OPERATING SYSTEM

EXPERIMENTAL REPORT I

BOOTING A PC AND MEMORY MANAGEMENT

Yinhao Li 1611303

Abstract

Through the study of Experiment 1, we learned how the operating system works when the computer starts and how the computer manages the memory.

In Chapter 1, we will discuss Computer Boot, Boot Loader and The Kernel in detail. We will start research on the X86 assembly instructions, BIOS execution, PC physical address space, how the Boot Loader loads the kernel, the connection address and load address, the use of virtual memory, the formatted output of the console, and the stack.

In Chapter 2, we will discuss physical page management, virtual memory, and kernel address space. Describe in detail the relationship between virtual addresses and linear and physical addresses, reference counts, page table management, permissions and fault isolation, initialization of linear portions of the kernel, and so on.

Key Words: Computer Boot Boot Loader The Kernel physical page management virtual memory kernel address space

CONTENTS

Chapter1 Booting a PC

1.1 Computer Boot

1.1.1 Simulating The X86

In this experiment, we chose QEMU to simulate a real computer. The use of QEMU is as follows.

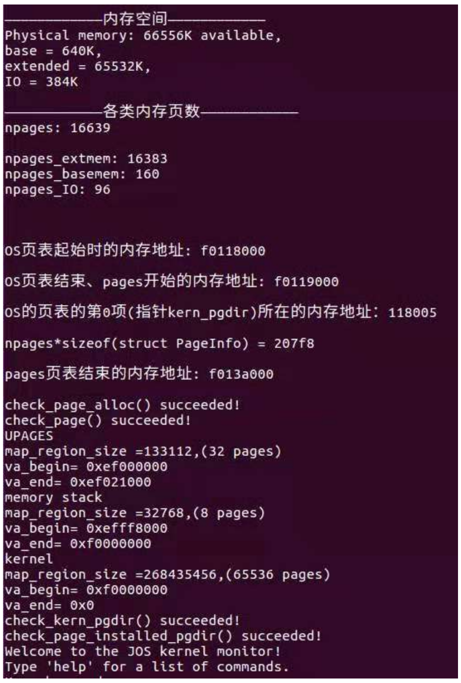


图 1.1: make qemu

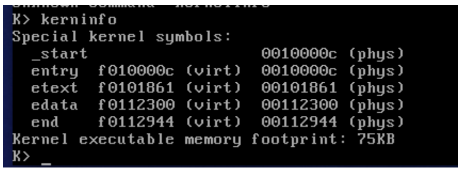


图 1.2: kerninfo

Through the above two instructions, we can see that the kernel information has been output in the console. It can be seen that QEMU can function as a simulation computer and the system can run normally.

1.1.2 The PC's Physical Address Space

The approximate physical distribution of the physical memory of the pc is as follows

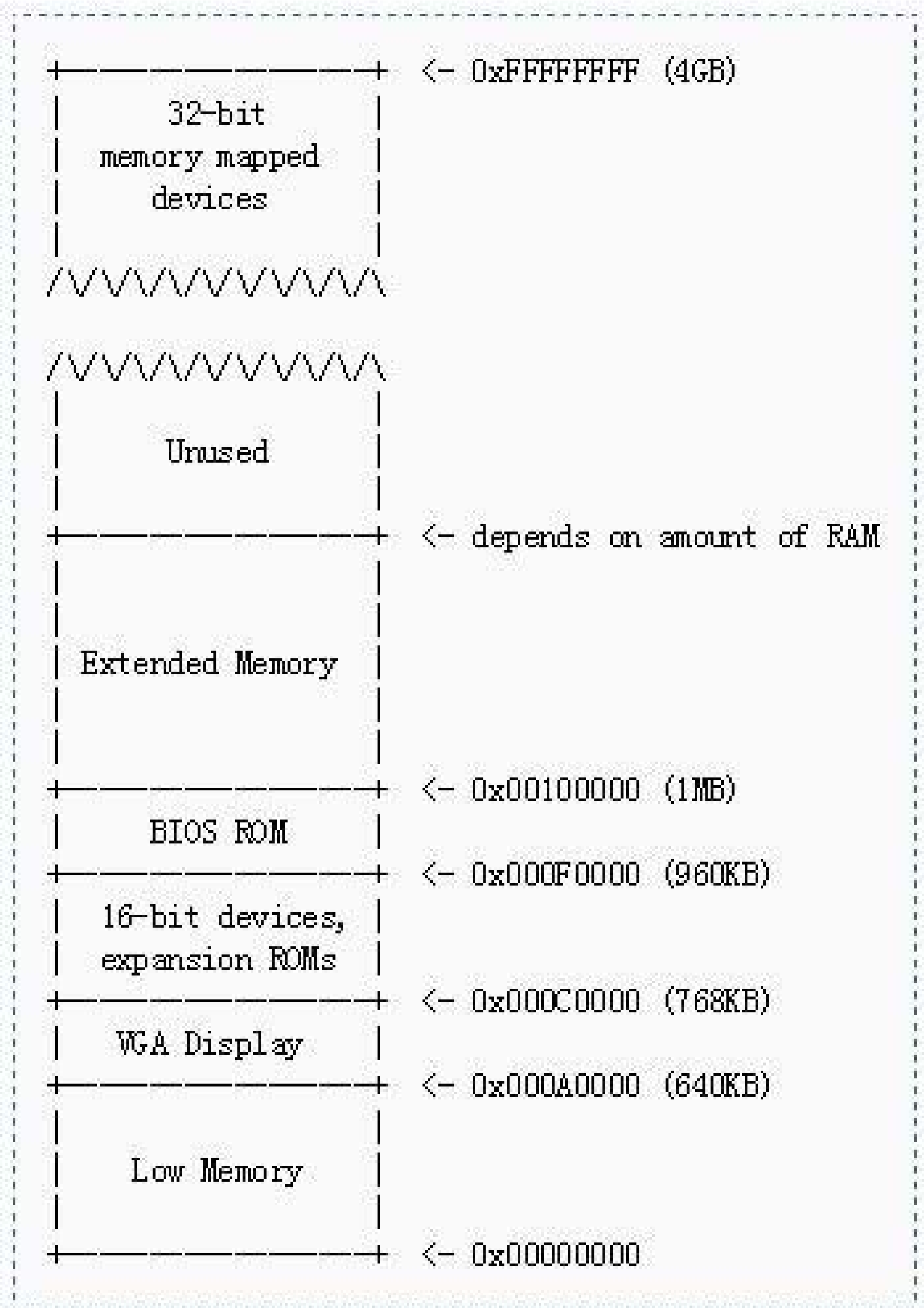
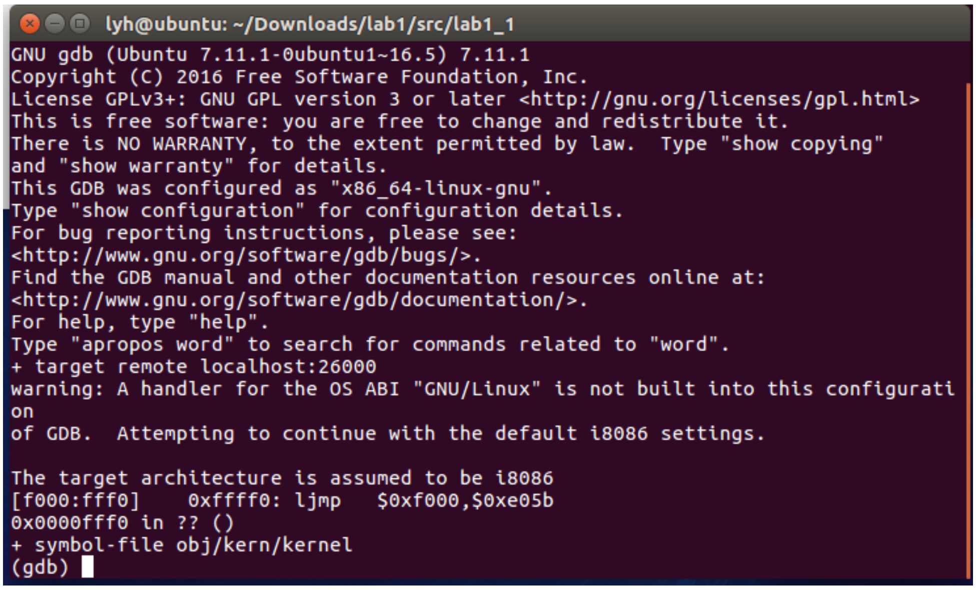


图 1.3: i 文件部分代码

The 384KB of physical address range 0x000A0000 0x000FFFFF is reserved for hardware devices such as VGA display buffers. The most important part of this part is as the basic input and output system (BIOS, Basic)Input/Output System) The 64KB area of 0x000F0000 0x000FFFFF. The BIOS handles the basic initialization tasks of the system, such as activating the display, checking memory, and so on. After completing these initialization tasks, the BIOS loads the operating system from a floppy disk, hard drive, optical drive, or network and transfers control to the operating system.

On the modern PC, the 0x000A0000 0x00100000 segment is left as a ”hole”, which divides the memory into two segments. The lower 640K is called ”low-segment memory”, and the remaining high-address portion is expanded memory. At the same time, in the highest part of the physical address space, above all physical RAM, some are also reserved by the BIOS for 32-bit PCI devices.

1.1.3 BIOS



```
lyh@ubuntu: ~/Downloads/lab1/src/lab1_1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

图 1.4: gdb

In the running content of gdb, we can see the first instruction that gdb runs (as shown below). In this instruction, we can see that pc starts from CS = 0xf000 and IP = 0xfff0. The execution of the first sentence is a JMP operation that jumps to CS = 0xf000 and IP = 0xe05b. Since the modern CPU is divided into real mode and protection mode, it runs in real mode at startup and runs in protected mode after startup. The BIOS is the software that runs when the PC first starts up, so it must work in real mode. So the real address of the jump is 0xf000«4+0xe05b = 0xfe05b.

