清华大学本科生考试试题专用纸

考试课程：**操作系统（A 卷）**　　　时间：2014 年 04 月 08 日上午 9:50~11:50

系别：_________ 班级：_________ 学号：_________ 姓名：

答卷注意事项：　1. 在开始答题前，请在试题纸和答卷本上写明系别、班级、学号和姓名。

2. 在答卷本上答题时, 要写明题号, 不必抄题。

3. 答题时, 要书写清楚和整洁。

4. 请注意回答所有试题。本试卷有 6 个题目，共 14 页。

5. 考试完毕, 必须将试题纸和答卷本一起交回。

一、（20 分）内存管理（Memory Management）是操作系统的重要职能之一，现代操作系统基于硬件所提供的内存管理单元（Memory Management Unit），可以为应用程序提供相互隔离的虚拟地址空间，同时对物理内存进行高效的管理。在 32 位 x86 架构提供的 MMU 中，除了传统的段模式以外，也同时包括页管理机制。页管理所需要的硬件支持包括两个部分：一是完成虚拟地址到物理地址映射的页表，二是页异常（Page Fault）。

1) 在 32 位的 x86 系统中，一般使用二级页表，分别用虚拟地址的 31-22 位和 21-12 位作为页表内相应页表项的偏移，此时一个物理页的大小为___(1a)____K；实际上，x86 系统同样允许只使用一级页表，页表项偏移仍然取虚拟地址的 31-22 位，此时一个物理页的大小为__(1b)__K。

2) 发生页异常时，硬件会保存执行时的上下文并关闭中断，然后跳转到操作系统设置好的页错误处理例程，这里的“上下文”应该包括（在你认为需要保存的寄存器前打勾，并简述如果不保存会产生什么问题）：

   (2a)（　）指令计数器（CS, EIP）

   ____

   (2b)（　）堆栈指针（SS, ESP）

   ____

   (2c)（　）通用寄存器（EAX、EBX、……）

   ____

   (2d)（　）执行时标志位寄存器（EFLAGS）

   ____

   (2e)（　）控制寄存器（CR0、CR3、……）

   ____

3) 页异常处理完毕后，返回用户程序继续执行，此时执行的第一条用户指令为（　　）

   A. 触发页异常指令的上一条指令

   B. 触发页异常的指令

   C. 触发页异常指令的下一条指令

4) 除了维护基本的地址映射关系外，x86 页表的每一个页表项还包括一些其它配置和信息位，例如该页是否可写（W），是否可以在 Ring 3 访问（U），是否曾被访问过（A），以及是否曾被写过（D）。请根据 x86 页表对这些位的定义，在下表中填写在页表项的几种情况下进行各种操作时，页表项的内容会如何变化。（只需写出会变化的位和/或会产生的事件，如缺页异常，形式可参考表中已填入的部分内容）

	W=0 U=0
	A=0 D=0
		W=1 U=0
		A=0 D=0
			W=0 U=1
			A=1 D=0
				W=1 U=1
				A=1 D=1

在 ring0 读				无变化
在 ring0 写		A→1，D→1		
在 ring3 读				
在 ring3 写			缺页异常	

二、（20 分）进程/线程管理（Process/Thread Management）是操作系统的重要职能之一，现代操作系统基于硬件所提供的内存管理单元（Memory Management Unit），可以为进程提供相互隔离的虚拟地址空间，为线程提供共享的虚拟地址空间。以下我们对 32 位 x86 架构的 uCore 所实现的进程管理机制进行讨论。

1) 在 ucore 中，管理一个用户进程的进程控制块数据结构为 proc_struct，管理一个内核线程的线程控制块数据结构为__(1a)__。在 proc_struct 中，为了有效管理用户进程，请问可唯一标识一个进程的 field 是__(1b)__；用户进程控制块与内核线程控制块相比，数据内容肯定不同的 field 包括__(1c)__、__(1d)__、__(1e)__。
2) 在 ucore 中，一个用户进程具有“自己”的用户栈，当用户进程通过系统调用进入到内核态开始继续执行 ucore 指令时，进程的页表起始地址是否会改变？__(2a)__。当用户进程在用户态执行时，硬件产生了一个中断，打断了用户进程的执行，这时 CPU 将开始执行中断服务例程，，这个时候的页表起始地址是否已经不是被打断的用户进程的页表起始地址了？__(2b)_。
3) 在 ucore 中，当用户进程访问的某个虚拟地址的映射关系不在 TLB 中时，是否一定会产生异常？__(3a)__。当用户进程访问的某个虚拟地址在其页表中没有 valid 的页表项，是否一定回产生异常？__(3b)__。
4) 在 ucore 中，当用户进程 A 与用户进程 B 进行进程上下文切换时，需要保存相关的寄存器内容，请问是否需要保存 CS？__(4a)__。是否需要保存 EIP？__(4b)_。是否需要保存 SS？__(4c)__。是否需要保存 ESP？__(4d)__。
5) 在 ucore 中，如果当父进程创建子进程时，如果没有 COW 机制，则 fork 系统调用会创建新的子进程的进程控制块，创建新的子进程的页表，并把父进程的代码段和数据段所占的物理内存空间复制一份到新的物理内存空间，并更新子进程页表。如果采用了 COW 机制，则 fork 系统调用的处理过程是？（用不超过 6 行文字进行描述）

