Project 0: Getting Real

Preliminaries

> Fill in your name and email address.

Yuxing Xiang 2000012959@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Booting Pintos

> A1: Put the screenshot of Pintos running example here.

- **Pintos booted in qemu mode**

```
root@78e2262fb243:~/pintos/src/threads/build# pintos --
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,file=/tmp/dbhIVP_g8J.dsk -m 4 -net none -no
graphic -monitor null
Pintos hda1
Loading...........
Kernel command line:
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  117,760,000 loops/s.
Boot complete.
```

- **Pintos booted in Bochs mode:**

```
root@78e2262fb243:~/pintos/src/threads/build# pintos --bochs --
squish-pty bochs -q
========================================================================
                       Bochs x86 Emulator 2.6.2
               Built from SVN snapshot on May 26, 2013
                 Compiled on Nov 18 2021 at 12:44:44
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
Pintos hda1
Loading...........
Kernel command line:
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  102,400 loops/s.
Boot complete.
```

Debugging

QUESTIONS: BIOS

> B1: What is the first instruction that gets executed?

```
ljmp $0xf000, $0xe05b
```

> B2: At which physical address is this instruction located?

At 0xffff0, persumably the last instruction coded in BIOS, as well as the first instruction fetched.

QUESTIONS: BOOTLOADER

> B3: How does the bootloader read disk sectors? In particular, what BIOS interrupt is used?

After `int $0x13` at 0x7d30 in function `read_sector`, control in transferred to 0xfd6f1, which is the BIOS code section, where disk sectors are read.
For more details, according to http://www.ctyme.com/intr/int.htm, `int $0x13` functions as an external storage access instruction, with register `%ah` (higher 8 bits of `%ax`) as parameter. Prior to this instruction in `read_sector` is

```
    mov $0x42, %ah                      # Extended read
```

As is documented, `%ah = 0x2` configures `int $0x13` to "read hard disk sector(s) into memory", and `%ah = 0x42` further tells the CPU to read in exteneded LBA mode, which also match the comment above.

> B4: How does the bootloader decides whether it successfully finds the Pintos kernel?

In function `check_partition` at 0x7c52, the instruction

```
cmpb $0x20, %es:4(%si)
```

checks if current partition on the hard disk drive is a Pintos partition. If not, next partition will be checked.

> B5: What happens when the bootloader could not find the Pintos kernel?

Apparently in `next_drive` (which is called when all partitions on a drive is invalid), after `inc %dl`, a `jnc 7c1e <read_mbr>` is followed. Considering that `%dl` is the lower 8-bit register, this tells us that up to 128 hard disks (`%dl` is initialized to 0x80, which represents hard disk 0) are searched before the bootloader gives up on trying.

If this happens, in `no_boot_partition`, a string "Not Found" is outputted, and `int $0x18` (defaultly means "diskless boot hook") is executed to tell BIOS that bootloading has failed.

> B6: At what point and how exactly does the bootloader transfer control to the Pintos kernel?

At the case of a bootable Pintos partition, `load_kernel` is executed, in which the kernel is read from disk (again by calling `read_sector`). After that, these code finally transfer control to the Pintos kernel:

```
    mov $0x2000, %ax
    mov %ax, %es
    mov %es:0x18, %dx
    mov %dx, start
    movw $0x2000, start + 2
    ljmp *start
```

Here `start` is actaully a temporary memory location, since legacy 8086 processor cannot jump to an absolute segment:offset address. With `ljmp *start`, control goes to Pintos kernel.

QUESTIONS: KERNEL

> B7: At the entry of pintos_init(), what is the value of expression `init_page_dir[pd_no(ptov(0))]` in hexadecimal format?

First, `ptov()`, as is found in `vaddr.h`, converts physical address to virtual address, which is done by

```
return (void *) (paddr + PHYS_BASE);
```

where `PHYS_BASE` is alias for `LOADER_PHYS_BASE` defined in `loader.h`:

```
#define LOADER_PHYS_BASE 0xc0000000     /**< 3 GB. */
```

This shows `ptov(0)` should be 0xc0000000.

Second, `pd_no()` defined in `pte.h` obtains page directory index from a virtual address, via:

```
return (uintptr_t) va >> PDSHIFT;
```

The virtual address is familiarly structured as:

```
 31                  22 21                  12 11                   0
+----------------------+----------------------+----------------------+
| Page Directory Index |   Page Table Index   |    Page Offset       |
+----------------------+----------------------+----------------------+
```

and `PDSHIFT` is just defined to be 22. Thus, `pd_no(0xc0000000)` should be 0x300.