```
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
```

图 1.5: The first instruction of the BIOS

1.2 Boot Loader

For PCs, floppy disks and hard disks can be divided into 512-byte areas called sectors. A sector is the smallest granularity of a disk operation. Each read or write operation must be one or more sectors. If a disk can be used to boot the operating system, the first sector of the disk is called the boot sector. The boot loader program described in this section is located in this boot sector. When the BIOS finds a floppy disk or hard disk that can be booted, it loads the 512-byte boot sector into the memory address 0x7c00 0x7dff. The boot loader must perform two main functions.

1. First, the boot loader will convert the processor from real mode to 32-bit protected mode, because only in this mode the software can access more than 1MB of content.

2. The boot loader can then access the IDE disk device registers directly from the disk by using x86 specific IO instructions.

For the boot loader, there is a file that is important, obj/boot/boot.asm. This file is a disassembled version of our real-run boot loader program. So we can compare it with its source code, boot.S and main.c.

1.2.1 Question I

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.

Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

1）At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
2）What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
3）Where is the first instruction of the kernel?

4）How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

The BIOS will copy the boot sector to the address 0x7c00, so the starting address of boot.S is 0x7c00.So we enter b *0x7c00 in the gdb window, then enter c, which means continue to run to the breakpoint, where we enter x/30i 0x7c00.This gdb instruction disassembles the instructions stored in 0x7c00 and the next 30 bytes of memory.

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/30i 0x7c00
=> 0x7c00:      cli
   0x7c01:      cld
   0x7c02:      xor    %ax,%ax
   0x7c04:      mov    %ax,%ds
   0x7c06:      mov    %ax,%es
   0x7c08:      mov    %ax,%ss
   0x7c0a:      in     $0x64,%al
   0x7c0c:      test   $0x2,%al
   0x7c0e:      jne    0x7c0a
   0x7c10:      mov    $0xd1,%al
   0x7c12:      out    %al,$0x64
   0x7c14:      in     $0x64,%al
   0x7c16:      test   $0x2,%al
   0x7c18:      jne    0x7c14
   0x7c1a:      mov    $0xdf,%al
   0x7c1c:      out    %al,$0x60
   0x7c1e:      lgdtw  0x7c64
   0x7c23:      mov    %cr0,%eax
   0x7c26:      or     $0x1,%eax
   0x7c2a:      mov    %eax,%cr0
   0x7c2d:      ljmp   $0x8,$0x7c32
   0x7c32:      mov    $0xd88e0010,%eax
   0x7c38:      mov    %ax,%es
---Type <return> to continue, or q <return> to quit---
```

图 1.6: Disassembly

We take it directly with boot.S and at obj/boot/boot.asm

```
obj/boot/boot.out:     file format elf32-i386


Disassembly of section .text:

00007c00 <start>:
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
    7c00:       fa                    cli
  cld                         # String operations increment
    7c01:       fc                    cld

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
    7c02:       31 c0                 xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    7c04:       8e d8                 mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    7c06:       8e c0                 mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    7c08:       8e d0                 mov    %eax,%ss

00007c0a <seta20.1>:
  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
    7c0a:       e4 64                 in     $0x64,%al
  testb   $0x2,%al
    7c0c:       a8 02                 test   $0x2,%al
  jnz     seta20.1
    7c0e:       75 fa                 jne    7c0a <seta20.1>

  movb    $0xd1,%al               # 0xd1 -> port 0x64
    7c10:       b0 d1                 mov    $0xd1,%al
  outb    %al,$0x64
    7c12:       e6 64                 out    %al,$0x64

00007c14 <seta20.2>:
```

图 1.7: boot.asm

```
boot.asm
#include <inc/mmu.h>

# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

图 1.8: boot.s

By comparing we can find that the three are not different in the instruction, but in the source code, we specify a lot of identifiers such as set20.1. Initially, these identifiers are converted to real physical addresses after being assembled into machine code. For example, set20.1 is converted to 0x7c0a, then this correspondence is listed in OBJ's /boot/boot.asm, but in the real case, in the first case, you can't see set20. 1 identifier, completely real physical address.

Then according to the title indication, we first traced to the bootmain function, we found that bootmain is at address 0x7c45, so we set the breakpoint there, and run here, then set the breakpoint at the readsect (0x7c7c) and jump here After using the si command to step through, found that into the waitdisk function (ox7c6a)

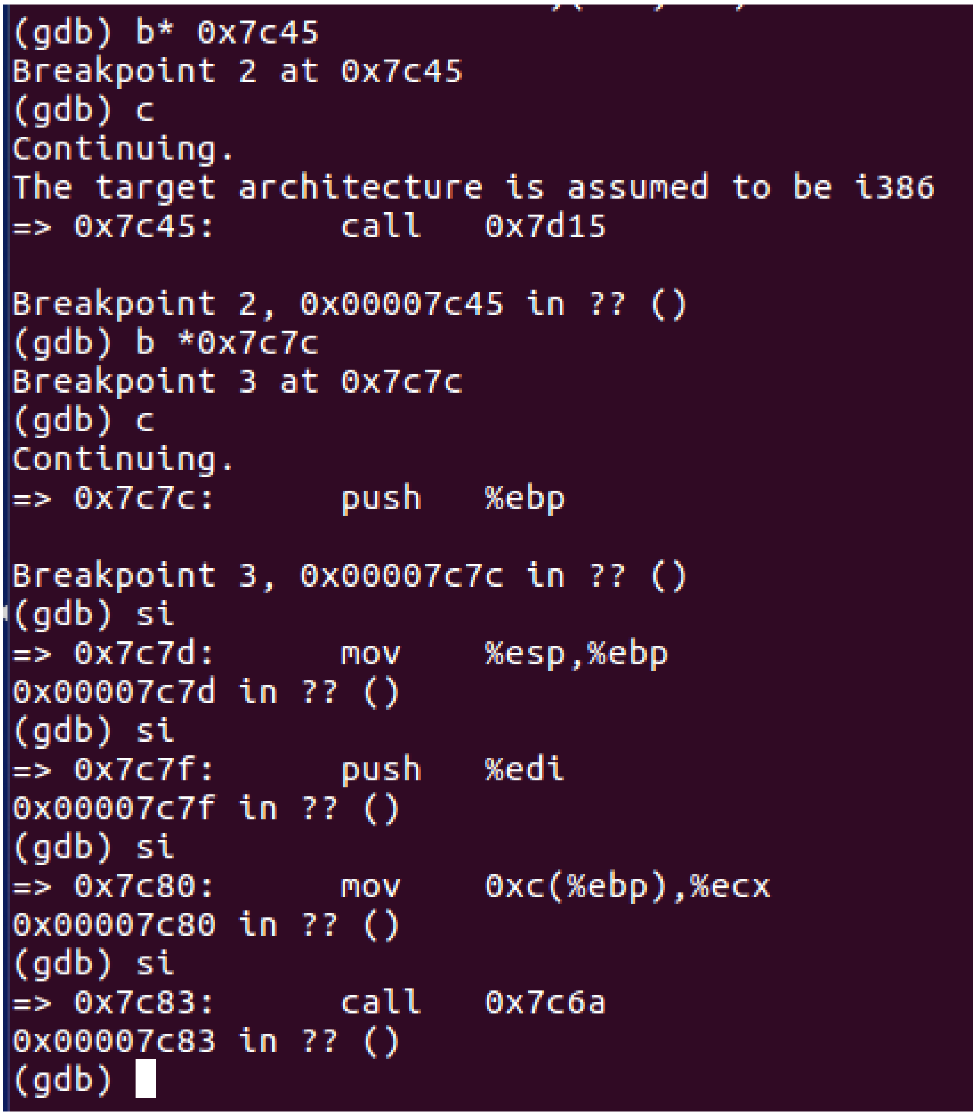


图 1.9: call bootmain

In boot.asm we can find the corresponding content of 0x7d6b. Jumping to here means ((void (*)(void)) (ELFHDR->e_entry))() the function is executed, where the meaning of the e_entry field of the ELF header is the first instruction of the executable. Virtual address. So the meaning of this sentence is to transfer control to the operating system kernel. Then we find the end of the for statement in the file and jump to this location (0x7d6b).

```
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
7d6b:       ff 15 18 00 01 00       call   *0x10018
}
```

图 1.10: end loop

```
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
7d6b:       ff 15 18 00 01 00       call   *0x10018
```

图 1.11: the instruction after loop

Answer:

1)

As we discussed earlier, When the PC starts up, the CPU runs in real mode (real mode), and when it enters the operating system kernel, it will run in protected mode (protected mode). When entering protection mode, the CPU will switch from 16 bits to 32 bits.

```
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
  7c2d:       ea                          .byte 0xea
  7c2e:       32 7c 08 00                 xor    0x0(%eax,%ecx,1),%bh
```

图 1.12: mode change

In the boot.S file, the computer first works in real mode, this time is the 16bit working mode. As can be seen from the above figure, when the "ljmp $PROT_MODE_CSEG, $protcseg" statement is run, the 32-bit working mode is officially entered. The root cause is that the CPU is working in protected mode at this time.

2)

```
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
7d6b:       ff 15 18 00 01 00       call   *0x10018
```

图 1.13: the instruction after loop

As shown in the figure above, the last statement executed by the boot loader is the last statement in the bootmain subroutine "((void (*)(void)) (ELFHDR->e_entry))(); ", that is, jump to the operating system The starting instruction of the kernel program. This first instruction is located in the /kern/entry.S file, the first sentence movw $0x1234, 0x472

3)

The first instruction is in the /kern/entry.S file.

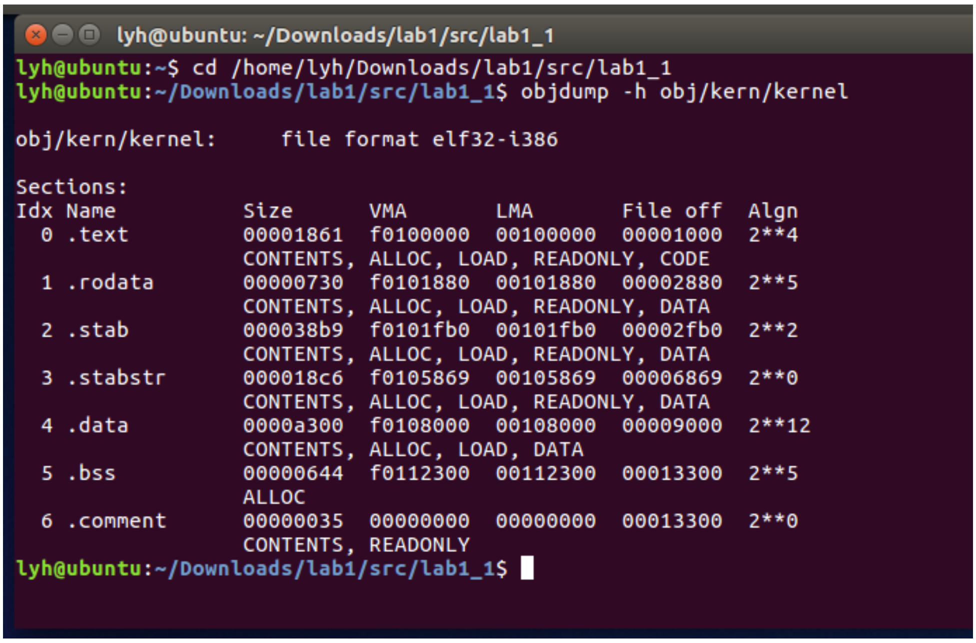
4)

First, how many segments are shared by the operating system, and how many sectors are in each segment are located in the Program Header Table in the operating system file. Each entry in this table corresponds to a segment of the operating system. And the content of each entry includes the size of the segment, the segment start address offset, and the like. So if we can find this table, we can determine how many sectors the kernel occupies by the information provided by the table entry. Then the information about where this table is stored is stored in the ELF header information of the operating system kernel image file.

1.2.2 Loading kernel

An executable ELF file consists of three main parts: a file header with loading information, followed by a program segment table, followed by several program segments. Each of these segments is a piece of continuous code or data. They are first loaded into memory when they are run. The job of the boot loader is to load them into memory.We can use the following instructions to examine the names, sizes, and addresses of all segments in the JOS kernel.

objdump -h obj/kern/kernel



