Project 0: Getting Real

陈震雄

武汉大学

2025 年 3 月 15 日

Table of Contents

1. Task 1: Booting Pintos
2. Task 2: Debugging
3. Task 3: Kernel Monitor

Table of Contents

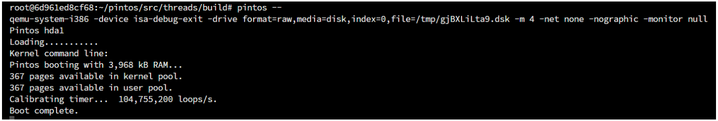
1 Task 1: Booting Pintos

2 Task 2: Debugging

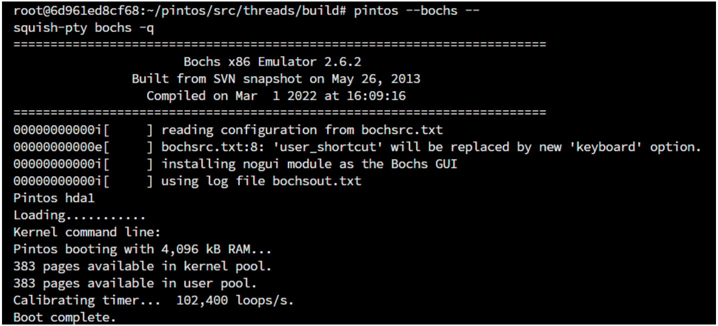
3 Task 3: Kernel Monitor

Exercise 1.1

- Take screenshots of the successful booting of Pintos in QEMU and Bochs.



```
root@6d961ed8cf68:~/pintos/src/threads/build# pintos --
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,file=/tmp/gjBXLiLta9.dsk -m 4 -net none -nographic -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...  104,755,200 loops/s.
Boot complete.
```



```
root@6d961ed8cf68:~/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 Mar  1 2022 at 16:09:16
========================================================================
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.
```

Table of Contents

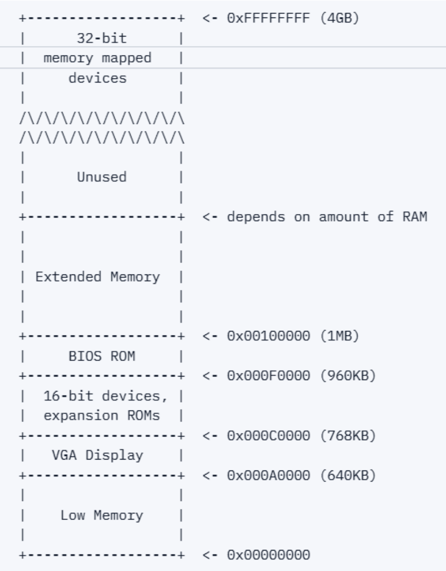
1. Task 1: Booting Pintos
2. Task 2: Debugging
3. Task 3: Kernel Monitor

Exercise 2.1

- What is the first instruction that gets executed?

```
(gdb) debugpintos
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
```

Exercise 2.1



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\
/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

Exercise 2.2

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

```
(gdb) break *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  sub    %eax,%eax
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8i $pc
=> 0x7c00:      sub    %eax,%eax
   0x7c02:      mov    %eax,%ds
   0x7c04:      mov    %eax,%ss
   0x7c06:      mov    $0xf000,%sp
   0x7c0a:      add    %al,(%eax)
   0x7c0c:      sub    %edx,%edx
   0x7c0e:      mov    $0xe3,%al
   0x7c10:      int    $0x14
   ...
   call read_sector
```

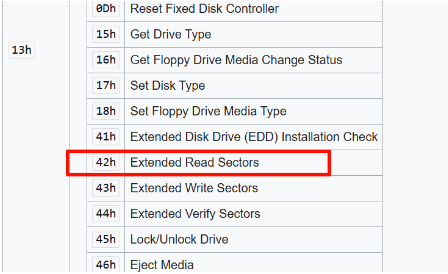
Exercise 2.2

loader.S

```
read_sector:
    pusha
    sub %ax, %ax
    push %ax            # LBA sector number [48:63]
    push %ax            # LBA sector number [32:47]
    push %ebx           # LBA sector number [0:31]
    push %es            # Buffer segment
    push %ax            # Buffer offset (always 0)
    push $1             # Number of sectors to read
    push $16            # Packet size
    mov $0x42, %ah     # Extended read
    mov %sp, %si       # DS:SI -> packet
    int $0x13          # Error code in CF
    popa                # Pop 16 bytes, preserve flags
```

