به نام خدا

گزارش Lab2 سیستم‌عامل پیشرفته

استاد:

دکتر جوادی

دانشجویان:

سید علیرضا حسینی – ۴۰۱۱۳۱۰۰۵

امیررضا زارع – ۴۰۱۱۳۱۰۰۸

تمرین ۱:

Exercise 1. In the file `kern/pmap.c`, you must implement code for the following functions.

```
boot_alloc()
page_init()
page_alloc()
page_free()
```

You also need to add some code to `x64_vm_init()` in `pmap.c`, as indicated by comments there. For now, just add the code needed before the call to `check_page_alloc()`.

You probably want to work on `boot_alloc()`, then `x64_vm_init()`, then `page_init()`, `page_alloc()`, and `page_free()`.

`check_page_alloc()` tests your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct.

همان‌طور که خواسته شده در فایل pmap.c توابع boot_allic() و page_init() و page_alloc() و page_free()را تغییر دادیم و تغییرات را اعمال کردیم. سپس پس از اجرای jos پیغام check_page_alloc() را دریافت کردیم.

تمرین ۲:

Exercise 2. Read chapters 4 and 5 of the AMD64 Architecture Programmer's Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.1). Although JOS relies most heavily on page translation, you will also need a basic understanding of how segmentation works in long mode to understand what's going on in JOS.

در این تمرین ارجاعی به فصل چهار و پنج کتاب AMD64 Architecture Programmer’s Manual شده است که در مورد مدیریت حافظه معماری AMD64اشاره می‌کند.

تمرین ۳:

Exercise 3. While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press `Ctrl-a c` in the terminal (the same binding returns to the serial console).

Use the `xp` command in the QEMU monitor and the `x` command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges of virtual memory are mapped and with what permissions.

Question

1. Assuming that the following JOS kernel code is correct, what type should variable x have, `uintptr_t` or `physaddr_t`?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

تابع return_a_pointer() یک آدرس مجازی برمی‌گرداند (همه پوینتر ها آدرس مجازی هستند)، x باید uintptr_t باشد. سپس x شامل نمایش عدد صحیح آدرس مجازی مقدار موردنظر خواهد بود. نوع x مشخص نشده است، در واقع ‘mystery_t’ یک اشاره‌گر به نوع داده‌ای نامعلوم باشد.

از آن جایی که قطعه کد اشاره‌گر را تغییر می‌دهد نمی‌تواند آدرس فیزیکی را با دورزدن ترجمه MMU برگرداند.

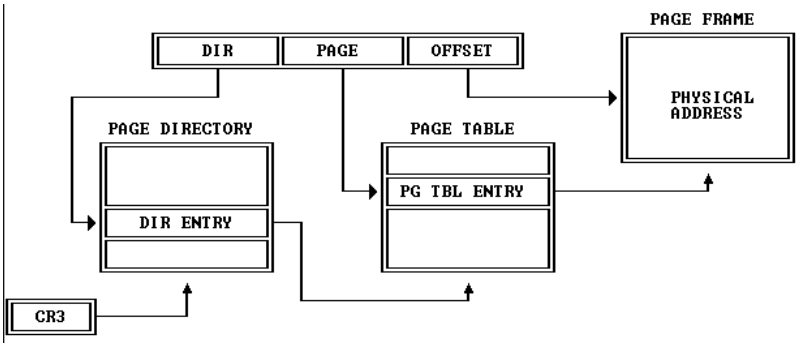
تمرین ۴:

Exercise 4. In the file `kern/pmap.c`, you must implement code for the following functions.

```
pml4e_walk()
pdpe_walk()
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

`page_check()`, called from `x64_vm_init()`, tests your page table management routines. You should make sure it reports success before proceeding.

نحوه ترجمه آدرس در معماری‌های x86 که در کلاس درس بیان شد:



فریم‌های حافظه فیزیکی 4KB هستند، جدول صفحه در فریم‌های حافظه فیزیکی قرار دارد؛ لذا جدول صفحه هم 4KB است. ۱۲ بیت پایین نشان دهنده offsetاست و برای نشان دادن ادرس فیزیکی(فریم)۲۰ بیت مورد نیاز می باشد. در این سوال با توجه به آدرس پایه دایرکتوری صفحه و یک آدرس مجازی، باید ورودی جدول صفحه را برگردانیم .اگر صفحه جدول صفحه وجود ندارد، یکی را اختصاص دهیم.

آدرس‌های موجود در فهرست صفحه و جدول صفحه همه آدرس‌های فیزیکی هستند. اما باید آدرس مجازی را برگردانیم. در نهایت در فایل pmap.c برای توابع walk_e4pml() و page_remove()و page_lookup() و boot_map_region()و Pgdir_walk()و insert_page() pdpe_walk() کد پیاده‌سازی کردیم.

تمرین ۵:

Exercise 5. Fill in the missing code in `x64_vm_init()` after the call to `page_check()`.

Your code should now pass the `check_boot_pml4e()` check.

Question

2. What entries (rows) in the page directory have been filled in at this point for the 4th page directory pointer entry(Make sure you understand why 4th pdpe entry)? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
511	?	Page table for top 2MB of phys memory
510	?	?
.	?	?
.	?	?
.	?	?
184	0xF0000000	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question?]

3. We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
4. What is the maximum amount of physical memory that this operating system can support? Why?
5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?
6. Read the simple page table setup code in `kern/bootstrap.S` .
The bootloader tests whether the CPU supports long (64-bit) mode. It initializes a simple set of page tables for the first 4GB of memory. These pages map virtual addresses in the lowest 3GB to the same physical addresses, and then map the upper 256 MB back to the lowest 256 MB of memory. At this point, the bootloader places the CPU in long mode. Note that our bootloader transitioning to long mode isn't strictly necessary; typically, a bootloader only runs in long mode to load a 64-bit kernel at a high (>4 GB) virtual memory address.
Note that once we transfer control to the kernel, the kernel assumes the CPU supports 64-bit mode. Assuming the kernel was loaded in the lower 4GB of virtual address space, the kernel itself could test whether the CPU supports long mode and determine dynamically whether to run in 64 or 32-bit mode. Of course, this would substantially complicate the boot process.