Finally, at the entry of `pintos_init()`, `init_page_dir` is an uninitialized global variable of type `uint32_t *`, thus `init_page_dir[0x300]` should be `*(0 + 4 * 0x300)`, which is 0 by gdb.

(A problem though: it seems that the .bss section is not zeroed before calling `bss_init()`, so considering `init_page_dir` to be 0 can be debatable. Nevertheless, my gdb experiment does print it as 0.)

Update

It turns out that gdb is smart enough to execute certain functions and print their return values, which I honesty didn't know. This way, answers to questions above are easily acquired:

```
Breakpoint 5, pintos_init () at ../../threads/init.c:79
(gdb) p ptov(0)
=> 0xc000efef:  int3
$9 = (void *) 0xc0000000
(gdb) p pd_no(ptov(0))
=> 0xc000efef:  int3
=> 0xc000efef:  int3
$10 = 768
(gdb) p init_page_dir[pd_no(ptov(0))]
=> 0xc000efef:  int3
=> 0xc000efef:  int3
$11 = 0
```

Nonetheless, I figure that going through the code was definitely worth something on its own.

> B8: When `palloc_get_page()` is called for the first time,

>> B8.1 what does the call stack look like?

```
(gdb) bt
#0 palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at ../../threads/palloc.c:112
#1 0xc00203aa in paging_init () at ../../threads/init.c:168
#2 0xc002031b in pintos_init () at ../../threads/init.c:100
#3 0xc002013d in start () at ../../threads/start.S:180
```

>> B8.2 what is the return value in hexadecimal format?

```
(gdb) finish
=> 0xc00203aa <paging_init+17>: add    $0x10,%esp
0xc00203aa in paging_init () at ../../threads/init.c:168
Value returned is $2 = (void *) 0xc0101000
```

The return value is 0xc0101000.

>> B8.3 what is the value of expression `init_page_dir[pd_no(ptov(0))]` in hexadecimal format?

In `paging_init()`, `init_page_dir` is initialized as

```
init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
```

which is just 0xc0101000 from previous question. Combined with B7, the answer should be `*0xc0101c00`, which by gdb is again 0.

> B9: When palloc_get_page() is called for the third time,

> > B9.1 what does the call stack look like?

```
Breakpoint 3, palloc_get_page (flags=PAL_ZERO) at ../../threads/palloc.c:113
(gdb) bt
#0  palloc_get_page (flags=PAL_ZERO) at ../../threads/palloc.c:113
#1  0xc0020a81 in thread_create (name=0xc002e895 "idle", priority=0, function=
    0xc0020eb0 <idle>, aux=0xc000efbc) at ../../threads/thread.c:178
#2  0xc0020976 in thread_start () at ../../threads/thread.c:111
#3  0xc0020334 in pintos_init () at ../../threads/init.c:119
#4  0xc002013d in start () at ../../threads/start.S:180
```

> > B9.2 what is the return value in hexadecimal format?

```
(gdb) finish
Value returned is $7 = (void *) 0xc0103000
```

The return value is 0xc0103000.

> > B9.3 what is the value of expression `init_page_dir[pd_no(ptov(0))]` in hexadecimal format?

```
(gdb) p/x init_page_dir[0x300]
$9 = 0x102027
```

Exciting! This should just be a newly created PDE, with the structure:

```
31                                 12 11                     0
+------------------------------------+------------------------+
|         Physical Address           |         Flags          |
+------------------------------------+------------------------+
```

This tells us a new page is now at physical address 0x102000, and permission bits 0x027 are just the normal configuration.
We do get to see something interesting at that memory location:

```
(gdb) x/100x 0xc0102000
0xc0102000:     0x00000003      0x00001003      0x00002003      0x00003003
0xc0102010:     0x00004003      0x00005003      0x00006003      0x00007003
0xc0102020:     0x00008003      0x00009003      0x0000a003      0x0000b003
0xc0102030:     0x0000c003      0x0000d003      0x0000e063      0x0000f003
...
```