Exercise 2.2

Reading hard disk sectors requires the use of the functions provided by BIOS. Specifically, as mentioned in the title, it triggers a BIOS interrupt. The instruction is located in line 242 (red box in the figure). A complete interrupt table is available for query under the BIOS interrupt call entry on Wikipedia:



13h		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
		45h	Lock/Unlock Drive
		46h	Eject Media

Exercise 2.2

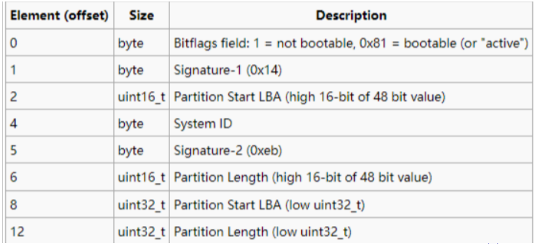
- How does the bootloader decide whether it successfully finds the Pintos kernel?

Exercise 2.2

```
    # Check for MBR signature--if not present, it's not a
    # partitioned hard disk.
    cmpw $0xaa55, %es:510
    jne next_drive

    mov $446, %si   # Offset of partition table entry 1.
    mov $'1', %al
check_partition:
    # Is it an unused partition?
    cmpl $0, %es:(%si)
    je next_partition
    # Print [1-4].
    call putc

    # Is it a Pintos kernel partition?
    cmpb $0x20, %es:4(%si)
    jne next_partition

    # Is it a bootable partition?
    cmpb $0x80, %es:(%si)
    je load_kernel
```

Exercise 2.2

```
next_partition:
 # No match for this partition, go on to the next one.
    add $16, %si   # Offset to next partition table entry.
    inc %al
    cmp $510, %si
    jb check_partition
 next_drive:
 # No match on this drive, go on to the next one.
    inc %dl
    jnc read_mbr
```



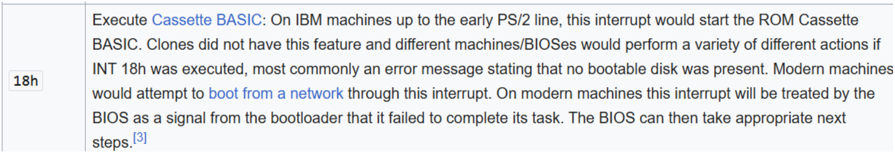
Element (offset)	Size	Description
0	byte	Bitflags field: 1 = not bootable, 0x81 = bootable (or "active")
1	byte	Signature-1 (0x14)
2	uint16_t	Partition Start LBA (high 16-bit of 48 bit value)
4	byte	System ID
5	byte	Signature-2 (0xeb)
6	uint16_t	Partition Length (high 16-bit of 48 bit value)
8	uint32_t	Partition Start LBA (low uint32_t)
12	uint32_t	Partition Length (low uint32_t)

Exercise 2.2

- What happens when the bootloader could not find the Pintos kernel?

```
no_such_drive:
no_boot_partition:
 # Didn't find a Pintos kernel partition anywhere, give up.
 call puts
 .string "\rNot found\r"

 # Notify BIOS that boot failed.  See [IntrList].
 int $0x18
```

Exercise 2.2



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]

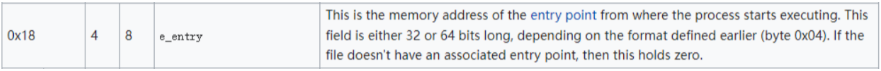
Exercise 2.2

- At what point and how exactly does the bootloader transfer control to the Pintos kernel?

Exercise 2.2

loader.S

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



0x18	4	8	e_entry	This is the memory address of the entry point from where the process starts executing. This field is either 32 or 64 bits long, depending on the format defined earlier (byte 0x04). If the file doesn't have an associated entry point, then this holds zero.

Exercise 2.3

- At the entry of pintos_init(), what is** **the value of the expression init_page_dir[pd_no(ptov(0))] in hexadecimal format?

Exercise 2.3

```
(gdb) b pintos_init
Breakpoint 1 at 0xc00202b6: file ../../threads/init.c, line 78.
(gdb) continue
Continuing.
The target architecture is assumed to be i386
=> 0xc00202b6 <pintos_init>:    push   %ebp

Breakpoint 1, pintos_init () at ../../threads/init.c:78
(gdb) p init_page_dir[pd_no(ptov(0))]
=> 0xc000efef:  int3
=> 0xc000efef:  int3
$1 = 0
```

Exercise 2.3

- When palloc_get_page() is called for the first time,
  - what does the call stack look like?
  - what is the return value in hexadecimal format?
  - what is the value of expression init_page_dir[pd_no(ptov(0))] in hexadecimal format?

