Lab 0: Design Doc

Preliminaries

- Name: 张龄心
- Email: 2100013018@stu.pku.edu.cn
- Reference:

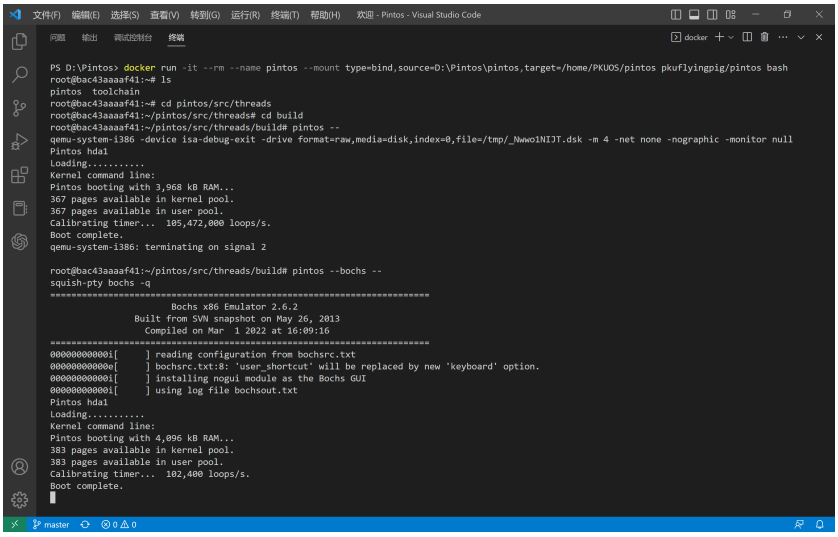
  https://blog.csdn.net/u011011827/article/details/125656962?ops_request_misc={"request_id":"167772676616800188575949","scm":"20140713.130102334.pc_all."}&request_id=167772676616800188575949&task-code-2~all~first_rank_ecpm_v1~rank_v31_ecpm-1-125656962-0-null-null.142^v73^insert_down1,201^v4^add_ask,239^v2^insert_chatgpt&utm_term=shell中实现backspace

  https://en.wikipedia.org/wiki/BIOS_interrupt_call#Interrupt_table

Booting Pintos

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

As the screenshot shows, the first booting is in QEMU and the second in Bochs.



Debugging

QUESTIONS: BIOS

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

`ljmp $0x3630,$0xf000e05b`. As shown in the screenshot.

```
(gdb) debugpintos
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
(gdb)
```

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

The PA is `0xfff0`.

As we know, `PA = VA - PHYS_BASE`, and `PHYS_BASE` is a multiple of `0x10000000`. As the screenshot shows, VA of the first instruction is `0xf000fff0`. So we can easily induce that `PHYS_BASE = 0xf0000000, PA = 0xfff0`

Ref:

- https://pkuflyingpig.gitbook.io/pintos/appendix/reference-guide/loading#physical-memory-map
- https://pkuflyingpig.gitbook.io/pintos/appendix/reference-guide/virtual-addresses#work-with-mapping-kernel-vm-one-to-one-to-pm

QUESTIONS: BOOTLOADER

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

First, the bootloader (beginning at `0x7c00`) checks all the sectors on different disks one by one.

```
 46   00007c1e <read_mbr>:
 47 > read_mbr: ···
 77 > 00007c47 <check_partition>: ···
 79 > check_partition: ···
103   00007c5e <next_partition>:
104
105 > next_partition: ···
114 > 00007c69 <next_drive>: ···
117 > next_drive: ···
124   00007c6d <no_boot_partition>:
```

Relevant functions.