三、（15 分）Bootloader 是 ucore 操作系统启动中的很重要的一部分，Bootloader 是由 BIOS 代码读入内存，然后跳转到它开始执行的。请参考 bootasm.S 和 bootmain.c 的源代码，回答下列问题：
1) Bootloader 包含在硬盘主引导扇区中，硬盘主引导扇区的主要特征有哪些？
2) Bootloader 执行的第一条指令是哪一行？Bootloader 从实模式进入保护模式后执行的第一条指令是哪一行？为什么要转换到保护模式？
3) Bootloader 在完成从硬盘扇区读入 ucore 内核映像后是如何跳转到 ucore 内核代码的？

=========/libs/elf.h=========

```
#ifndef __LIBS_ELF_H__
#define __LIBS_ELF_H__

#include <defs.h>

```

```
#define ELF_MAGIC    0x464C457FU          // "\x7FELF" in little endian

/* file header */
struct elfhdr {
    uint32_t e_magic;     // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;      // 1=relocatable, 2=executable, 3=shared object, 4=core image
    uint16_t e_machine;   // 3=x86, 4=68K, etc.
    uint32_t e_version;   // file version, always 1
    uint32_t e_entry;     // entry point if executable
    uint32_t e_phoff;     // file position of program header or 0
    uint32_t e_shoff;     // file position of section header or 0
    uint32_t e_flags;     // architecture-specific flags, usually 0
    uint16_t e_ehsize;    // size of this elf header
    uint16_t e_phentsize; // size of an entry in program header
    uint16_t e_phnum;     // number of entries in program header or 0
    uint16_t e_shentsize; // size of an entry in section header
    uint16_t e_shnum;     // number of entries in section header or 0
    uint16_t e_shstrndx;  // section number that contains section name strings
};

/* program section header */
struct proghdr {
    uint32_t p_type;   // loadable code or data, dynamic linking info,etc.
    uint32_t p_offset; // file offset of segment
    uint32_t p_va;     // virtual address to map segment
    uint32_t p_pa;     // physical address, not used
    uint32_t p_filesz; // size of segment in file
    uint32_t p_memsz;  // size of segment in memory (bigger if contains bss)
    uint32_t p_flags;  // read/write/execute bits
    uint32_t p_align;  // required alignment, invariably hardware page size
};

#endif /* !__LIBS_ELF_H__ */
```

=========/boot/bootasm.S=========

```
#include <asm.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

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
.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

    # Switch from real to protected mode, using a bootstrap GDT
    # and segment translation that makes virtual addresses
    # identical to physical addresses, so that the
    # effective memory map does not change during the switch.
    lgdt gdtdesc
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0

    # Jump to next instruction, but in 32-bit code segment.
    # Switches processor into 32-bit mode.
    ljmp $PROT_MODE_CSEG, $protcseg

.code32                                             # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax                           # Our data segment selector
    movw %ax, %ds                                     # -> DS: Data Segment
    movw %ax, %es                                     # -> ES: Extra Segment
    movw %ax, %fs                                     # -> FS
    movw %ax, %gs                                     # -> GS
    movw %ax, %ss                                     # -> SS: Stack Segment

    # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain

    # If bootmain returns (it shouldn't), loop.
spin:
    jmp spin

# Bootstrap GDT
.p2align 2                                          # force 4 byte alignment
gdt:
    SEG_NULLASM                                     # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)               # code seg for bootloader and kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)                   # data seg for bootloader and kernel

gdtdesc:
    .word 0x17                                      # sizeof(gdt) - 1
    .long gdt                                       # address gdt
```