Exercise 2.3

```
(gdb) b palloc_get_page
Breakpoint 2 at 0xc002311a: file ../../threads/palloc.c, line 113.
(gdb) continue
Continuing.
=> 0xc002311a <palloc_get_page+6>:      sub    $0x8,%esp
Breakpoint 2, palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO))
at ../../threads/palloc.c:113
(gdb) bt
#0  palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at
../../threads/palloc.c:113
#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
(gdb) fin
Run till exit from #0  palloc_get_page (flags=(PAL_ASSERT | PAL_ZERO)) at
../../threads/palloc.c:113
=> 0xc00203aa <paging_init+17>: add    $0x10,%esp
0xc00203aa in paging_init () at ../../threads/init.c:168
Value returned is $2 = (void *) 0xc0101000
(gdb) p/x init_page_dir[pd_no(ptov(0))]
=> 0xc000ef8f:  int3
=> 0xc000ef8f:  int3
$3 = 0x0
```

Exercise 2.3

- When palloc_get_page() is called for the third time,
  - what does the call stack look like?
  - what is the return value in hexadecimal format?
  - what is the value of expression init_page_dir[pd_no(ptov(0))] in hexadecimal format?

Exercise 2.3

```
(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
(gdb) fin
Run till exit from #0  palloc_get_page (flags=PAL_ZERO) at
../../threads/palloc.c:113
=> 0xc0020a81 <thread_create+55>:       add    $0x10,%esp
0xc0020a81 in thread_create (name=0xc002e895 "idle", priority=0,
function=0xc0020eb0 <idle>, aux=0xc000efbc) at ../..
/threads/thread.c:178
Value returned is $4 = (void *) 0xc0103000
(gdb) p/x init_page_dir[pd_no(ptov(0))]
=> 0xc000ef4f:  int3
=> 0xc000ef4f:  int3
$5 = 0x102027
```

Table of Contents

1 Task 1: Booting Pintos

2 Task 2: Debugging

3 Task 3: Kernel Monitor

Exercise 3.1

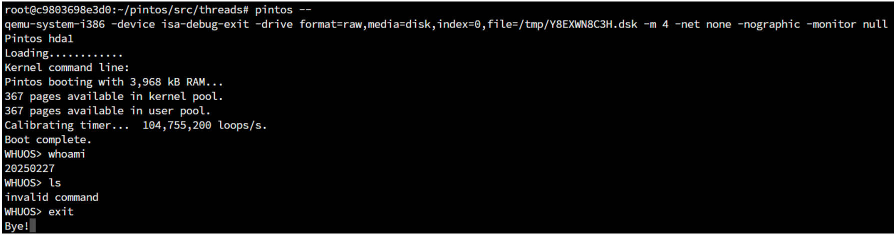
- Enhance threads/init.c to implement a tiny kernel monitor in Pintos.
  Requirments:
- It starts with a prompt WHUOS> and waits for user input.
- As the user types in a printable character, display the character.
- When a newline is entered, it parses the input and checks if it is whoami. If it is whoami, print your student id. Afterward, the monitor will print the command prompt WHUOS> again in the next line and repeat.
- If the user input is exit, the monitor will quit to allow the kernel to finish. For the other input, print invalid command. Handling special input such as backspace is not required.
- If you implement such an enhancement, mention this in your design document.

Exercise 3.1

```
size_t max_len = 10;
char* buf = (char*) malloc(max_len);
while(1)
{
    printf("WHUOS> ");
    memset(buf,'\0',max_len);
    size_t index = 0;
    while(1)
    {
    char c = input_getc();
    if (c == 13)
    {
        printf("\n");
        break;
    }
    if (c == 127)
    {
        if (index > 0) {
            buf[--index] = '\0';
            printf("\b \b");
        }
```

```
      continue;
  }
  if (index >= max_len) continue;
  buf[index++] = c;
  if (c > 31 && c < 127)
  {
      printf("%c", c);
  }
  }
  if (!strcmp(buf, "whoami"))
  {
      printf("20250227\n");
     continue;
    }
    if (!strcmp(buf, "exit"))
      break;
    printf("invalid command\n");
  }
  free(buf);
  printf("Bye!");
}
```

Exercise 3.1



```
root@c9803698e3d0:~/pintos/src/threads# pintos --
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,file=/tmp/Y8EXWN8C3H.dsk -m 4 -net none -nographic -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...  104,755,200 loops/s.
Boot complete.
WHUOS> whoami
20250227
WHUOS> ls
invalid command
WHUOS> exit
Bye!
```

The End

Thank you!

Any questions?