هسته و محیط کاربر را در یک فضای آدرس قرار داده‌ایم. چرا برنامه‌های کاربر قادر به خواندن یا نوشتن حافظه هسته نیستند؟ چه مکانیسم‌های خاصی از حافظه هسته محافظت می‌کند؟

پاسخ: به دلیل bit permission ها

حداکثر مقدار حافظه فیزیکی که این سیستم‌عامل می‌تواند پشتیبانی کند چقدر است؟ چرا؟

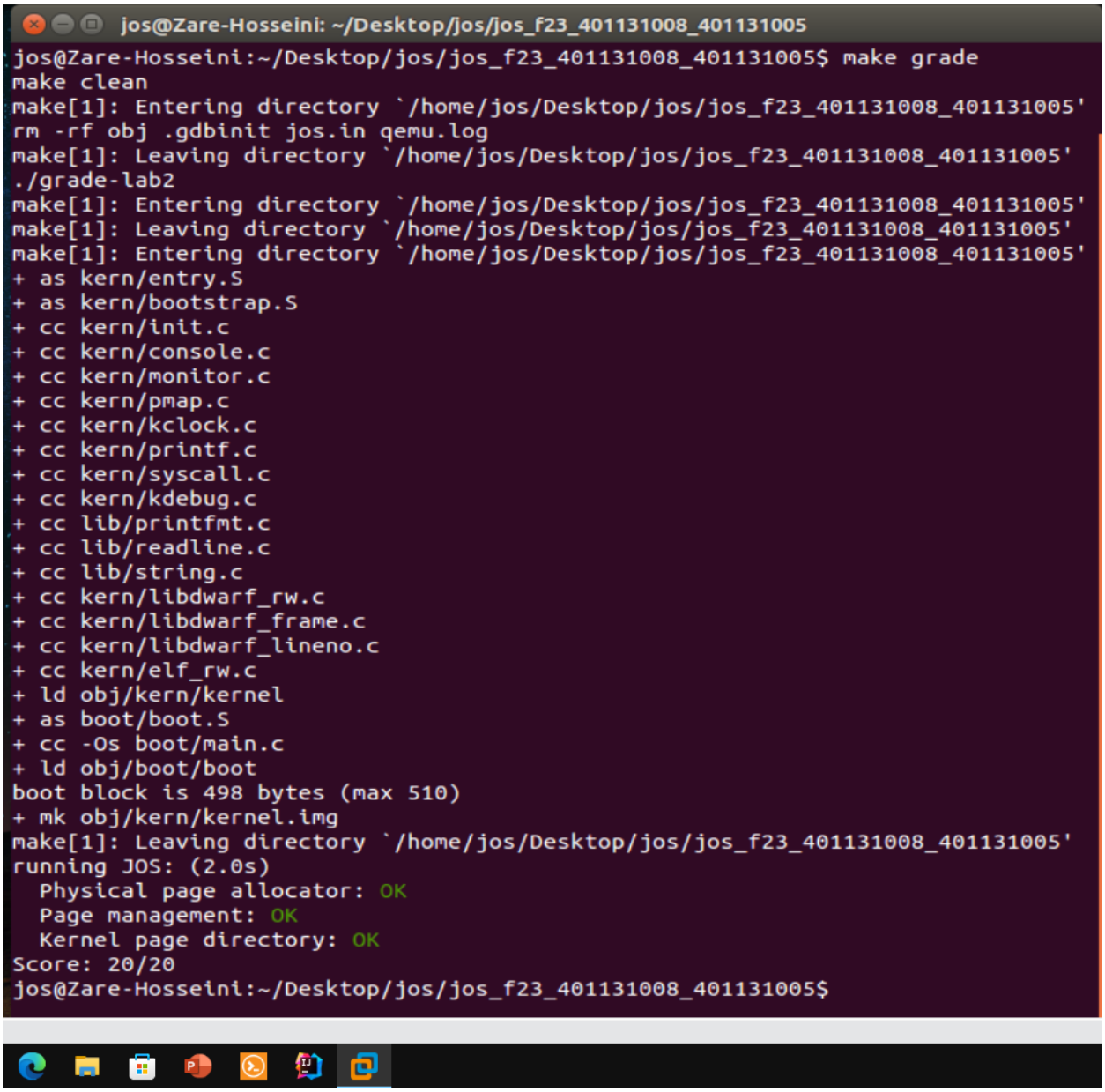
512*512*4KB = 1GB

چقدر فضای سربار برای مدیریت حافظه وجود دارد، اگر واقعاً حداکثر مقدار را داشته باشیم حافظه فیزیکی؟ این سربار چگونه تجزیه می‌شود؟

پاسخ: برای صفحات فیزیکی، ۸ مگابایت برای PageInfo struct و ٦ مگابایت برای جداول صفحه، در نهایت ٦ کیلوبایت برای دایرکتوری یک صفحه استفاده خواهد شد.

تنظیمات جدول صفحه را در S.entry/kern و c.entrypgdir/kern مجدداً مشاهده کنید. بلافاصله پس از اینکه صفحه‌بندی را روشن می‌کنیم، EIP هنوز عدد پایینی است کمی بیش از ۱ مگابایت(.در چه نقطه ای به اجرای در EIP باالتر از KERNBASEتغییر می کنیم؟ چه چیزی این امکان را برای ما فراهم میکند که بین زمانی که صفحهبندی را فعال میکنیم تا زمانی که شروع به اجرا در EIPباالتر از KERNBASE میکنیم، به اجرای با EIP پایین ادامه دهیم؟ چرا این انتقال ضروری است؟

پاسخ: بعد از دستور jmp* %eax . این امکان وجود دارد؛ زیرا در c.enterpgdirآدرس مجازی ۹ تا ٦ مگابایت را به آدرس فیزیکی ۹ تا ٦ مگابایت نگاشت میکند. این موضوع الزم است زیرا بعداً یک pgdir_kern بارگذاری می شود و (M4 0, [va رها می شود.


jos@Zare-Hosseini: ~/Desktop/jos/jos_f23_401131008_401131005
jos@Zare-Hosseini:~/Desktop/jos/jos_f23_401131008_401131005$ make grade
make clean
make[1]: Entering directory `/home/jos/Desktop/jos/jos_f23_401131008_401131005'
rm -rf obj .gdbinit jos.in qemu.log
make[1]: Leaving directory `/home/jos/Desktop/jos/jos_f23_401131008_401131005'
./grade-lab2
make[1]: Entering directory `/home/jos/Desktop/jos/jos_f23_401131008_401131005'
make[1]: Leaving directory `/home/jos/Desktop/jos/jos_f23_401131008_401131005'
make[1]: Entering directory `/home/jos/Desktop/jos/jos_f23_401131008_401131005'
+ as kern/entry.S
+ as kern/bootstrap.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/pmap.c
+ cc kern/kclock.c
+ cc kern/printf.c
+ cc kern/syscall.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ cc kern/libdwarf_rw.c
+ cc kern/libdwarf_frame.c
+ cc kern/libdwarf_lineno.c
+ cc kern/elf_rw.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld obj/boot/boot
boot block is 498 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/jos/Desktop/jos/jos_f23_401131008_401131005'
running JOS: (2.0s)
Physical page allocator: OK
Page management: OK
Kernel page directory: OK
Score: 20/20
jos@Zare-Hosseini:~/Desktop/jos/jos_f23_401131008_401131005$