```
lyh@ubuntu: ~/Downloads/lab1/src/lab1_1
lyh@ubuntu:~$ cd /home/lyh/Downloads/lab1/src/lab1_1
lyh@ubuntu:~/Downloads/lab1/src/lab1_1$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001861  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000730  f0101880  00101880  00002880  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000038b9  f0101fb0  00101fb0  00002fb0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018c6  f0105869  00105869  00006869  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000644  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      00000035  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
lyh@ubuntu:~/Downloads/lab1/src/lab1_1$
```

图 1.14: All segment information in the kernel

1.2.3 Connection address and load address

There are two more important fields in each segment, VMA (link address), LMA (load address). The load address represents the physical address of the segment after it is loaded into memory. The link address refers to the logical address to which this segment is expected to be stored.

Each ELF file has a Program Headers Table that indicates which parts of the ELF file are loaded into memory and the addresses that are loaded into memory. We get the information of the Program Headers Table of the kernel by entering the following instructions:

objdump -x obj/kern/kernel

```
                    CONTENTS, READONLY
lyh@ubuntu:~/Downloads/lab1/src/lab1_1$ objdump -x obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x0000712f memsz 0x0000712f flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000a300 memsz 0x0000a944 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001861  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000730  f0101880  00101880  00002880  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000038b9  f0101fb0  00101fb0  00002fb0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018c6  f0105869  00105869  00006869  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000644  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      00000035  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
f0100000 l    d  .text  00000000 .text
f0101880 l    d  .rodata        00000000 .rodata
f0101fb0 l    d  .stab  00000000 .stab
f0105869 l    d  .stabstr       00000000 .stabstr
f0108000 l    d  .data  00000000 .data
f0112300 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    df *ABS*  00000000 obj/kern/entry.o
f010002f l       .text  00000000 relocated
f010003e l       .text  00000000 spin
00000000 l    df *ABS*  00000000 entrypgdir.c
00000000 l    df *ABS*  00000000 init.c
00000000 l    df *ABS*  00000000 console.c
f0100177 l     F .text  0000001f serial_proc_data
f0100196 l     F .text  00000043 cons_intr
f0112320 l     O .bss   00000208 cons
```

图 1.15: Kernel's Program Headers Table information

1.3 The Kernel

1.3.1 Question

问题：

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?
2. Explain the following from `console.c`:

```
if (crt_pos >= CRT_SIZE) {
        int i;
        memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
                crt_buf[i] = 0x0700 | ' ';
        crt_pos -= CRT_COLS;
}
```

Answer:

1)

Console.c defines how to display a character to the console, which is on top of our display, which includes many operations on the IO port.What is defined in printf.c is the top-level formatted output subroutine we will use in programming, such as printf, sprintf, and so on. The function exported by console.c which is used by printf.c is cputchar(), That function prints a character in the parellel port and in the display.

Specific call relationship: cprintf -> vcprintf -> putch -> cputchar. Kernel's cprintf() function calls vprintfmt() (from lib/printfmt.c) to Actually print in the console, vprintfmt() does the needed formatting and Then call a function passed to it to actually print in the display.

```
#include <inc/types.h>
#include <inc/stdio.h>
#include <inc/stdarg.h>


static void
putch(int ch, int *cnt)
{
        cputchar(ch);
        *cnt++;
}
```

图 1.16: Function call

2)

Crt_buf: This is a character array buffer that holds the characters to be displayed on the screen.

Crt_pos: This indicates the position of the current last character displayed on the screen.

When crt_pos >= CRT_SIZE, where CRT_SIZE = 80*25, since we know that the value range of crt_pos is 0 (80*25-1), if this condition is true, it means that the content output on the screen has exceeded one page. . So at this point you have to scroll the page up one line, that is, put the original line 1 79 on the current 0 78 line, and then replace the line 79 with a line of spaces (of course not all spaces, 0 characters) To display the character int c) you entered. So the memcpy operation is to copy the contents of lines 1 79 in the crt_buf character array to the position of lines 0 78. The next for loop is to turn the last line, line 79 into a space. Finally, you need to modify the value of crt_pos.

1.3.2 Use virtual memory

As we discussed earlier, the computer is divided into real mode and protected mode when it starts up.When running the boot loader, the link address (virtual address) and the load address (physical address) in the boot loader are the same. But when you enter the kernel, the two addresses are no longer the same. In the virtual address space, we put the operating system at the high address 0xf0100000, but in the actual memory we store the operating system in a low physical address space, such as 0x00100000. Then when the user program wants to access an instruction of an operating system kernel, the first is to give a high virtual address, and then the virtual address is mapped to a real physical address by a certain mechanism in the computer, thus solving the above problem. . Then such an organization is usually implemented through segment management and paging management.

1.3.3 Homework I

We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

Answer

To answer this question, we should first understand the three files \ kern \ printf.c, \ kern \ console.c, \ lib \ printfmt.c to understand the relationship between them. First of all, we should pay attention to the console.c file, the most important of which is the cputchar subroutine. We can find that this program is the IO control program of the high-level console. In addition, the implementation of cputchar is actually done by calling cons_putc. And the function of the cons_putc program is to output a character to the console.

```
// `High'-level console I/O.  Used by readline and cprintf.

void
cputchar(int c)
{
	cons_putc(c);
}
```

图 1.17: cputchar

```
// output a character to the console
static void
cons_putc(int c)
{
	serial_putc(c);
	lpt_putc(c);
	cga_putc(c);
}
```

图 1.18: cons_putc

Then, let's focus on the printfmt.c file. By commenting, we can see that the subroutine defined in this file is the key to the information we can use to directly output information to the screen during programming using the printf function.

console.c

```
// Stripped-down primitive printf-style formatting routines,
// used in common by printf, sprintf, fprintf, etc.
// This code is also used by both the kernel and user programs.

#include <inc/types.h>
#include <inc/stdio.h>
#include <inc/string.h>
#include <inc/stdarg.h>
#include <inc/error.h>
```

图 1.19: printfmt

Finally, let's take a look at the printf.c file.By commenting, we can see that the function of this file is to implement the simple implementation of cprintf console output for the kernel.

```
// Simple implementation of cprintf console output for the kernel,
// based on printfmt() and the kernel console's cputchar().

#include <inc/types.h>
#include <inc/stdio.h>
#include <inc/stdarg.h>


static void
putch(int ch, int *cnt)
{
	cputchar(ch);
	*cnt++;
}

int
vcprintf(const char *fmt, va_list ap)
{
	int cnt = 0;

	vprintfmt((void*)putch, &cnt, fmt, ap);
	return cnt;
}

int
cprintf(const char *fmt, ...)
{
	va_list ap;
	int cnt;

	va_start(ap, fmt);
	cnt = vcprintf(fmt, ap);
	va_end(ap);

	return cnt;
}
```

图 1.20: printf

In this question, we follow the case 'u' to write the code.

```
// (unsigned) octal
case 'o':
        // Replace this with your code.
        /*putch('X', putdat);
        putch('X', putdat);
        putch('X', putdat);
        break;*/

        num = getuint(&ap, lflag);
        base = 10;
        goto number;
// pointer
```

图 1.21: answer

After that we modify the monitor.c file and then run qemu on the terminal to test our results.

```
void
monitor(struct Trapframe *tf)
{
        char *buf;

        cprintf("Welcome to the JOS kernel monitor!\n");
        cprintf("Type 'help' for a list of commands.\n");
        int x = 20;
        cprintf("x %d,x %o\n",x,x);

        while (1) {
                buf = readline("K> ");
                if (buf != NULL)
                        if (runcmd(buf, tf) < 0)
                                break;
        }
}
```

图 1.22: Change monitor

