentler intro to VM

Part 1 of pgtbl lab
  Explain vm output in terms of fig 3-4

   page table 0x0000000087f67000
    ..0: pte 0x0000000021fd8c01 pa 0x0000000087f63000 fl 0x0000000000000001
    .. ..0: pte 0x0000000021fd8801 pa 0x0000000087f62000 fl 0x0000000000000001
    .. .. ..0: pte 0x0000000021fd901f pa 0x0000000087f64000 fl 0x000000000000001f
    .. .. ..1: pte 0x0000000021fd840f pa 0x0000000087f61000 fl 0x000000000000000f
    .. .. ..2: pte 0x0000000021fd801f pa 0x0000000087f60000 fl 0x000000000000001f
    ..255: pte 0x0000000021fd9801 pa 0x0000000087f66000 fl 0x0000000000000001
    .. ..511: pte 0x0000000021fd9401 pa 0x0000000087f65000 fl 0x0000000000000001
    .. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000 fl 0x0000000000000007
    .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000 fl 0x000000000000000b

    what is entry 0, 1, and 2?
    what is 510 and 511?
    - 511: trampoline
    - 510: trapframe
    why are the protection bits as they are?
    - 510: RWV
    - 511: XV
    - 1: no U
    are the physical addresses contiguous?

Part 2 of pgtbl lab
  Add a kernel page table to each process in prep for Part 3
  - exact copy of *the* kernel_pagetable

  How hard can this be?  hard! some reasons:
  - xv6 code specialized to one kernel page table
    no kvmcreate_page_table()
  - kvminit() isn't the full story
    also mappings in procinit(), and virtio_disk.c
  - cleanup must be done care
    don't free kernel stacks
  - easy to make a small error
    even small errors are time consuming

  Two possible approaches to part II
  1) generalize kvminit()
    create a new per-process kernel page table
  2) share kernel_pagetable
    make the per-process kernel page table share much with kernel_pagetable
    - my solutions takes approach 2; other staff members too approach 1
    - i took approach 2 mostly out of "laziness"
```

```
       don't have to think too hard what is in the kernel_pagetable
       didn't want to write much code
       - not sure it is shorter, though
       approach 2 has some benefits I didn't realize
       - avoided certain class of bugs (e.g., kernel stacks)

   Implementation approach:
     small incremental baby steps
     keep existing code working
   allows comparing old and new easily
   allows rolling back easily

   Approach 2
   - kvmcreate()
     copy entries 1..512
     add a few devices that live in 0
     - anything below 1G (i.e., all devices)
   - kvmfree()
     only entry 0 is worth considering
     just need to free intermediate pages (see 3-level pic)
     a bit ugly
     where to call kvmfree()
     - exit()?
     - freeproc()?
  - scheduler()
  - usertrapret()

 Part 3 of pgtbl lab
   map user pgtbl of process into bottom of process's kernel page table
   - draw figure
   why? convenient for kernel programmers
   - can replace copyin with copy_new()
   copyin_new is just a memmove(), avoid the walkaddr()'s
   - hardware can do the walks for us

   mapping user page tables into kernel page table
   - kvmmapuser()
     copies upte into kpte, adjust permissions

   Using kvmmapuser():
   - userinit()
   - exec()
    no need to replace kernel page table
   - fork()
   - sbrk()

   memmove()
   - why the checks before the memmove()?
   - what does each case cover

 Syscall lab
   systems calls look like functions calls
   BUT they are not function calls
   - U/K boundary transition
   user/usys.pl
   - generates stubs for stubs
   kernel/syscall.c
   - demultiplexes system calls
   trace
   - proc.h
   - fork()
   sysinfo
   - copyout()

 Util lab
```

```
  primes
  - close fd's to terminate correctly
  xargs
  - setting up argv

=== Questions ===

HW

===

In the page table lab, the kernel's hardware address translator won't
allow kernel to dereference user pages. More specifically, if the page
table entry has "PTE_U" bit set, which means user can access it, then
kernel won't be able to access it through the hardware address
translator unless we set "PTE_U" to false in the per-process kernel
page table. Is there a specific reason for this?

===

Why is the kernel not allowed to read memory mapped with the PTE_U
flag? To prevent malicious memory from being read?