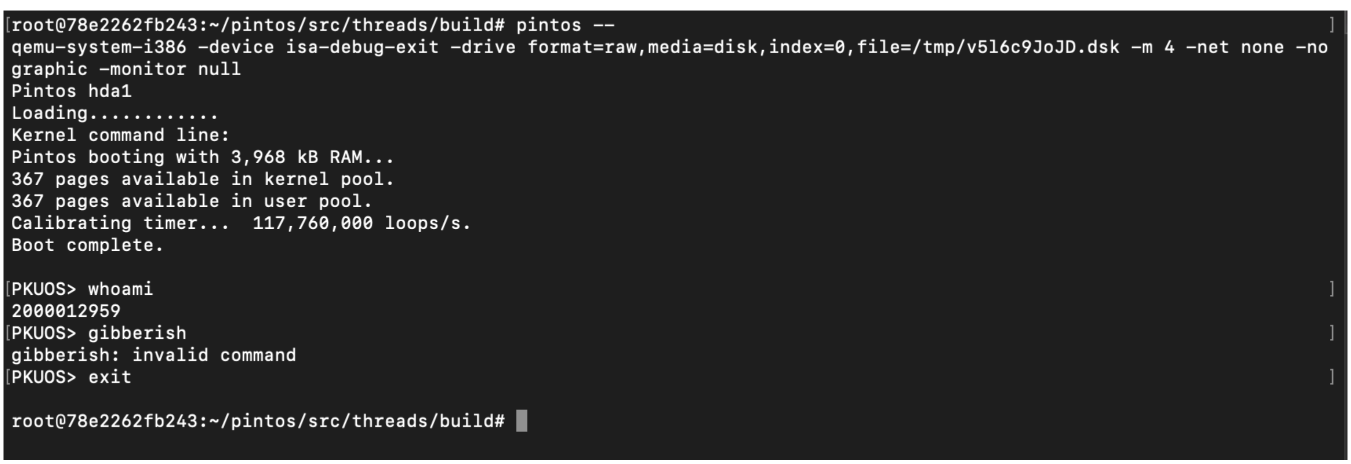
which are persumably the actual pages in this page-directory-page. In fact, considering how `ptov()` is currently implemented, these should the kernel pages (also implied by permission bit `0x003`, the `U/S` bit is clear). In fact, if we now check the BIOS memory location:

```
(gdb) x/10i 0xc0007c00
0xc0007c00:  sub    %eax,%eax
0xc0007c02:  mov    %eax,%ds
0xc0007c04:  mov    %eax,%ss
0xc0007c06:  mov    $0xf000,%sp
...
```

These are actually BIOS code!

Kernel Monitor

> C1: Put the screenshot of your kernel monitor running example here. (It should show how your kernel shell respond to `whoami` , `exit` , and `other input` .)



My kernel shell also supports basic command line parsing, which enables it to correctly respond to spaces, empty lines and commands with arguments.

Besides, I implemented support for backspaces and left/right arrow keys, just to make my own life easier. Details of this are shown below.

> C2: Explain how you read and write to the console for the kernel monitor.
>
> *As I implemented some more features, this is more of an overview of my kernel shell. For anwser to **how I/O is done**, refer to **Echoing User Input**.*

Logistics

For expandability, the interactive part of the kernel monitor is placed in a new function in `init.c` called `run_monitor()`, with an extra `ks_parseline()` function for parsing the command line. The parameters `KS_BUFFER_SIZE` and `KS_MAXARGS` along with a utility struct `cmdline_tokens` are defined in `init.h`.

Core features

· Echoing User Input

`input_getc()` from `devices/input.c` is called for reading user key strokes one at a time, and `putchar()` is used to echo that character back onto terminal. The symbols read are also stored in a string `input` for later parsing; when a `\r` or `\n` is read, loop is closed, and parsing starts.
A safety measure is implemented when user types more characters than `KS_BUFFER_SIZE`: the shell would detect that, stop adding to `input`, and refuse to echo back, providing a nice visual hint for users to stop smashing their keyboards.

· Advanced Edit Control

Our kernel supports left/right arrow keys and backspace. This is done by detecting special input characters: `0x7f(del)` for backspace and `0x1b 0x5b ...` for arrow keys.
Along with this, to correctly modify `input` and refresh terminal, we further introduce two marker variables `cursor` and `end`.
`cursor` is the position in `input` that user is now modifying, and `end` just points to the end

of `input`.

The gist is, when a normal (printable) character is read, in lieu of appending it to `input`, we **insert** it to `input` at location `cursor`, modify markers accordingly, and use a bunch of `putchar('\b')` to overlap printed characters on terminal. The backspace works similarly, and the left/right arrow keys just changes the `cursor` value.

Up/down arrow keys are simply disabled, not only because we feel a history system can be an overkill, but it would add more complexity to how markers are managed.

This is a somewhat messy approach, but we are happy with how it makes our shell feel much easier to use.

· Parsing Command Line

Improved upon simply `strcmp` the `input` with certain strings, we build a basic parser for interpreting command line with multiple arguments and extra white-spaces. One bonus effect is making our shell neglect empty lines correctly.

The core function of this is implemented in `ks_parseline()`, which is modified from my code in shelllab back in ICS course (well, this feature is optional anyway). Using `strspn()` to partition command line into tokens, it could be expanded upon to recognize I/O redirection and background flag.