When checking a specific sector, it uses `int 0x13` to call Interrupt 0x13 (Line 346, `0x7d30`). The number in register %ah is 0x42 (Line 342, `0x7d2c`). Looking up the table we know that the BIOS interrupt is "***Extended Read Sectors***".

```
322 read_sector:
323     pusha
324     7d1f:   60                      pusha
325     sub %ax, %ax
326     7d20:   29 c0                   sub    %eax,%eax
327     push %ax            # LBA sector number [48:63]
328     7d22:   50                      push   %eax
329     push %ax            # LBA sector number [32:47]
330     7d23:   50                      push   %eax
331     push %ebx           # LBA sector number [0:31]
332     7d24:   66 53                   push   %bx
333     push %es            # Buffer segment
334     7d26:   06                      push   %es
335     push %ax            # Buffer offset (always 0)
336     7d27:   50                      push   %eax
337     push $1             # Number of sectors to read
338     7d28:   6a 01                   push   $0x1
339     push $16            # Packet size
340     7d2a:   6a 10                   push   $0x10
341     mov $0x42, %ah          # Extended read
342     7d2c:   b4 42                   mov    $0x42,%ah
343     mov %sp, %si            # DS:SI -> packet
344     7d2e:   89 e6                   mov    %esp,%esi
345     int $0x13           # Error code in CF
346     7d30:   cd 13                   int    $0x13
347     popa                # Pop 16 bytes, preserve flags
348     7d32:   61                      popa
```

13h	0Ch	Seek to specified track
	0Dh	Reset Fixed Disk Controller
	15h	Get Drive Type
	16h	Get Floppy Drive Media Change Status
	17h	Set Disk Type
	18h	Set Floppy Drive Media Type
	41h	Extended Disk Drive (EDD) Installation Check
	42h	Extended Read Sectors
	43h	Extended Write Sectors
	44h	Extended Verify Sectors

https://en.wikipedia.org/wiki/BIOS_interrupt_call#Interrupt_table

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

The bootloader checks if a partition meets three criteria:

- An used partition?
- A Pintos kernel partition?
- A bootable partition?

If yes, this is the bootable Pintos kernel. And we jump to `load_kernel` to load the kernel.

```
79  check_partition:
80      # Is it an unused partition?
81      cmpl $0, %es:(%si)
82      7c47:   26 66 83 3c 00 74       cmpw   $0x74,%es:(%eax,%eax,1)
83      je next_partition
84      7c4d:   10 e8                   adc    %ch,%al
85
86      # Print [1-4].
87      call putc
88      7c4f:   ae                      scas   %es:(%edi),%al
89      7c50:   00 26                   add    %ah,(%esi)
90
91      # Is it a Pintos kernel partition?
92      cmpb $0x20, %es:4(%si)
93      7c52:   80 7c 04 20 75          cmpb   $0x75,0x20(%esp,%eax,1)
94      jne next_partition
95      7c57:   06                      push   %es
96
97      # Is it a bootable partition?
98      cmpb $0x80, %es:(%si)
99      7c58:   26 80 3c 80 74          cmpb   $0x74,%es:(%eax,%eax,4)
100     je load_kernel
101     7c5d:   20                      .byte 0x20
```

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

If bootloader could not find the kernel, it calls *interrupt 18h* by the instruction `INT 0x18`. This interrupt outputs an error message telling BIOS that the bootloader fails to find the kernel.

```
no_such_drive:
no_boot_partition:
    # Didn't find a Pintos kernel partition anywhere, give up.
    call puts
    7c6d:   e8 77 00 0d 4e          call   4e0d7ce9 <__bss_start+0x4e0cfee9>
    7c72:   6f                      outsl  %ds:(%esi),(%dx)
    7c73:   74 20                   je     7c95 <load_kernel+0x17>
    7c75:   66 6f                   outsw  %ds:(%esi),(%dx)
    7c77:   75 6e                   jne    7ce7 <puts>
    7c79:   64                      fs
    7c7a:   0d                      .byte 0xd
    7c7b:   00 cd                   add    %cl,%ch
    .string "\rNot found\r"

    # Notify BIOS that boot failed.  See [IntrList].
    int $0x18
    7c7d:   18                      .byte 0x18
```