===

In the last lecture, it was mentioned that even the kernel in
supervisor mode could not directly edit anything and everything in the
memory. The kernel accesses memory through the kernel page table, and
thus is limited. However, couldn't the kernel just set the satp
register to 0, and then there is no MMU translation, the CPU directly
accesses physical addresses. Could that situation not arise? That
certainly the case before page tables are set up and paging is
enabled.

===

A minor question: why the specific order of the trapframe in proc.h
(t0..t2, then s0, s1, a0..a7, s2..s11)?

===

PGTBL LAB

I'm confused why the kernel can directly dereference the pointer in
part 3 of pagetable because we have now mapped the user's virtual
address into the user's kernel pagetable. Since it is still a virtual
address, doesn't it need to just walk the user's kernel page table? Or
is it because the user code now enters the kernel, and the kernel's
pagetable directly maps virtual addresses to physical addressses?
However, in my implementation, it seems to be that the kernel still
maps the same virtual addresses to the same physical addresses as the
user's page table, , which is not a direct mapping.

===

In the pgtbl lab, during Part 3 (copyin), I ran into an interesting
bug while changing a process's user mappings. In fork(), I was calling
my helper function to copy the user mappings from the process's user
page table to its kernel page table. When I tried to copy the parent
process's user page table, running "usertests" in the xv6 kernel just
caused it to return and none of the tests were run. However, copying
the child's user page table worked. Shouldn't either instruction be
ok, since the child's user page table was just copied over from the
parent?
```

===

For the page tables lab, what was the correct way to copy and change the process's kernel page table the same way in fork(), exec(), and sbrk()? I tried to copy over a range of entries from the process's page table into the process's kernel page table directly, but my code was panicking when trying to use mappages() after my change to the 3 functions.

===

If xv6 was written with the modifications from the pgtbl lab, we would not need to worry about switching the pagetables in the trampline code, right? Would there even be a need for the trampoline code to be mapped in the user pages? My understanding was that that code can be mapped in the per-process kernel pgtbls and be used from there during the trap handling. I was also curious about why when a PTE has the PTE_U bit is set, code in supervisor mode cannot access that page table. It is confusing to me because I thought that the kernel has the whole memory mapped and it can modify any virtual address even if some user data lives there. In the last lab, we could even know the exact va that the user pgtbl has for each of the pa that it uses and I think maybe be able to corrupt user's data.

===

Is there a more efficient way to free each process's kernel pagetable? I basically copied the freewalk function and removed the panic for leaves. Although this seemed fast enough for the tests, it still seemed a little slow. When I tried to optimize it by reducing recursion depth, I ran into troubles. Overall, this seems like a computationally intensive task, especially if all physical memory is mapped in the kernel pagetable for all processes. I was a little worried about breaking something, but could I have stopped mapping all physical memory?

===

Do operating systems using hierarchical pagetables ever set up the pagetable pages so that the lower parts of the hierachy are (atleast in part) shared?

===

For the pagetable lab, I was trying to implement copyin/copyinstr. However, I got stuck in an infinite loop with the scheduler because the scheduler seemed to have no process that needs to be run. I am curious as to what could happen to the pagetable so that the OS would stop a process from running (especially the main process). What part of the code does the operating system set the state of a process to not running if the pagetable is broken?

===

I don't understand why when I moved