```
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
x 20,x 24
K>
```

图 1.23: Result test

As can be seen from the results, the 20 we defined is under %d, and the %o is correctly changed to octal 24

1.3.4 Stack

Exercise

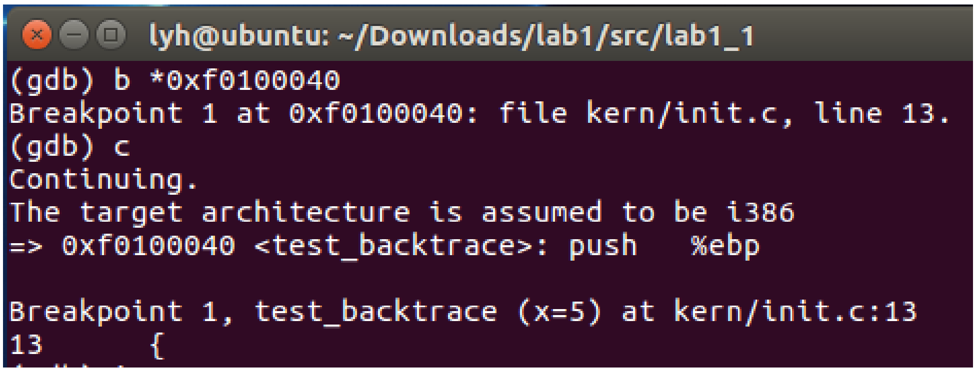
To become familiar with the C calling conventions on the x86,find the address of the test_backtrace function in obj/kern/kernel.asm,set a breakpoint there,and examine what happens each time it gets called after the kernel starts.How many 32-bits words does each recursive nesting level of test_backtrace push on the stack,and what are those words?

Answer

First, let's take a look at the source code of the test_backtrace function in kernel.asm

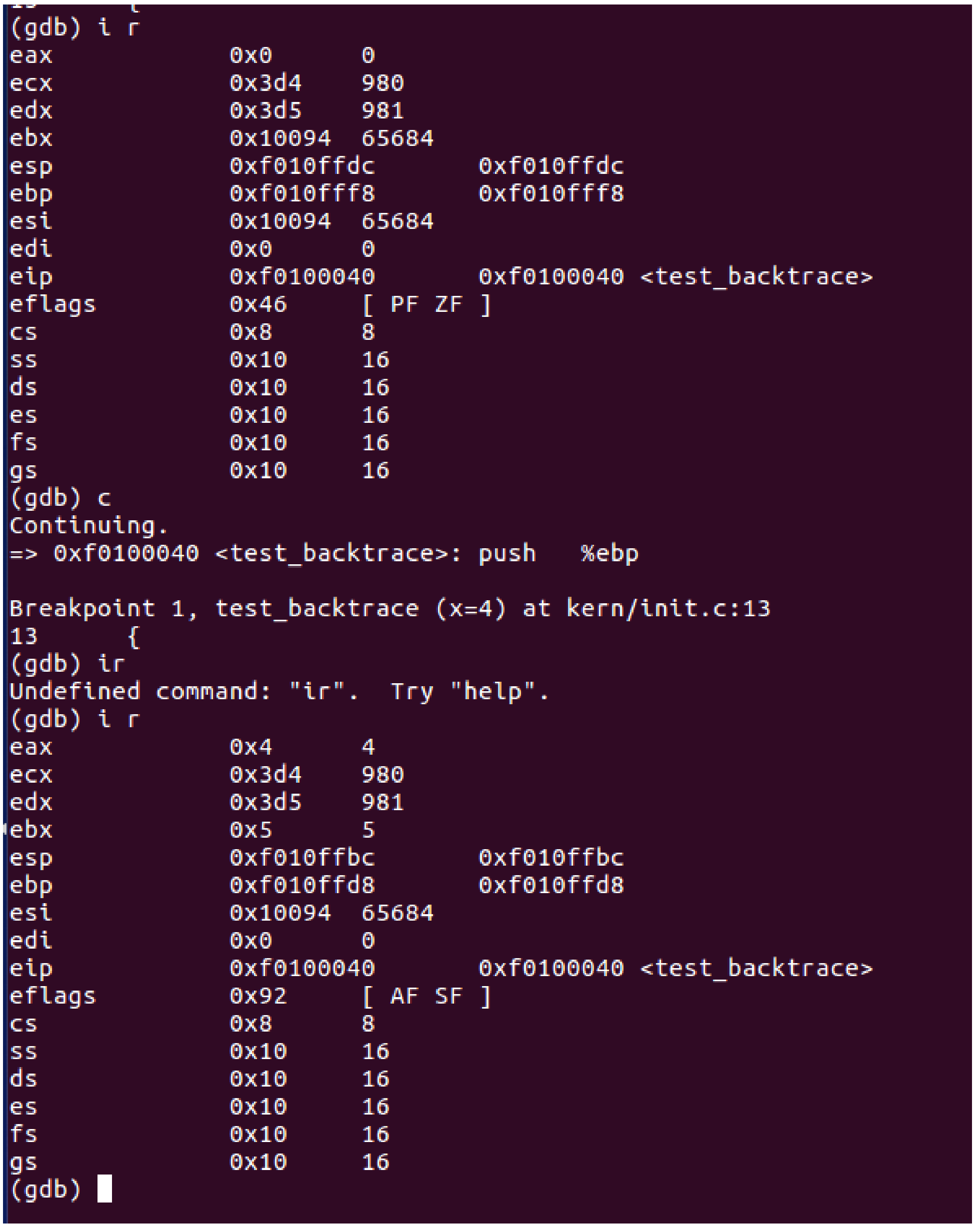
```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 0c                sub    $0xc,%esp
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       53                      push   %ebx
f010004b:       68 00 18 10 f0          push   $0xf0101800
f0100050:       e8 a6 08 00 00          call   f01008fb <cprintf>
        if (x > 0)
f0100055:       83 c4 10                add    $0x10,%esp
f0100058:       85 db                   test   %ebx,%ebx
f010005a:       7e 11                   jle    f010006d <test_backtrace+0x2d>
                test_backtrace(x-1);
f010005c:       83 ec 0c                sub    $0xc,%esp
f010005f:       8d 43 ff                lea    -0x1(%ebx),%eax
f0100062:       50                      push   %eax
f0100063:       e8 d8 ff ff ff          call   f0100040 <test_backtrace>
f0100068:       83 c4 10                add    $0x10,%esp
f010006b:       eb 11                   jmp    f010007e <test_backtrace+0x3e>
        else
                mon_backtrace(0, 0, 0);
f010006d:       83 ec 04                sub    $0x4,%esp
f0100070:       6a 00                   push   $0x0
f0100072:       6a 00                   push   $0x0
f0100074:       6a 00                   push   $0x0
f0100076:       e8 e5 06 00 00          call   f0100760 <mon_backtrace>
f010007b:       83 c4 10                add    $0x10,%esp
        cprintf("leaving test_backtrace %d\n", x);
f010007e:       83 ec 08                sub    $0x8,%esp
f0100081:       53                      push   %ebx
f0100082:       68 1c 18 10 f0          push   $0xf010181c
f0100087:       e8 6f 08 00 00          call   f01008fb <cprintf>
}
f010008c:       83 c4 10                add    $0x10,%esp
f010008f:       8b 5d fc                mov    -0x4(%ebp),%ebx
f0100092:       c9                      leave
f0100093:       c3                      ret

f0100094 <i386_init>:
```

From the code we can see the address of test_backtrace, so we set a breakpoint here, then jump to here, as shown below:



```
lyh@ubuntu: ~/Downloads/lab1/src/lab1_1
(gdb) b *0xf0100040
Breakpoint 1 at 0xf0100040: file kern/init.c, line 13.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
```

Then we use the i r instruction to view the register contents and repeat the above operation



```
(gdb) i r
eax            0x0      0
ecx            0x3d4    980
edx            0x3d5    981
ebx            0x10094  65684
esp            0xf010ffdc       0xf010ffdc
ebp            0xf010fff8       0xf010fff8
esi            0x10094  65684
edi            0x0      0
eip            0xf0100040       0xf0100040 <test_backtrace>
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13      {
(gdb) ir
Undefined command: "ir".  Try "help".
(gdb) i r
eax            0x4      4
ecx            0x3d4    980
edx            0x3d5    981
ebx            0x5      5
esp            0xf010ffbc       0xf010ffbc
ebp            0xf010ffd8       0xf010ffd8
esi            0x10094  65684
edi            0x0      0
eip            0xf0100040       0xf0100040 <test_backtrace>
eflags         0x92     [ AF SF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
(gdb)
```

By comparison, we can find that the value of ebp is reduced by 20 (hexadecimal), so we can judge every time it pushes 8 4-byte words.

According to the structure of the test_backtrace function and the change of the stack when the function is called, each time the function is called, the ebp, the return address, the saved ebx value, and the parameters of the next call function must be pushed onto the stack. There are also 4 reserved words.

1.3.5 Homework II

Question

You can do mon_backtrace() entirely in C.You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.The backtrace function should display a listing of function call frames in the following format:

Waring:

read_ebp

display format

Answer

We first call the read_ebp function to get the value of the current EBP register. We treat the entire call stack as an array, EBP[0] represents the ebp value of the previous function, and EBP [1] stores the function return address, EBP [2] What is stored later is the value of the input parameter.

Ebp of the previous function
ebx
Parameter 1
Parameter 2
Parameter 3
Parameter 4
Parameter 5

表 1.1: Stack structure diagram

The modified mon_backtrace function code is shown in the figure

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{

            uint32_t ebp=read_ebp();
            uint32_t *tmp_ebp=(uint32_t *)ebp;
            int i;
            while(tmp_ebp)
            {
                    cprintf("ebp:0x%x ",tmp_ebp);
                    cprintf("eip:0x%x ",tmp_ebp[1]);
                    cprintf("arg:");
                    for(i=2;i<7;i++)
                    {
                       cprintf("0x%x ",tmp_ebp[i]);
                    }
                    cprintf("\n");

           tmp_ebp=(uint32_t *)(*tmp_ebp);

           }


        return 0;
}
```

After analysis, we can know that when the function is called, the parameters are first pushed onto the stack, then the eip is pushed onto the stack, and finally the ebp is pushed onto the stack, and the new ebp points to the position of the old ebp. Therefore, the location pointed to by ebp stores the value of the previous function ebp, and the location of ebp+1 stores the value of eip. So we first print out tmp_ebp as ebp, tmp_ebp[1] as eip. Then we print out the values of 5 parameters according to the requirements of the topic. Finally, we take the value in the memory space pointed to by ebp and repeat the above process until the first function is called.

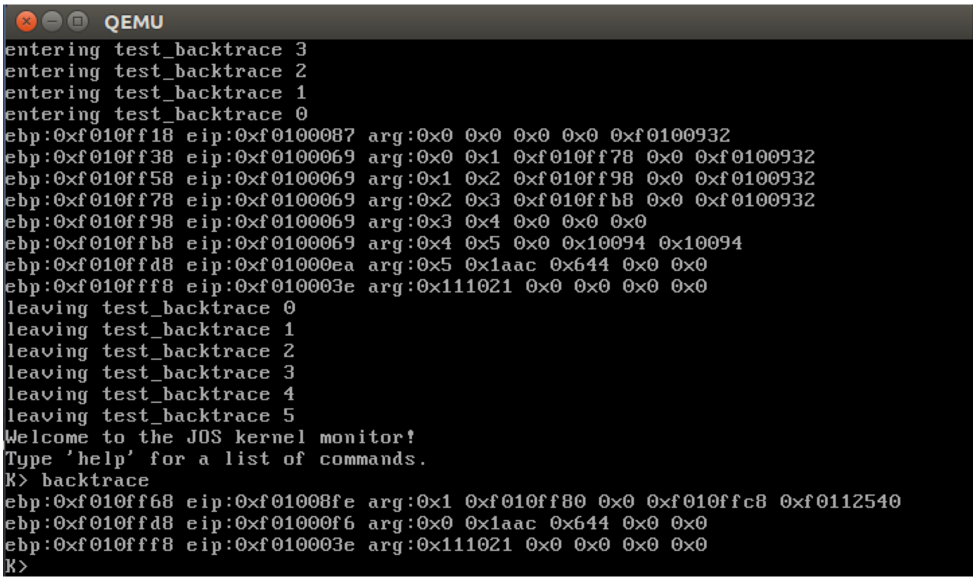
The result of the operation is as follows

```
QEMU
Booting from Hard Disk...
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
ebp:0xf010ff18 eip:0xf0100087 arg:0x0 0x0 0x0 0x0 0xf0100932
ebp:0xf010ff38 eip:0xf0100069 arg:0x0 0x1 0xf010ff78 0x0 0xf0100932
ebp:0xf010ff58 eip:0xf0100069 arg:0x1 0x2 0xf010ff98 0x0 0xf0100932
ebp:0xf010ff78 eip:0xf0100069 arg:0x2 0x3 0xf010ffb8 0x0 0xf0100932
ebp:0xf010ff98 eip:0xf0100069 arg:0x3 0x4 0x0 0x0 0x0
ebp:0xf010ffb8 eip:0xf0100069 arg:0x4 0x5 0x0 0x10094 0x10094
ebp:0xf010ffd8 eip:0xf01000ea arg:0x5 0x1aac 0x644 0x0 0x0
ebp:0xf010fff8 eip:0xf010003e arg:0x111021 0x0 0x0 0x0 0x0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Add backtrace to the command list