18h	Execute Cassette BASIC: On IBM machines up to the early PS/2 line, this interrupt would start the ROM Cassette BASIC. Clones did not have this feature and different machines/BIOSes would perform a variety of different actions if INT 18h was executed, most commonly an error message stating that no bootable disk was present. Modern machines would attempt to boot from a network through this interrupt. On modern machines this interrupt will be treated by the BIOS as a signal from the bootloader that it failed to complete its task. The BIOS can then take appropriate next steps.[3]

https://en.wikipedia.org/wiki/BIOS_interrupt_call#Interrupt_table

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

`ljmp *start` (Line 225), the bootloader jumps to the start point of the kernel.
The bootloader saves the start address in position "*start*" (which is initially part of the loader's code).

```
212     mov $0x2000, %ax
213     7cbf:   b8 00 20 8e c0          mov    $0xc08e2000,%eax
214     mov %ax, %es
215     mov %es:0x18, %dx
216     7cc4:   26 8b 16                mov    %es:(%esi),%edx
217     7cc7:   18 00                   sbb    %al,(%eax)
218     mov %dx, start
219     7cc9:   89 16                   mov    %edx,(%esi)
220     7ccb:   d7                      xlat   %ds:(%ebx)
221     7ccc:   7c c7                   jl     7c95 <load_kernel+0x17>
222     movw $0x2000, start + 2
223     7cce:   06                      push   %es
224     7ccf:   d9 7c 00 20             fnstcw 0x20(%eax,%eax,1)
225     ljmp *start
226     7cd3:   ff 2e                   ljmp   *(%esi)
227     7cd5:   d7                      xlat   %ds:(%ebx)
228     7cd6:   7c                      .byte 0x7c
```

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?

The answer is 0.
Use gdb to set a breakpoint in `pintos_init`. And use `print` to print the value of expression.

```
(gdb) b pintos_init
Breakpoint 1 at 0xc00202b6: file ../../threads/init.c, line 82.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xc00202b6 <pintos_init>:    push   %ebp

Breakpoint 1, pintos_init () at ../../threads/init.c:82
(gdb) p init_page_dir[pd_no(ptov(0))]
=> 0xc000efef:  int3
=> 0xc000efef:  int3
$1 = 0
(gdb)
```

▼ to delete

`ptov(0)` transfers PA 0 to VA, which is `VA = PA + PHYS_BASE = 0xf0000000` . `pd_no = VA>>PDSHIFT = VA>>(10+12) = d`

```
static inline void *ptov (uintptr_t paddr)
{
  ASSERT ((void *) paddr < PHYS_BASE);
  return (void *) (paddr + PHYS_BASE);
} // transfer PA to VA

static inline uintptr_t pd_no (const void *va) {
  return (uintptr_t) va >> PDSHIFT;
} // Obtains page directory index from a virtual address
```

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

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

Set a breakpoint in `palloc_get_page` and run until the breakpoint.
Issue `bt` to see the call stack, which includes `start()`, `pintos_init()`, `paging_init()` and current `palloc_get_page()`.

```
Breakpoint 1, palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at ../../threads/palloc.c
(gdb) bt
#0  palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at ../../threads/palloc.c:112
#1  0xc0020522 in paging_init () at ../../threads/init.c:219
#2  0xc002031b in pintos_init () at ../../threads/init.c:104
#3  0xc002013d in start () at ../../threads/start.S:180
```

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

Issue `finish` to run until the function finishes. The return value is `0xc0101000`, which is a void pointer.

```
(gdb) finish
Run till exit from #0  palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at ../../threads/palloc.c:112
=> 0xc0020522 <paging_init+17>: add    $0x10,%esp
0xc0020522 in paging_init () at ../../threads/init.c:219
Value returned is $1 = (void *) 0xc0101000
```

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

Use gdb `print` , and we see the value is **0**.

```
(gdb) p init_page_dir[pd_no(ptov(0))]
=> 0xc000ef7f:  int3
=> 0xc000ef7f:  int3
$2 = 0
```