=========/boot/bootmain.c=========

```
#include <defs.h>
#include <x86.h>
#include <elf.h>

/* *********************************************************************
 * This a dirt simple boot loader, whose sole job is to boot
 * an ELF kernel image from the first IDE hard disk.
 *
 * DISK LAYOUT
 *  * This program(bootasm.S and bootmain.c) is the bootloader.
 *    It should be stored in the first sector of the disk.
 *
 *  * The 2nd sector onward holds the kernel image.
 *
 *  * The kernel image must be in ELF format.
 *
 * BOOT UP STEPS
 *  * when the CPU boots it loads the BIOS into memory and executes it
 *
 *  * the BIOS intializes devices, sets of the interrupt routines, and
 *    reads the first sector of the boot device(e.g., hard-drive)
 *    into memory and jumps to it.
 *
 *  * Assuming this boot loader is stored in the first sector of the
 *    hard-drive, this code takes over...
 *
 *  * control starts in bootasm.S -- which sets up protected mode,
 *    and a stack so C code then run, then calls bootmain()
 *
 *  * bootmain() in this file takes over, reads in the kernel and jumps to it.
 * */

#define SECTSIZE        512
#define ELFHDR          ((struct elfhdr *)0x10000)      // scratch space

/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}

/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1);                         // count = 1
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);                      // cmd 0x20 - read sectors

    // wait for disk to be ready
```

```c
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}

/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}

/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph ++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
```

四、（15 分）进程管理是操作系统提供给应用程序的一种用于进程控制的服务。下面是一个用 fork 系统调用完成进程创建的程序。试回答下面问题：

1) 描述 fork 系统调用的功能、调用接口。
2) 补全程序的输出信息。

=========fork.c=========

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#define DEFAULT_TIME 5
#define DEFAULT_STATUS 0
int main (int argc, char **argv) {
    int child_id;
    int seconds;
    int status;
    pid_t whodied;

    status = DEFAULT_STATUS;

    if (argc == 1)
        seconds = DEFAULT_TIME;
    else
        seconds = atoi (argv[1]);
    printf ("Here I am in the program!  Time to wait = %d\n", seconds);
    system ("ps -l");
    child_id = fork();
    if (child_id) {
        printf ("I'm the parent at Line 33.  My parent's process ID is %d, My process ID is %d, st
atus = %d.\n", getpid(), getppid(), status);
```

```
        whodied = wait (&status);
        printf ("Child %d exited ", whodied);
/* WIFEXITED evaluates to true when the process exited by using an exit(2V) call.
 If WIFEXITED(status) is non-zero, WEXITSTATUS evaluates to the low-order byte of the
argument that the child process passed to _exit() (see exit(2V)) or exit(3), or the value the
 child process returned from main() (see execve(2V)).
*/
        if (! WIFEXITED(status)) {
            printf ("abnormally!\n");
            }
        else {
            printf ("with status %d.\n", WEXITSTATUS(status));
            }
        printf ("I'm the parent at Line 43.  My parent's process ID is %d, My process ID is %d, st
atus = %d.\n",getpid(), getppid(), WEXITSTATUS(status));
        return status;
        }
    else {
        status = 17;
        sleep(seconds);
        printf ("I'm the child.  My parent's process ID is %d, My process ID is %d, status = %d.\n
",getpid(), getppid(), status);
        printf ("Bye now!\n");
        return status;
        }
    }
```

fork 程序的两次执行时的输出信息

```
xyong@portal:~/work$ ./a.out
Here I am in the program!  Time to wait = __(1)__
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 11739 11738  0  80   0 -  6926 wait   pts/0    00:00:00 bash
0 S  1000 11862 11739  0  80   0 -  1041 wait   pts/0    00:00:00 a.out
0 S  1000 11863 11862  0  80   0 -  1101 wait   pts/0    00:00:00 sh
0 R  1000 11864 11863  0  80   0 -  2433 -      pts/0    00:00:00 ps
I'm the parent at Line 33.  My parent's process ID is __(2)__, My process ID is __(3)__, status = __(4)__.
I'm the child.  My parent's process ID is __(5)__, My process ID is __(6)__, status = __(7)__.
Bye now!
Child 11865 exited with status __(8)__.
I'm the parent at Line 43.  My parent's process ID is __(9)__, My process ID is __(10)__, status = __(11)__.
xyong@portal:~/work$ ./a.out 3
Here I am in the program!  Time to wait = __(12)__
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 11739 11738  0  80   0 -  6926 wait   pts/0    00:00:00 bash
0 S  1000 11866 11739  0  80   0 -  1041 wait   pts/0    00:00:00 a.out
```