```
static struct Command commands[] = {
        { "help", "Display this list of commands", mon_help },
        { "kerninfo", "Display information about the kernel", mon_kerninfo },
        { "backtrace","Display stack backtrace",mon_backtrace},
};
```

The results are as follows:



1.3.6 Challenge homework

Question

Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip. In debuginfo_eip, where do ___STAB_* come from?

Answer

After reading the contents of stab.h and using the objdump -G kernel command, you can roughly know the fields of the stab table that store debugging information and their meanings, as follows

Uint32_t n_strx; index, can be added to stabstr to get the corresponding address of the string information.

Uint8_t n_type; The type of the line content, such as functions, function parameters, a line in the source file, etc.

Uint8_t n_other; usually empty.

Uint16_t n_desc; Stores the specific number of rows when type is n_sline.

Uintptr_t n_value; The position of the corresponding content runtime in memory (or relative to the location where the function starts the instruction).

Collect the contents of the header file and the stab table, and analyze the meaning of each field as follows:

```
lijiajun@ubuntu: ~/os/lab1/src/lab1_1/obj/kern
80      LSYM    0       0       00000000 533     double:t(0,13)=r(0,1);8;0;
81      LSYM    0       0       00000000 560     long double:t(0,14)=r(0,1);12;0;
82      LSYM    0       0       00000000 593     _Decimal32:t(0,15)=r(0,1);4;0;
83      LSYM    0       0       00000000 624     _Decimal64:t(0,16)=r(0,1);8;0;
84      LSYM    0       0       00000000 655     _Decimal128:t(0,17)=r(0,1);16;0;
85      LSYM    0       0       00000000 688     void:t(0,18)=(0,18)
86      BINCL   0       0       00000000 2643    ./inc/stdio.h
87      BINCL   0       0       00000650 2657    ./inc/stdarg.h
88      LSYM    0       0       00000000 2672    va_list:t(2,1)=(2,2)=*(0,2)
89      EINCL   0       0       00000000 0
90      EINCL   0       0       00000000 0
91      BINCL   0       0       00000000 2700    ./inc/string.h
92      EXCL    0       0       0000607a 720     ./inc/types.h
93      EINCL   0       0       00000000 0
94      FUN     0       0       f0100040 2715    test_backtrace:F(0,18)
95      PSYM    0       0       00000008 2738    x:p(0,1)
96      SLINE   0       12      00000000 0
97      SLINE   0       13      0000000a 0
98      SLINE   0       14      0000001a 0
99      SLINE   0       15      0000001e 0
100     SLINE   0       17      0000002b 0
101     SLINE   0       18      00000047 0
102     SLINE   0       19      00000057 0
103     RSYM    0       0       00000003 2747    x:r(0,1)
```

The function for finding symbol information is debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info) Analysis of kdbug.h, found that the function of this function is to fill the corresponding fields of the Eipdebuginfo structure. Analyze the debuginfo_eip function and find that its execution logic is roughly: the structure of the eip value to be searched and the information to save the result. In the stab table, a binary search is performed on the item whose n_type is file, and it is found that the value of the eip is in the middle of the label of the two source files (the first item in the figure). Then, the labels of the two source files are used as the left and right intervals, and the items in which the type is fun are binary searched, and the value is found between the two function labels. Finally, the eip value is subtracted from the left interval function value, and a binary search is performed to find the corresponding eip line number. The added code is as follows:

```
// Your code here.
 stab_binsearch(stabs,&lline,&rline, N_SLINE,addr);
 if(lline==rline)
 {
     info->eip_line=stabs[lline].n_desc;
 }
 else{
         return -1;
 }
```

Modify the mon_backtrace code to output the search result

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{

            uint32_t ebp=read_ebp();
            uint32_t *tmp_ebp=(uint32_t *)ebp;
            int i,j;
            struct Eipdebuginfo *info;
            struct Eipdebuginfo infos;
            info=&infos;
            while(tmp_ebp)
            {
                   cprintf("ebp:0x%x ",tmp_ebp);
                   cprintf("eip:0x%x ",tmp_ebp[1]);
                   cprintf("arg:");
                   for(i=2;i<7;i++)
                   {
                      cprintf("0x%x ",tmp_ebp[i]);
                   }
                   cprintf("\n");
                   debuginfo_eip(tmp_ebp[1],info);
                   cprintf("     %s:%d     ",info->eip_file,info->eip_line);
                   for(j=0;j<info->eip_fn_namelen;j++)
                       cprintf("%c",info->eip_fn_name[j]);
                   cprintf("+%d",tmp_ebp[1]-info->eip_fn_addr);
                   cprintf("\n");
           tmp_ebp=(uint32_t *)(*tmp_ebp);

           }


        return 0;
}
```

Eip_fn_namelen is used to exclude colon information after the function name

Tmp_ebp[1](eip)-info->eip_fn_addr outputs the byte difference between the current instruction and the first instruction address of the function.

Chapter2 Memory Management

2.1 Physical Page Management

2.1.1 Homework III

Question

In the file kern/pmap.c, you need to implement the code for the following function (see below, given in order):

Boot_alloc()

Mem_init() (before calling check_page_free_list(1))

Page_init()

Page_alloc()

Page_free()

Check_page_free_list() and check_page_alloc() will test your physical page allocator. You need to guide JOS

Then check the success report for check_page_alloc(). It would be helpful to add your own assert() to verify that your assumptions are correct.

Theoretical preparation

The operating system must keep track of which memory regions are free and which are occupied. The JOS kernel manages memory with the minimum granularity of pages (pages), which uses the MMU to map and protect the memory allocated for each block.

Here we have to write the allocation subfunction of the physical memory page. It uses a linked list of structure PageInfo to record which pages are free, and each node in the linked list corresponds to a physical page.

Analysis & Answer

First, let's look at the structure of PageInfo. From the comments we can see that pp_link points to Next page on the free list.pp_ref is the count of pointers (usually in page table entries) to this page, for pages allocated using page_alloc. Pages allocated at boot time using pmap.c's Boot_alloc do not have valid reference count fields.

```
struct PageInfo {
        // Next page on the free list.
        struct PageInfo *pp_link;

        // pp_ref is the count of pointers (usually in page table entries)
        // to this page, for pages allocated using page_alloc.
        // Pages allocated at boot time using pmap.c's
        // boot_alloc do not have valid reference count fields.

        uint16_t pp_ref;
};
```

图 2.1: PageInfo

We enter the pmap.c file, we can find the mem_init function, this sub-function will be called when the kernel starts running, and some initialization settings are set for the memory management system of the whole operating system, such as setting the page table and so on. In this function, we can see that the first step is to execute the i386_detect_memory function. According to the comment, we can know that it is used to detect how much memory space is available in the system.

```
void
mem_init(void)
{
        uint32_t cr0;
        size_t n;

        // Find out how much memory the machine has (npages & npages_basemem).
        i386_detect_memory();
```

图 2.2: i386_detect_memory

Jos divides the entire physical memory space into three parts:

One is from 0x00000 0xA0000, this part is also called basemem, which is available.

This is followed by 0xA0000 0x100000. This part is called IO hole and is not available. It is mainly used to allocate to external devices.

This is followed by 0x100000 0x. This part is called extmem and is available. This is the most important memory area.

This sub-function includes three variables, where npages records the total number of pages in memory, npages_basemem records the number of pages in basemem, and npages_extmem records the number of pages in extmem.

After executing this function, the next instruction is:

Kern_pgdir = (pde_t *) boot_alloc(PGSIZE);

Memset(kern_pgdir, 0, PGSIZE);

Where kern_pgdir is a pointer, pde_t *kern_pgdir, which is a pointer to the operating system's page directory table. When the operating system is working in virtual memory mode, the page directory table is required for address translation. The memory size space we allocate for this page directory table is PGSIZE, which is the size of one page. And first clear this part of the memory.

```
//////////////////////////////////////////////////////////////////////
// create initial page directory.
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
```

图 2.3: kern_pgdir&memset

As the comment says, this function works as Allocate a chunk large enough to hold 'n' bytes, then update nextfree. Make sure nextfree is kept aligned to a multiple of PGSIZE. it is only used temporarily as a page allocator, then the real page allocator we use is the page_alloc() function. The core idea of this function is to maintain a static variable nextfree, which stores the virtual address of the next free memory space that can be used, so every time we want to allocate n bytes of memory, we need to modify this variable. Value.

In boot_alloc, nextfree is the virtual memory address of the next free memory, which is initialized when nextfree is empty.npages is the number of pages. The available memory size is npages × PGSIZE. According to lab1, KERNBASE is the starting address for allocating memory. If nextfree is greater than the value of KERNBASE + npages × PGSIZE, the pointer address overflows.The modified code is as follows

```
boot_alloc(uint32_t n)
{
        static char *nextfree;  // virtual address of next byte of free memory
        char *result;

        // Initialize nextfree if this is the first time.
        // 'end' is a magic symbol automatically generated by the linker,
        // which points to the end of the kernel's bss segment:
        // the first virtual address that the linker did *not* assign
        // to any kernel code or global variables.
        if (!nextfree) {
                extern char end[];
                nextfree = ROUNDUP((char *) end, PGSIZE);
        }

        // Allocate a chunk large enough to hold 'n' bytes, then update
        // nextfree.  Make sure nextfree is kept aligned
        // to a multiple of PGSIZE.
        //
        // LAB 2: Your code here.
        result = nextfree;
        nextfree = ROUNDUP(nextfree+n, PGSIZE);
        if((uint32_t)nextfree - KERNBASE > (npages*PGSIZE))
                panic("Out of memory!\n");
        return result;

}
```

图 2.4: boot_alloc

Then back to the mem_init function, we need to add an instruction that asks us to allocate an array of npages 'struct PageInfo's and store it in 'pages'.The kernel uses this array to keep track of physical pages: for each physical page, there is a corresponding struct PageInfo in this array. 'npages' is the number of physical pages in memory. Use memset to initialize all fields of each struct PageInfo to 0.

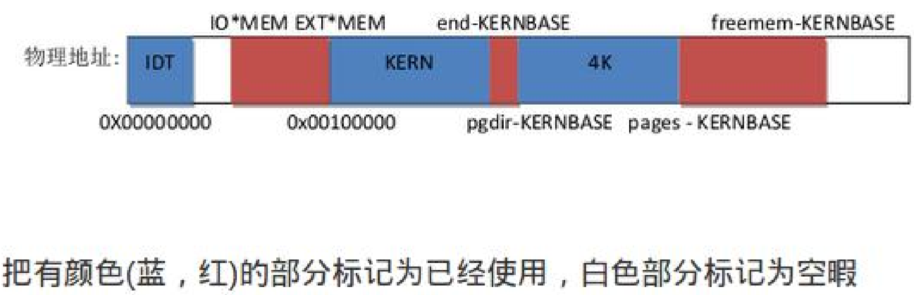
We naturally think of the boot_alloc function that was just completed. By calling it and the memset function, we can complete the required functions. The modified code is as follows:

```
// each physical page, there is a corresponding struct PageInfo in this
// array.  'npages' is the number of physical pages in memory.  Use memset
// to initialize all fields of each struct PageInfo to 0.
// Your code goes here:
pages = (struct PageInfo*)boot_alloc(npages * sizeof(struct PageInfo))
memset(PageInfo,0,npages * sizeof(struct PageInfo))
```

图 2.5: mem_init

Referring to the figure below, we can know that the various blocks of memory are as follows:

I. Physical page 0 is in use for the sake of the fact that we store IDT and some BIOS structure in it.
II. The IO hole (IOPHYSMEM, EXTPHYSMEM) should never be allocated.
III. The extended memory, which mainly refer to the memory we allocate to 'pages' and 'kern pgdir' are in use.
IV. Then base physical memory are free.