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

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

After the breakpoint is set, issue `c` three times to get to the thrid calling of `palloc_get_page()`.
Issue `bt` to see the call stack, which includes `start()`, `pintos_init()`, `thread_start()` , `thread_create()` and current `palloc_get_page()`.

```
(gdb) c
Continuing.
=> 0xc002328c <palloc_get_page>:        push   %ebp

Breakpoint 1, palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at ../../threads/palloc.c:112
(gdb) c
Continuing.
=> 0xc002328c <palloc_get_page>:        push   %ebp

Breakpoint 1, palloc_get_page (flags=PAL_ZERO) at ../../threads/palloc.c:112
(gdb) bt
#0  palloc_get_page (flags=PAL_ZERO) at ../../threads/palloc.c:112
#1  0xc0020bf9 in thread_create (name=0xc002ea51 "idle", priority=0, function=0xc0021028 <idle>, aux=0xc000efac) at ../../threads/thread.c:178
#2  0xc0020aee in thread_start () at ../../threads/thread.c:111
#3  0xc0020334 in pintos_init () at ../../threads/init.c:123
#4  0xc002013d in start () at ../../threads/start.S:180
```

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

Issue `fin`, and we see the return value is `0xc01030000`, which is a void pointer.

```
(gdb) fin
Run till exit from #0  palloc_get_page (flags=PAL_ZERO) at ../../threads/palloc.c:112
=> 0xc0020bf9 <thread_create+55>:       add    $0x10,%esp
0xc0020bf9 in thread_create (name=0xc002ea51 "idle", priority=0, function=0xc0021028 <idle>, aux=0xc000efac) at ../../threads/thread.c:178
Value returned is $3 = (void *) 0xc0103000
```

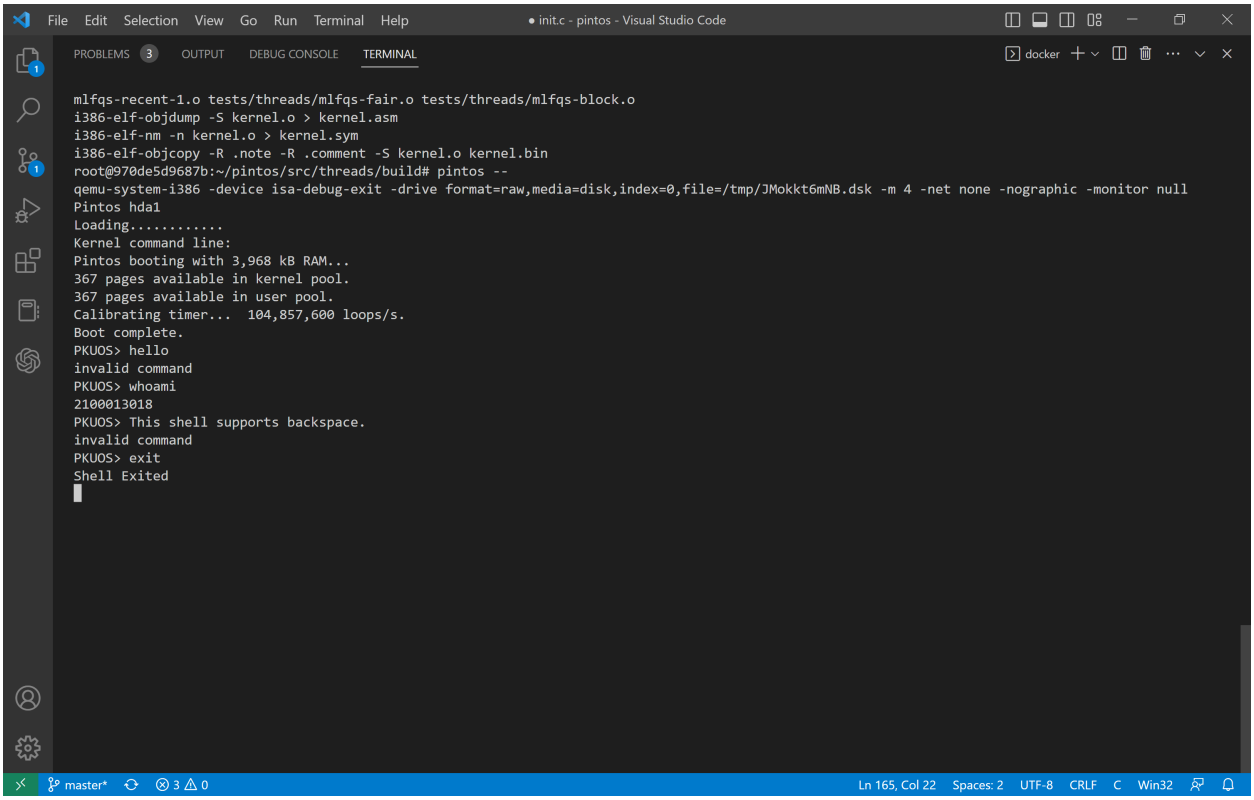
> B9.3 what is the value of expression init_page_dir[pd_no(ptov(0))] in hexadecimal format?