```
w_satp(MAKE_SATP(kernel_pagetable));
sfence_vma();
```

outside of

```
if(found == 0) {
  intr_on();
  asm volatile("wfi");
}
```

in scheduler that everything in part 2 of the page table lab
worked. Because before then, it worked to a certain degree, where
sometimes it'd pass and sometimes it'd fail.

===

In the pagetable lab, is there a significant impact on performance
from copying to per-process kernel page tables so frequently? If so,
how does this balance out with the improvements that come from being
able to directly de-reference user address in kernel space?

===

How are pipes implemented in xv6, and how would the changes to the
page tables implemented in the pgtbl lab effect this implementation?
Do the per-process page tables hinder each process' ability to
communicate with each other?

===

1. Why did uvmfree/freewalk originally panic at a leaf, and why did we need to
change that.

2. Why does the inclusion of the user PTE_U bit not allow the kernel to access
the PTE?

===

The lab mentioned that "the goal of this section and the next is to
allow the kernel to directly dereference user pointers." However, I
didn't understand how this adds functionality to the kernel, or why
this is an important function for the kernel to be able to do.

===

For the page table lab, I had some questions on access rights. If the
user bit is set, does that mean in user mode both read/write/execute
options can be performed, or can only read/write/execute be done in
kernel mode?

Also, are there pages in the kernel page table accessible in user mode?

===

In the pgtbl lab, we added kernel pagetables to each process in order
to simplify reading userspace pointers. Was there any reason we
couldn't do the same for copying data out to userspace? I think you'd
have to be a little careful in exec(), since copyout is used there
while setting up the process, but apart from that?

Also, one of the challenge exercises for that lab is to remove the
PLIC limit. How would you do that--set up a new pagetable for PLIC
access, and switch to that when trying to access PLIC registers?

===

I was talking to some friends who solved the pagetable part 3 by using
a modified uvmcopy but they needed to limit the pages they were
copying to pass the runtime limit. What's taking so long in that
function? Does the walking make the copy function less optimized for
the job?

===

Question: About the page table lab, when I was debugging my code, I

realized for reparent test, I sometimes pass it but I sometimes fail. I fixed this issue by realizing I need to switch back to kernel_pagetable in the scheduler after exiting the process's kernal pagetable. I wonder why such mistakes would cause reparent to behave non-deterministaclly? Was there some sort of race condition?

===

In our "copy user page table to kernel page table" functions in the pagetable lab, why was it necessary to panic when a pagetable entry didn't exist or was not valid for a given virtual address? Why couldn't we have just skipped that virtual address? More broadly, how could we assume that all of the user page table values that we were copying over were present and valid?

===

It really messes with my head that you have to define a C function for allocating memory (kalloc). This is one of the first things xv6 does in main.c, directly after which it creates the kernel page table (using a call to kalloc). Given the way kalloc works, is the kernel page table always going to be at the "end" address in the kernel's virtual memory space since its the first piece of memory to be allocated? And do you have to be careful about using memory prior to calling kinit in main.c?

===

This question is regarding the pgtbl lab. I had some confusion about the exact motivation for the lab, specifically creating a separate kernel page table for each process. I wasn't sure what the lab meant by "directly dereferencing the user address". From my understanding, what the kernel currently does to translate the virtual address of the user process is to "walk" down the user pagetable.

So if you essentially combine the kernel page table and user page tables, wouldn't you still "walk" down the new per-process kernel page table to get the physical address? I don't understand how this changes anything since the two approaches differ only in which page table the kernel is walking down.

===

For question 1 in the page table lab, was page 1 the guard page? This made sense to me since it has the user flag off so it would cause errors if the user tried to access it. However I believe the textbook said that the guard page will have its valid bit off in which case it would not show up in vmprint.

====

Why does the initial page table (as printed in vmprint) have indices 0 and 255?I deduced that these pages should eventually point towards the initial user program data (text, guard page and stack) and the trampoline/trapframe respectively. However, the trampoline should be at the very top of the virtual address space, so it should have the highest possible virtual address. Therefore, shouldn't the root page table contain pages 0 and 511 rather than 0 and 255?

====

VM IN GENERAL

====

How was copyin with per process kernel pagetables exploited, as
mentioned in the pgtbl lab?

===

This is less of a question, but I'd love to hear more about how Xv6 compares to
various other operating systems that we might use in our day-to-day lives.

I don't have specific questions, but for example:

- How does Linux set up its page tables? The lab mentioned this may have
  changed with Spectre/Meltdown, but what do they do now?

- Do we know how pages/traps are handled in Windows or macOS, or what tradeoffs
  were made in the design decisions there? I read somewhere that Windows XP
  source code leaked - did we learn anything interesting from that, or does
  Microsoft just use standard techniques?

- Does x86-64 have instructions that significantly simplify or combine or
  increase performance of particular parts of the pagetable or traps chain of
  code in interesting ways?

- How active is the development of OS techniques like syscalls, pagetables,
  etc.? Are they mostly considered a "solved" problem by modern operating
  systems and only really experimented on in research?

===

SYSCALL LAB

===

Could you possibly walk through the roles of the different places we
had to update when we added a new system call? (e.g. user/user.h,
user/usys.pl, kernel/syscall.h, and a few others)

I'm mostly wondering (1) in what order these are visited when the code
is compiled and (2) why there isn't a more centralized way to do this?
For example, in kernel/syscall.c, why do we need to manually edit both
the extern list and the syscalls array beneath it?

===

UTIL LAB

===

One of the challenges for the first lab was to add history/tab
completion. However, one thing that gave me some trouble was being
able to parse single keystrokes (namely the arrow keys). Is there a
way to read a single character (not having to press enter)?