In the for loop, we determine the state of the current page. If the page is already occupied, set the pp_ref attribute in the PageInfo structure to one; if it is a free page, then send the page to the pages_free_list list, the modified code show as below

```c
void
page_init(void)
{
	// The example code here marks all physical pages as free.
	// However this is not truly the case.  What memory is free?
	//  1) Mark physical page 0 as in use.
	//     This way we preserve the real-mode IDT and BIOS structures
	//     in case we ever need them.  (Currently we don't, but...)
	//  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
	//     is free.
	//  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
	//     never be allocated.
	//  4) Then extended memory [EXTPHYSMEM, ...).
	//     Some of it is in use, some is free. Where is the kernel
	//     in physical memory?  Which pages are already in use for
	//     page tables and other data structures?
	//
	// Change the code to reflect this.
	// NB: DO NOT actually touch the physical memory corresponding to
	// free pages!
	size_t i;
	page_free_list = NULL;

	//num_alloc：The number of pages that have been occupied in the extmem area
	int num_alloc = ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE;
	//num_iohole：Number of pages occupied in the io hole area
	int num_iohole = 96;

	for(i=0; i<npages; i++)
	{
	    if(i==0)
	    {
	        pages[i].pp_ref = 1;
	    }
	    else if(i >= npages_basemem && i < npages_basemem + num_iohole + num_alloc)
	    {
	        pages[i].pp_ref = 1;
	    }
	    else
	    {
	        pages[i].pp_ref = 0;
	        pages[i].pp_link = page_free_list;
	        page_free_list = &pages[i];
	    }
}
```

图 2.6: page_init

Then, we go to implement the page_alloc function, through the annotation, we know that the function of the function is to allocate a physical page. We can refer to the instructions in the comments to implement it, the implementation code is as follows

```c
struct PageInfo *
page_alloc(int alloc_flags)
{
	// Fill this function in
	struct PageInfo *result;
	if(page_free_list == NULL)
	{
		return NULL;
	}
	else
	{
		result= page_free_list;
		page_free_list = result->pp_link;
		result->pp_link = NULL;
	}
	if (alloc_flags & ALLOC_ZERO)
		memset(page2kva(result), 0, PGSIZE); //Hint
	return result;
}
```

图 2.7: page_alloc

Then, we will complete the page_free function, we should note that this function should only be called when pp->pp_ref reach 0. Then, the way we implement it is to add it to the page_free_list, the implementation code is as follows

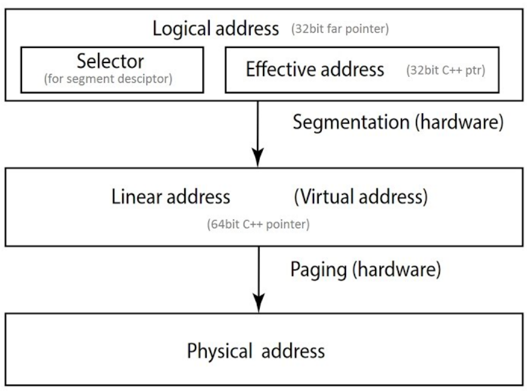
```
void
page_free(struct PageInfo *pp)
{
        // Fill this function in
        // Hint: You may want to panic if pp->pp_ref is nonzero or
        // pp->pp_link is not NULL.
        assert(pp->pp_ref == 0);
        assert(pp->pp_link == NULL);

        pp->pp_link = page_free_list;
        page_free_list = pp;
}
```

图 2.8: page_free

2.2 Virtual Memory

2.2.1 Virtual address, linear address and physical address

In the x86 architecture, a virtual address is composed of two parts, one is a segment selector and the other is a segment offset. A Linear Address refers to an address obtained by converting a virtual address by a segment address translation mechanism. A physical address (Physical Addresses) is the real memory address obtained by the paging address translation mechanism after converting the linear address. This address will eventually be sent to the address bus of your memory chip. The specific relationship of the three addresses is as follows



Exercise

GDB can only access QEMU's memory through virtual addresses, but when learning to create virtual addresses, we also need to check the physical address at the same time. Learn QEMU's monitor command, especially the xp command, which allows you to check physical memory.

Answer

Open Terminal, enter qemu-system-i386 -hda obj/kern/kernel.img -monitor stdio -gdb tcp::26000 -D qemu.log , after the correct installation, we can use the monitor command.

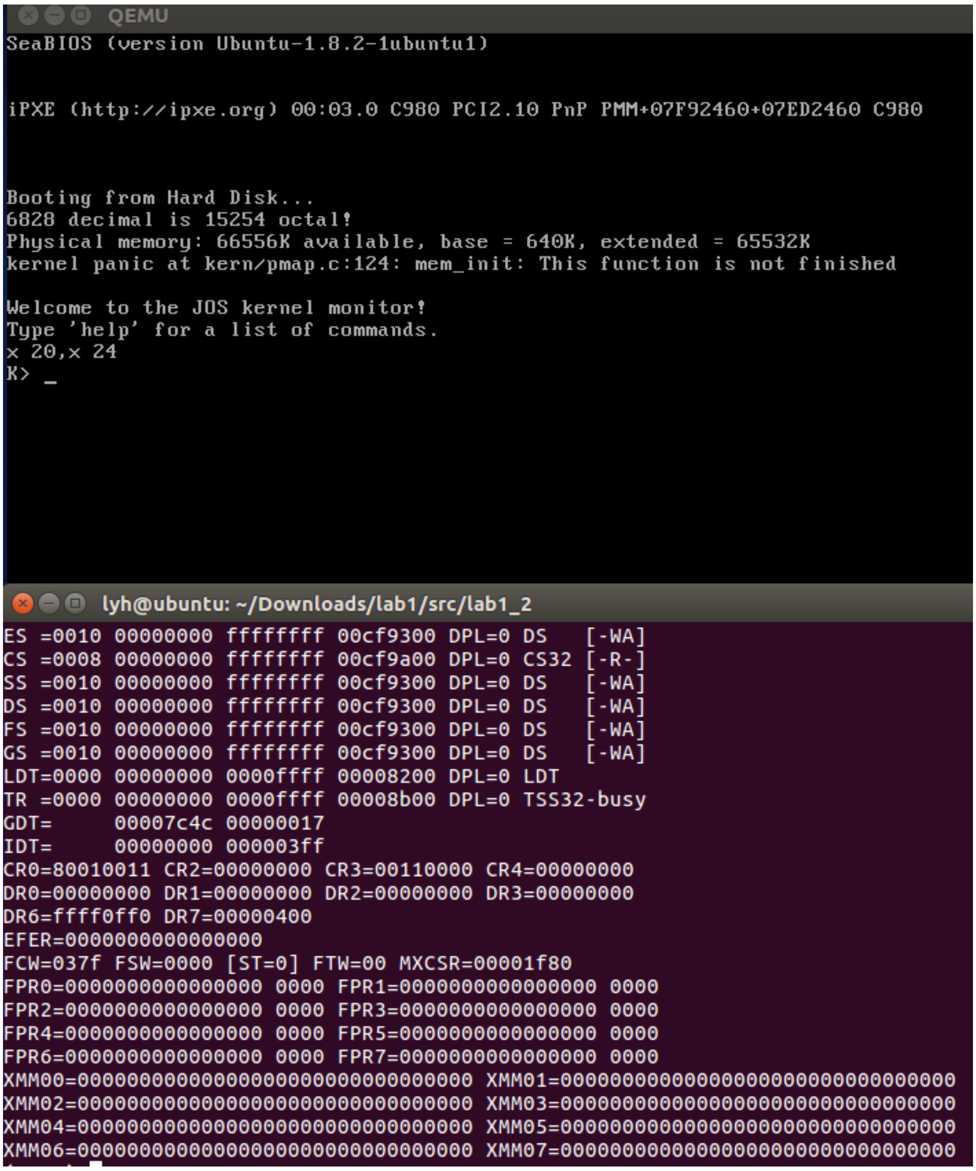


图 2.9: qemu command

Exercise

Assuming the following kernel code is correct, then what type of variable x will be, uintptr_t or Physaddr_t?

Mystery_t x;
Char* value = return_a_pointer();
*value = 10;
x =(mystery_t) value;

Answer

Since the * operator is used here to resolve the address, the variable x should be of type uintptr_t.

2.2.2 Reference counting

In the previous experiment, we encountered pp_ref, which records how many different virtual addresses exist on each physical page to reference it. When this value becomes 0, the physical page can be released. In general, the pp_ref value of any physical page p is equal to the number of times it is mapped by the virtual page under the virtual address UTOP in all page table entries (the address range above UTOP is already mapped at startup) Finished, will not be changed afterwards).

2.2.3 Page Table Management

Now we can start writing programs that manage page tables: including inserting and deleting linear address-to-physical address mappings, and creating page tables.

2.2.4 Homework IV

Question

In the file kern/pmap.c, you must implement code for the following functions.
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

Theoretical preparation

In the previous description, we have a basic understanding of the concept of virtual address, linear address, and physical address.

Physical address: A unit address for memory chip level that corresponds to the address bus to which the processor and CPU are connected.

Logical address: refers to the portion of the offset address associated with the segment generated by the program. For example, in C language pointer programming, you can read the value of the pointer variable itself (& operation). In fact, this value is the logical address. It is relative to the address of your current process data segment and is not related to the absolute physical address.

Linear address or virtual address: Similar to the logical address, it is also an unreal address. If the logical address is the corresponding hardware platform segment management pre-conversion address, then the linear address corresponds to the pre-conversion address of the hardware page memory.