```
0 S  1000 11867 11866  0  80   0 -  1101 wait   pts/0    00:00:00 sh
0 R  1000 11868 11867  0  80   0 -  2433 -      pts/0    00:00:00 ps
I'm the parent at Line 33.  My parent's process ID is __(13)__, My process ID is __(14)__, status =
__(15)__.
I'm the child.  My parent's process ID is __(16)__, My process ID is __(17)__, status = __(18)__.
Bye now!
Child __(19)__ exited with status __(20)__.
I'm the parent at Line 43.  My parent's process ID is 11866, My process ID is __(21)__, status = __(22)__.
xyong@portal:~/work$
```

五、（15 分）为实现函数的调用和返回功能，X86 指令集中提供了 call 和 ret 两条指令。为在操作系统内核执行过程中分析了解函数函数的嵌套调用关系，ucore 中实现了函数 print_stackframe，用于跟踪函数调用堆栈中记录的返回地址。如果能够正确实现此函数，它将在 qemu 模拟器中得到类似如下的输出：

```
……
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c53
    kern/init/init.c:28: kern_init+88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d72 –
……
```

请回答如下问题。

1) 描述函数调用和返回指令的执行过程。
2) ucore 中的函数调用参数是如何从调用函数（caller）传递给被调用函数（callee）的。
3) 补全函数调用堆栈跟踪函数 print_stackframe。

=========/kern/debug/kdebug.c=========

```c
#include <defs.h>
#include <x86.h>
#include <stab.h>
#include <stdio.h>
#include <string.h>
#include <kdebug.h>
#define STACKFRAME_DEPTH 20
extern const struct stab __STAB_BEGIN__[];  // beginning of stabs table
extern const struct stab __STAB_END__[];    // end of stabs table
extern const char __STABSTR_BEGIN__[];      // beginning of string table
extern const char __STABSTR_END__[];        // end of string table
/* debug information about a particular instruction pointer */
struct eipdebuginfo {
    const char *eip_file;                   // source code filename for eip
    int eip_line;                           // source code line number for eip
    const char *eip_fn_name;                // name of function containing eip
    int eip_fn_namelen;                     // length of function's name
    uintptr_t eip_fn_addr;                  // start address of function
    int eip_fn_narg;                        // number of function arguments
};
/* *
 * stab_binsearch - according to the input, the initial value of
 * range [*@region_left, *@region_right], find a single stab entry
 * that includes the address @addr and matches the type @type,
 * and then save its boundary to the locations that pointed
 * by @region_left and @region_right.
 *
 * Some stab types are arranged in increasing order by instruction address.
 * For example, N_FUN stabs (stab entries with n_type == N_FUN), which
 * mark functions, and N_SO stabs, which mark source files.
 *
 * Given an instruction address, this function finds the single stab entry
 * of type @type that contains that address.
 *
 * The search takes place within the range [*@region_left, *@region_right].
 * Thus, to search an entire set of N stabs, you might do:
 *
 *      left = 0;
 *      right = N - 1;     (rightmost stab)
 *      stab_binsearch(stabs, &left, &right, type, addr);
 *
 * The search modifies *region_left and *region_right to bracket the @addr.
 * *@region_left points to the matching stab that contains @addr,
 * and *@region_right points just before the next stab.
 * If *@region_left > *region_right, then @addr is not contained in any
 * matching stab.
 *
 * For example, given these N_SO stabs:
 *      Index  Type   Address
 *      0      SO     f0100000
 *      13     SO     f0100040
 *      117    SO     f0100176
 *      118    SO     f0100178
 *      555    SO     f0100652
 *      556    SO     f0100654
 *      657    SO     f0100849
 * this code:
 *      left = 0, right = 657;
 *      stab_binsearch(stabs, &left, &right, N_SO, 0xf0100184);
 * will exit setting left = 118, right = 554.
 * */
static void
stab_binsearch(const struct stab *stabs, int *region_left, int *region_right,
           int type, uintptr_t addr) {
    ……
}
/* *
 * debuginfo_eip - Fill in the @info structure with information about
 * the specified instruction address, @addr.  Returns 0 if information
 * was found, and negative if not.  But even if it returns negative it
 * has stored some information into '*info'.
 * */
int
debuginfo_eip(uintptr_t addr, struct eipdebuginfo *info) {