Use gdb print to get the value, which is `1056807` in decimal format.
Use p/x to get the hexadecimal value `0x102027`.

```
(gdb) p init_page_dir[pd_no(ptov(0))]
=> 0xc000ef3f:  int3
=> 0xc000ef3f:  int3
$4 = 1056807
(gdb) p/x init_page_dir[pd_no(ptov(0))]
=> 0xc000ef3f:  int3
=> 0xc000ef3f:  int3
$5 = 0x102027
```

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.)



C2: Explain how you read and write to the console for the kernel monitor.

I use an infinite loop to print `"PKUOS> "` and get command line input. If a new line is detected, we enter the next loop.

```
144     while(print_prompt){
145       printf("%s", prompt);
146       int idx=0;
147       memset(cmdline, 0, MAXLINE);
148
149       // get cmdline
150 >     while(1){···
172
173       // parse cmdline
174 >     if(strcmp(cmdline, "whoami") == 0){···
177 >     else if(strcmp(cmdline, "exit") == 0){···
180 >     else{···
183     }
```

For each line, I write a `while` loop to get the chars one by one and:

- print it if printable
- delete a char if it is a backspace
- overlook it if the length of a line has exceeded `MAXLINE` (a macro of max line length)
- break current loop and begin a new line if it is a `\r` (`\n` results in error)

```
149     // get cmdline
150     while(1){
151       char c = input_getc();
152       // newline dealing
153       if(c == '\r'){ // not \n
154         printf("\n");
155         cmdline[idx++] = '\0';
156         break;
157       }
158       // backspace dealing. Ref:https://blog.csdn.net/u011011827/article/details/125656962
159       else if(c == 127){···
167       if(idx >= MAXLINE-1) continue; // do nothing
168       if(c >= 32) // printable
169         printf("%c", c);
170       cmdline[idx++] = c;
171     }
```

Note: As for the backspace, I refer to the article below.

```
158       // backspace dealing. Ref:https://blog.csdn
159       else if(c == 127){
160         if(idx>0){
161           cmdline[idx--]='\0';
162           // printf("\b"); // Errors Occur
163           printf("\b \b");
164         }
165         continue;
166       }
```

https://blog.csdn.net/u011011827/article/details/125656962?ops_request_misc={"request_id":"167772676616800188575949","scm":"20140713.130102334.pc_all."}&request_id=167772676616800188575949&biz_task-code-2~all~first_rank_ecpm_v1~rank_v31_ecpm-1-125656962-0-null-null.142^v73^insert_down1,201^v4^add_ask,239^v2^insert_chatgpt&utm_term=shell中实现backspace

(Note: We wrote a similar shell in ICS Shelllab, so I refer to my own code (ˋ • ω • ´)y)