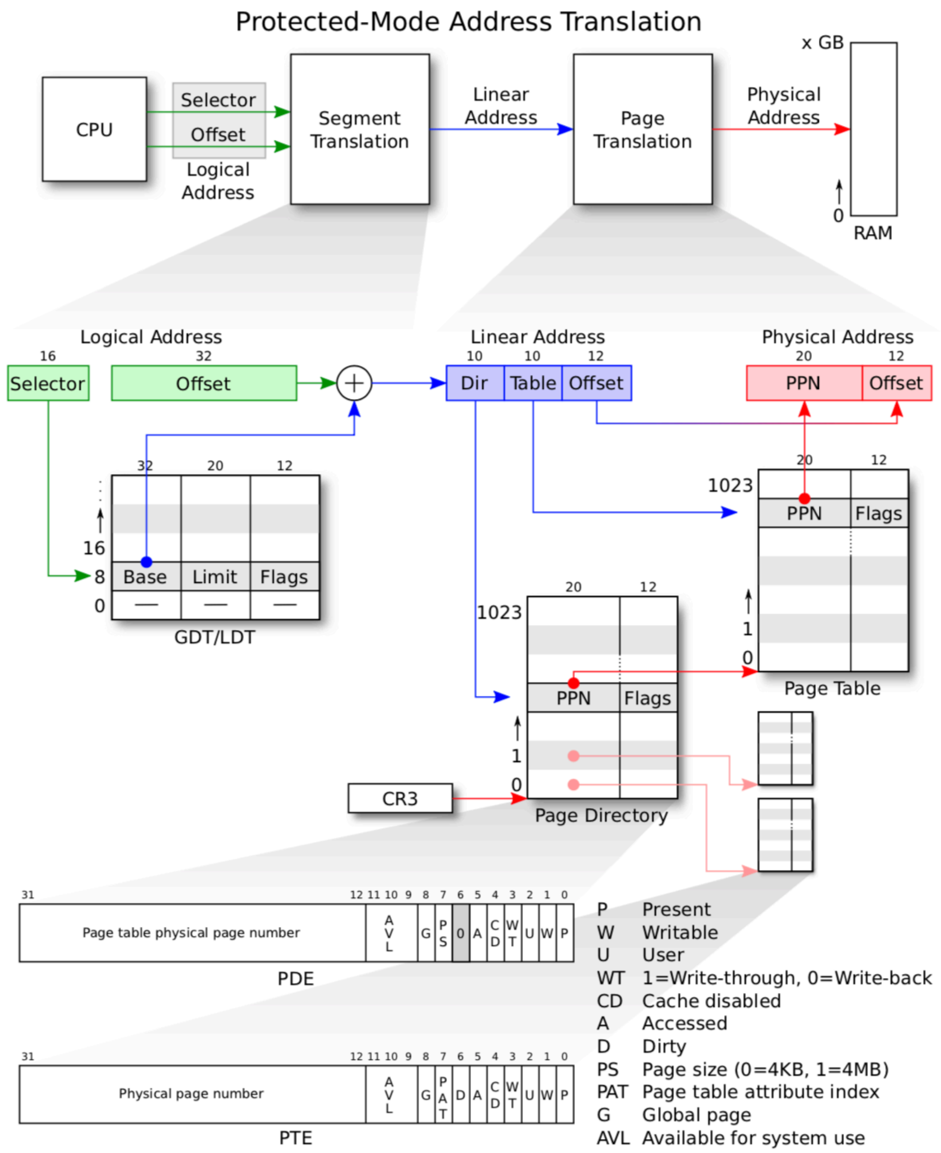
We also need to know the knowledge about the page table.

Our physical memory has a total of 4GB, and we assign it to the page, each page is 4KB in size.

The 4GB (2 to the 32th power) linear address space can be divided into 1048576 (2 to the 20th power, that is, 1M, can also be regarded as 1024 * 1024) pages, so you can randomly extract these pages, every 1024 The pages are a group and can be divided into 1024 groups. For each group of 1024 pages of physical addresses, arranged in a certain order can constitute a table (each entry is the physical address of a page), this table is the page table. The size of the page table is 1024*4B=4KB, which is exactly the size of a physical page.

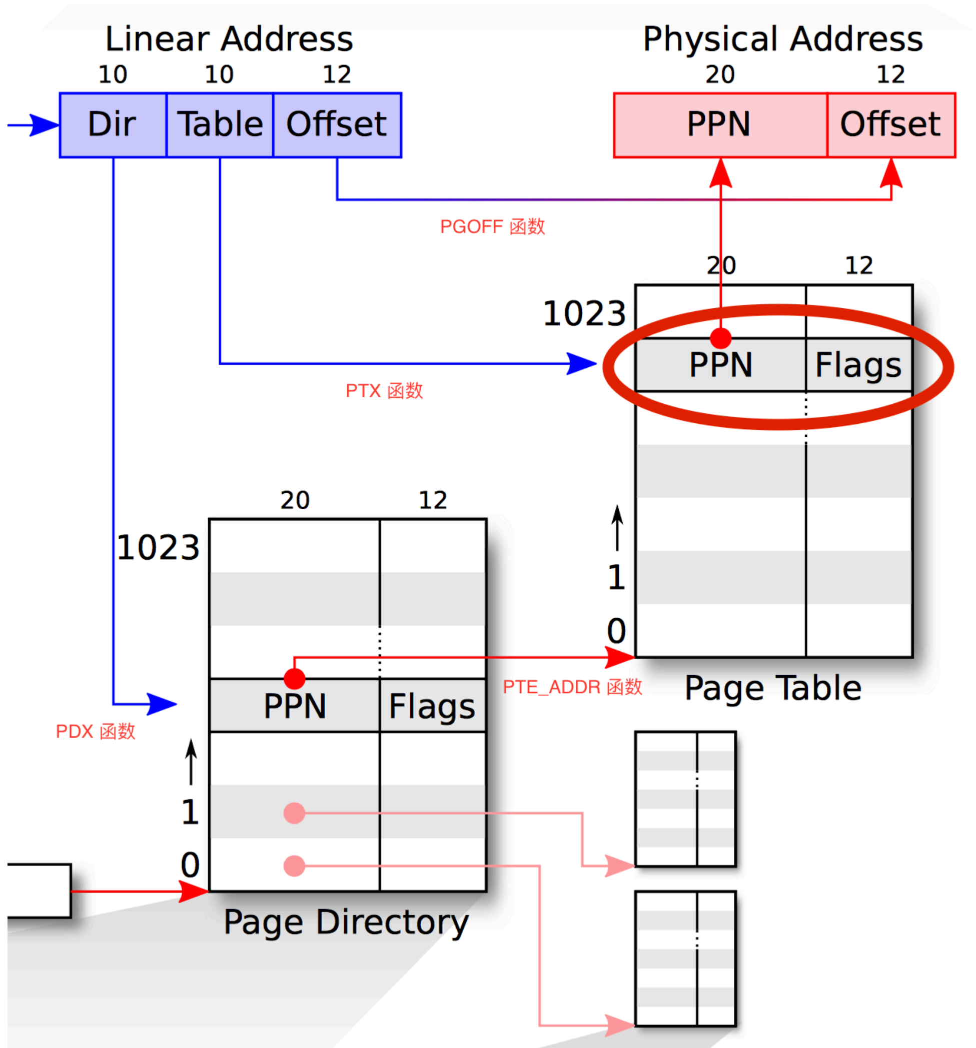
Because it has been divided into 1024 groups, each group has a page table (size 4KB), so these 1024 page tables can be pointed to by a table, this is the page directory. Similar to the page table, the page directory has a total of 1024 entries (called page directory entries), and the content of each page directory entry is the physical address of a page table. The size of the page table is 1024*4B=4KB, which is exactly the size of a physical page.

The conversion of three addresses and the page table mechanism are shown in the figure below.



Analysis & Answer

By commenting, we can know that the function of this function is given a page directory table pointer pgdir , which should return the page table entry pointer corresponding to the linear address va.We need to complete the transformation as shown in the figure, return the corresponding page table address, that is, the virtual address of the part circled by the red circle:



In addition, we need to understand the meaning of the three parameters.In addition, we need to understand the meaning of the three parameters, pgdir means the page directory item pointer, va means linear address, JOS equals virtual address, create means if page directory entry does not exist or not.

Here we should complete the following steps:

1. Find the page table page where the virtual address is located by the page directory table for the page directory entry address dic_entry_ptr in the page directory. (7-8)

2. Determine if the page table page corresponding to this page directory entry is already in memory. (10)

3. If yes, calculate the base address page_base of this page table page, and then return the address of the page entry corresponding to va &page_base[page_off] (23-25)

4. If not, and create is true, a new page is allocated, and the information of this page is added to the page directory entry dic_entry_ptr. (11-18)

5. If create is false, it returns NULL. (19-20)

The modified code is shown in the figure

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        unsigned int page_off;
        pte_t * page_base = NULL;
        struct PageInfo* new_page = NULL;

        unsigned int dic_off = PDX(va);
        pde_t * dic_entry_ptr = pgdir + dic_off;

        if(!(*dic_entry_ptr & PTE_P))
        {
             if(create)
             {
                    new_page = page_alloc(1);
                    if(new_page == NULL) return NULL;
                    new_page->pp_ref++;
                    *dic_entry_ptr = (page2pa(new_page) | PTE_P | PTE_W | PTE_U);
             }
             else
                 return NULL;
        }

        page_off = PTX(va);
        page_base = KADDR(PTE_ADDR(*dic_entry_ptr));
        return &page_base[page_off];
}
```

图 2.10: pgdir_walk

Next we complete the boot_map_region function. By comment, we can see that the function of this function is to map [va, va+size) of virtual address space to physical [pa, pa+size) in the page table rooted at pgdir. Size is a multiple of PGSIZE,and va and pa are both page-aligned.Use permission bits perm|PTE_P for the entries.

The mapping of the virtual address space range [va, va+size) to the physical space [pa, pa+size) is added to the page table pgdir. The main purpose of this function is to set the address range above the virtual address UTOP. The address mapping of this part is static and will not change during the operation of the operating system, so the value of the pp_ref field in the PageInfo structure of this page. No change will happen.

The steps to be completed by this function are as follows:

Need to complete a loop, using the pgdir_walk function we just completed, we can use a loop to map all the memory in size bytes.The modified code is shown in the figure:

```c
//
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
	uint32_t i;
	pte_t *pt_table_entry = NULL;
	uint32_t page_num = size / PGSIZE;

	for (i = 0; i < page_num; i++) {
		pg_table_entry = pgdir_walk(pgdir, (void *) va, 1);
		assert(pg_page_entry != NULL);
		*pg_table_entry = pa | perm | PTE_P;
		va += PGSIZE;
		pa += PGSIZE;
	}

}
```

图 2.11: boot_map_region

Next, we complete the page_lookup function, which functions as return the page mapped at virtual address 'va'.If the pte_store parameter is not 0, the page table entry address of this physical page is stored in pte_store.

We only need to call the pgdir_walk function to get the page table entry corresponding to this va, and then determine whether the page is already in memory, and if so, return the PageInfo structure pointer of this page. And store the contents of this page table entry in pte_store.The modified code is shown in the figure:

```c
//
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
	// Fill this function in
	pte_t *entry = NULL;
	struct PageInfo *ret = NULL;

	entry = pgdir_walk(pgdir, va, 0);
	if (entry == NULL)
		return NULL;
	if (!(*entry & PTE_P))
		return NULL;

	ret = pa2page(PTE_ADDR(*entry));
	if (pte_store != NULL)
		*pte_store = entry;

	return ret;
}
```

图 2.12: page_lookup

Go ahead and we'll complete the page_remove function. By commenting, we can know that the function of this function is unmaps the physical page at virtual address 'va'.The modified code is shown in the figure:

```
void
page_remove(pde_t *pgdir, void *va)
{
        // Fill this function in
        pte_t *entry = NULL;
        struct PageInfo *page = page_lookup(pgdir, va, &entry);

        if (page == NULL)
                return;

        page_decref();
        tlb_invalidate(pgdir, va);
        *entry = 0;
}
```

图 2.13: page_remove

Go ahead and we'll complete the page_insert function. By commenting, we can know that the function of the function is map the physical page 'pp' at virtual address 'va'. Combining the previously completed pgdir_walk function and the page_remove function, we first find the page table corresponding to the virtual address va through the pgdir_walk function. Item, modify the value of pp_ref, check the page table entry, determine whether va has been mapped, if it is mapped, delete the mapping, and add the mapping relationship between va and pp to the page table entry.The modified code is shown in the figure:

```
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
        pte_t *entry = NULL;
        entry = pgdir_walk(pgdir, va, 1);

        if (entry == NULL)
                return -E_NO_MEM;

        pp->pp_ref++;
        if(*entry & PTE_P) {
                tlb_invalid(pgdir, va);
                page_remove(pgdir, va);
        }

        *entry = (page2pa(pp) | perm | PTE_P);
        pgdir[PDX(va)] = entry;

        return 0;
}
```

图 2.14: page_insert

2.3 Kernel address space

JOS divides the 32-bit linear address virtual space into two parts. The user environment (process running environment) usually occupies the part of the low address, called the user address space. The operating system kernel always occupies the part of the high address, called the kernel address space. The dividing line between these two parts is a macro ULIM defined in the memlayout.h file. JOS reserves nearly 256MB of virtual address space for the kernel. This can be understood, why in the experiment 1 to design a high address address space for the operating system. If you don't do this, the address space of the user environment is not enough.

2.3.1 Homework V

Question

Fill in the missing code in mem_init() after the call to check_page().

Theoretical preparation

Since the kernel and user memory are present in the address space of each environment, we need to use the permission bits in the x86 page table to allow the user code to access only the user portion of the address space. Otherwise, bugs in the user code may overwrite the kernel data, causing a crash; or the user code can steal private data from other environments. The user environment will have no access to any memory above ULIM, and the kernel can read and write this portion of memory. For the address range [UTOP, ULIM), the kernel and the user environment have the same permissions: they can only be read and not written. Under UTOP The address space is used by the user environment, and the user environment will set permissions to access this part of the memory.

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig -------->  +------------------------------+
 *                     |                              | RW/--
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     :              .               :
 *                     :              .               :
 *                     :              .               :
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                     |                              | RW/--
 *                     |   Remapped Physical Memory   | RW/--
 *                     |                              | RW/--
 *    KERNBASE, ---->  +------------------------------+ 0xf0000000      --+
 *    KSTACKTOP        |     CPU0's Kernel Stack      | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                   |
 *                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
 *                     +------------------------------+                   |
 *                     |     CPU1's Kernel Stack      | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                 PTSIZE
 *                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
 *                     +------------------------------+                   |
 *                     :              .               :                   |
 *                     :              .               :                   |
 *    MMIOLIM ------>  +------------------------------+ 0xefc00000      --+
 *                     |       Memory-mapped I/O      | RW/--  PTSIZE
 * ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
 *                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
 *    UVPT      ---->  +------------------------------+ 0xef400000
 *                     |          RO PAGES            | R-/R-  PTSIZE
 *    UPAGES    ---->  +------------------------------+ 0xef000000
 *                     |           RO ENVS            | R-/R-  PTSIZE
 * UTOP,UENVS ------>  +------------------------------+ 0xeec00000
 * UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebff000
 *                     |       Empty Memory (*)       | --/--  PGSIZE
 *    USTACKTOP  --->  +------------------------------+ 0xeebfe000
 *                     |      Normal User Stack       | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebfd000
 *                     |                              |
 *                     |                              |
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     .                              .
 *                     .                              .
```

Analysis &Answer

In this exercise, three virtual addresses are mapped to the physical page.

First, we will complete the UPAGES mapping UPAGES (0xef000000   0xef400000) up to 4MB, which is the data structure of JOS recording physical page usage.

Currently only one page directory is created, kernel_pgdir, so the first parameter is obviously kernel_pgdir. The second parameter is the virtual address, and UPAGES is originally given as a virtual address. The third parameter is the mapped memory block size. The fourth parameter is the physical address mapped to, and the physical address of pages can be taken directly. Permissions PTE_U indicates that the user has permission to read.Currently only one page directory is created, kernel_pgdir, so the first parameter is obviously kernel_pgdir. The second parameter is the virtual address, and UPAGES is originally given as a virtual address. The third parameter is the mapped memory block size. The fourth parameter is the physical address mapped to, and the physical address of pages can be taken directly. Permissions PTE_U indicates that the user has permission to read. The modified code is shown in the figure

```
//////////////////////////////////////////////////////////////////////
// Now we set up virtual memory

//////////////////////////////////////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE
// Your code goes here:
cprintf("UPAGES\n");
boot_map_region(kern_pgdir, (uintptr_t) UPAGES, npages*sizeof(struct PageInfo), PADDR(pages), PTE_U | PTE_P);
```

Then there is the memory stack, the kernel stack (0xefff8000 0xf0000000) 32k-B Bootstrap represents the lowest address of the stack. Since the stack grows to the lower address, it is actually the top of the stack. We will map the address space in [KSTACKTOP-KSTKSIZE, KSTACKTOP) The modified code is shown in the figure

```
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:
cprintf("memory stack\n");
boot_map_region(kern_pgdir, (uintptr_t) (KSTACKTOP-KSTKSIZE), KSTKSIZE, PADDR(bootstack), PTE_W | PTE_P);
```

Finally, the kernel part, we will map the kernel (0xf0000000 0xffffffff) 256MB The modified code is shown in the figure

```
//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
cprintf("kernel\n");
boot_map_region(kern_pgdir, (uintptr_t) KERNBASE, ROUNDUP(0xffffffff - KERNBASE, PGSIZE), 0, PTE_W | PTE_P);
```

As a result, the code works successfully.

```
——————————内存空间——————————
Physical memory: 66556K available,
base = 640K,
extended = 65532K,
IO = 384K

——————————各类内存页数——————————
npages: 16639

npages_extmem: 16383
npages_basemem: 160
npages_IO: 96



os页表起始时的内存地址: f0118000

os页表结束、pages开始的内存地址: f0119000

os的页表的第0项(指针kern_pgdir)所在的内存地址: 118005

npages*sizeof(struct PageInfo) = 207f8

pages页表结束的内存地址: f013a000

check_page_alloc() succeeded!
check_page() succeeded!
UPAGES
map_region_size =133112,(32 pages)
va_begin= 0xef000000
va_end= 0xef021000
memory stack
map_region_size =32768,(8 pages)
va_begin= 0xefff8000
va_end= 0xf0000000
kernel
map_region_size =268435456,(65536 pages)
va_begin= 0xf0000000
va_end= 0x0
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

```
nikoni@ubuntu:~/Documents/lab1/src/lab1_1$ make grade
make clean
make[1]: Entering directory '/home/nikoni/Documents/lab1/src/lab1_1'
rm -rf obj .gdbinit jos.in qemu.log
make[1]: Leaving directory '/home/nikoni/Documents/lab1/src/lab1_1'
./grade-lab2
make[1]: Entering directory '/home/nikoni/Documents/lab1/src/lab1_1'
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ 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/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/nikoni/Documents/lab1/src/lab1_1'
running JOS: (1.1s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

2.3.2 Question IV

1)

What entries (rows) in the page directory have been filled in at this point? 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):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

Answer

Entry	Base Virtual Address	Points to(logically)
1023	0xffc00000 Points to(logically)	Page table for top 4MB of physical memory.This is the last address finding page table that the kernel can use.
1022	0xff800000	Page table for 248MB–(252MB-1)physical memory
...	...	Page table for physical memory
960	0xf0000000(KERNBASE)	static data 0–(4MB-1) physical memory
959	0xefc00000(VPT)	Page directory self (kernel RW).This is the first page table and it ’ s in the bottom of the physical memory.
958	0xef800000(ULIM)	Page table for kernel stack.It is mapped into the physical memory which is the same as bootstack.We only map the memory that the same as KSTACKSIZE.The rest memory that wasn’ t mapped is used to avoid the overflow of the kernel stack.
957	0xef400000(UVPT)	Same as 959(user kernel R)
956	0xef00000(UPAGES)	Page table for structure pages[]
...	...	NULL
2	0x00800000	NULL
1	0x00400000	NULL
0	0x00000000	The start of the virtual memory .The same as 960(then turn to NULL)

表 2.1: Answer

2)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?

Answer

User is not allowed to access kernel memory for safety reasons. If user have the permission, bugs in user code may led to crash. It is the paging mechanism that protects kernel address in JOS. If the flag bit PTE_U is 0 in a page, that means user have no permission to read or write the page.

3)What is the maximum amount of physical memory that this operating system can support? Why?

Answer

```
// Page directory and page table constants.
#define NPDENTRIES      1024            // page directory entries per page directory
#define NPTENTRIES      1024            // page table entries per page table

#define PGSIZE          4096            // bytes mapped by a page
#define PGSHIFT         12              // log2(PGSIZE)

#define PTSIZE          (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry
#define PTSHIFT         22              // log2(PTSIZE)

#define PTXSHIFT        12              // offset of PTX in a linear address
#define PDXSHIFT        22              // offset of PDX in a linear address
```

图 2.15: PTSIZE

```
// User read-only virtual page table (see 'uvpt' below)
#define UVPT            (ULIM - PTSIZE)
// Read-only copies of the Page structures
#define UPAGES          (UVPT - PTSIZE)
// Read-only copies of the global env structures
#define UENVS           (UPAGES - PTSIZE)
```

图 2.16: UPAGES

```
struct PageInfo {
        // Next page on the free list.
        struct PageInfo *pp_link;

        // pp_ref is the count of pointers (usually in page table entries)
        // to this page, for pages allocated using page_alloc.
        // Pages allocated at boot time using pmap.c's
        // boot_alloc do not have valid reference count fields.

        uint16_t pp_ref;
};
```

图 2.17: Page

2GB.

Pages use up to 4MB space, and each PageInfo use 8Byte. 4M / 8 * 4kB=2GB (4kB per page).

4)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?

Answer

The total overhead to manage maxium amount of physical memory is:

786432 bytes (struct Pages [1])

4096 bytes (one page directory [2])

262144 bytes (64 page tables [3])

1052672 bytes (1MB)

The only way I can see to reduce that amount is to use 4MB pages, this would reduce the struct Page allocations to 768 bytes and no need to allocate page tables.

On the hand, the greater the granularity the greater the amount of unused chunks we'll have on the allocated pages which means we'll spend memory...

[1] struct Page overhead was calculated this way 256 * 1024 * 1024 / 4096 * 12 256MB Page size size of struct Page

[2] Page directory is 4096 bytes long by definition

[3] Page table overhead was calculated this way

(256 * 1024 * 1024 / (4096 * 1024)) * 4096 256MB PG maps 4MB PG size