…..
}
/* *
 * print_kerninfo - print the information about kernel, including the location
 * of kernel entry, the start addresses of data and text segements, the start
 * address of free memory and how many memory that kernel has used.
 * */
void
print_kerninfo(void) {
    extern char etext[], edata[], end[], kern_init[];
    cprintf("Special kernel symbols:\n");
    cprintf("  entry  0x%08x (phys)\n", kern_init);
    cprintf("  etext  0x%08x (phys)\n", etext);
    cprintf("  edata  0x%08x (phys)\n", edata);
    cprintf("  end    0x%08x (phys)\n", end);
    cprintf("Kernel executable memory footprint: %dKB\n", (end - kern_init + 1023)/1024);
}
```

```c
/* *
 * print_debuginfo - read and print the stat information for the address @eip,
 * and info.eip_fn_addr should be the first address of the related function.
 * */
void
print_debuginfo(uintptr_t eip) {
    struct eipdebuginfo info;
    if (debuginfo_eip(eip, &info) != 0) {
        cprintf("    <unknow>: -- 0x%08x --\n", eip);
    }
    else {
        char fnname[256];
        int j;
        for (j = 0; j < info.eip_fn_namelen; j ++) {
            fnname[j] = info.eip_fn_name[j];
        }
        fnname[j] = '\0';
        cprintf("    %s:%d: %s+%d\n", info.eip_file, info.eip_line,
                fnname, eip - info.eip_fn_addr);
    }
}
static __noinline uint32_t
read_eip(void) {
    uint32_t eip;
    asm volatile("movl 4(%%ebp), %0" : "=r" (eip));
    return eip;
}
/* *
 * print_stackframe - print a list of the saved eip values from the nested 'call'
 * instructions that led to the current point of execution
 *
 * The x86 stack pointer, namely esp, points to the lowest location on the stack
 * that is currently in use. Everything below that location in stack is free. Pushing
 * a value onto the stack will invole decreasing the stack pointer and then writing
 * the value to the place that stack pointer pointes to. And popping a value do the
 * opposite.
 *
 * The ebp (base pointer) register, in contrast, is associated with the stack
 * primarily by software convention. On entry to a C function, the function's
 * prologue code normally saves the previous function's base pointer by pushing
 * it onto the stack, and then copies the current esp value into ebp for the duration
 * of the function. If all the functions in a program obey this convention,
 * then at any given point during the program's execution, it is possible to trace
 * back through the stack by following the chain of saved ebp pointers and determining
 * exactly what nested sequence of function calls caused this particular point in the
 * program to be reached. This capability can be particularly useful, for example,
 * when a particular function causes an assert failure or panic because bad arguments
 * were passed to it, but you aren't sure who passed the bad arguments. A stack
 * backtrace lets you find the offending function.
 *
 * The inline function read_ebp() can tell us the value of current ebp. And the
 * non-inline function read_eip() is useful, it can read the value of current eip,
 * since while calling this function, read_eip() can read the caller's eip from
 * stack easily.
 *
 * In print_debuginfo(), the function debuginfo_eip() can get enough information about
 * calling-chain. Finally print_stackframe() will trace and print them for debugging.
 *
 * Note that, the length of ebp-chain is limited. In boot/bootasm.S, before jumping
 * to the kernel entry, the value of ebp has been set to zero, that's the boundary.
 * */
void
print_stackframe(void) {
     /* LAB1 YOUR CODE : STEP 1 */
    /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
     * (2) call read_eip() to get the value of eip. the type is (uint32_t);
     * (3) from 0 .. STACKFRAME_DEPTH
     * (3.1) printf value of ebp, eip
     * (3.2) (uint32_t)calling arguments [0..4] = the contents in address (unit32_t)ebp +2 [0..4]
     * (3.3) cprintf("\n");
     *(3.4) call print_debuginfo(eip-1)to print the C calling function name and line number, etc.
     *(3.5) popup a calling stackframe
     *           NOTICE: the calling funciton's return addr eip  = ss:[ebp+4]
     *                   the calling funciton's ebp = ss:[ebp]
     */
}
```

六、（15 分）中断（Interrupt）是操作系统为处理意外事件而提供的一种响应机制，中断可分为硬件中断（Hardware interrupt）和软件中断（software interrupt）。中断响应需要硬件和软件的协调合作来完成。在虚拟机中的中断响应需要宿主机（Host OS）、虚拟机（Guest OS）和硬件的协调合作来完成。试回答下面问题。

1) 描述硬件中断、软件中断和系统调用（system call）的区别。
2) 简要描述外部中断的响应处理过程，并说明各处理操作的执行者。
3) 简要描述虚拟机中客户操作系统对硬件中断的响应处理过程。