第五章. 抢占式调度（lab3）(v0.1)

5.1. 实验目标

MIT 这次实验是在 Lab2 的基础上实现，目标是在他们的 JOS 操作系统中实现进程管理和中断的功能。

程序的几乎所有代码都集中在 env.c 和 trap.c 文件中。实际上，该实验可以分为 2 部分：进程环境和中断处理（包括系统调用）。前者通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现如何对进程进行管理；后者通过设置中断描述符表 IDT，编写通用中断派发程序，实现如何管理中断等。

在进程管理上，需要完成的函数包括：

env_init()

env_setup_vm()

segment_alloc()

load_icode()

env_create()

env_run()

在中断处理中，需要完成的函数包括：

trapentry.S 中相关代码

trap_dispatch()

idt_init()

syscall()

在本次实验过程中，并没有像上次实验的检查函数，因为进程运行有很多意想不到的错误，在实验中多使用 panic 或者 cprintf 打印一下相关信息，如果程序不符合实验原来的设想的话，只能自己找问题所在了。即使只写了一小段程序，都可以打印一下相关信息，以确保走的是正确的路。

5.2. 背景知识

通过实验 2，我们知道，JOS 在启动之后开启了内存分页管理，在实验 2 的主要函数 i386_init()函数之后，并没有进入循环，而是相应的对进程结构初始化和中断初始化，i386_init()函数最后会调用 env_run(&envs[0]);运行一个进程。一个进程的执行不能对内核（kernel）和其他进程产生干扰，当进程执行特权指令时，需要处理器产生中断，从用户态切换到内核态，完成任务后中断返回到用户态。

在本实验中，JOS 只创建了一个进程，但是我们需要让 JOS 支持多进程（像现代操作系统那样），这在实验四中允许一个进程创建其他进程。

5.3. 进程管理的实现

在这个实验中，主要是建立合理的数据结构对进程进行管理，并对进程分配空间，然后将给定的 2 进制形式存储的 elf 文件重新映射到进程空间中分配好的内存空间里，运行初始化完毕的进程。

5.3.1 进程管理

在 JOS 系统中，对于进程的管理和对物理页面的管理是一样的，也是通过双向链表进行管理的。

在讨论进程管理之前，有几个常识性的问题需要先澄清一下。首先，进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位，又称任务（Task）。系统为了管理进程设置一个专门的数据结构：进程控制块（PCB：Process Control Block)，用它来记录进程的外部特征，描述进程的运动变化工程（又称进程描述符）。

下面，我们分别对进程管理和初始化进行讨论。

5.3.1.1 进程的结构

首先，我们来讨论用于表示进程的数据结构。实验中进程的结构是在 inc/env.h 中规定的，每一个 Environment 对应一个进程：

```
struct Env {
	struct Trapframe env_tf; // Saved registers
	LIST_ENTRY(Env) env_link;	// Free list link pointers
	envid_t env_id;			// Unique environment identifier
	envid_t env_parent_id;		// env_id of this env's parent
	unsigned env_status;		// Status of the environment
	uint32_t env_runs;		// Number of times environment has run

	// Address space
	pde_t *env_pgdir;		// Kernel virtual address of page dir
	physaddr_t env_cr3;		// Physical address of page dir
};
```

同实验 2 中内存管理一样，进程也用双向链表结构进程管理，其中链表结构可以参看实验 2 文档中的 3.1.1：

```
LIST_ENTRY(Env) env_link;	// Free list link pointers
```

而 LIST_ENTRY 的定义则在 queue.h 中：

```
#define	LIST_ENTRY(type)				\
struct {						\
	struct type *le_next;	/* next element */		\
	struct type **le_prev;	/* ptr to ptr to this element */ \
}
```

通过分析，我们可以写出 Env 结构：

struct Env {

```
        struct Trapframe env_tf;
        struct {
                struct Env *le_next;
                struct Env **le_prev;
        }env_link;
        envid_t env_id;
        envid_t env_parent_id;
        unsigned env_status;
        uint32_t env_runs;

        pde_t *env_pgdir;
        physaddr_t env_cr3;
};
```

所以该结点的结构可以用图 5-1 来表示：

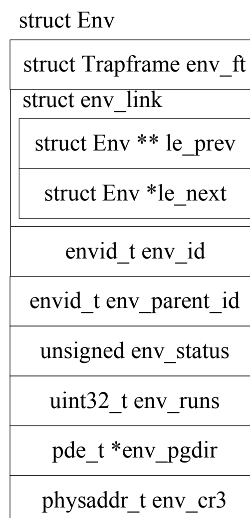

图 5-1. 进程管理双向链表结点结构

该结构存在 9 个成员：一个 CPU 现场保护信息 env_tf（保存进程切换时此进程的寄存器的信息）、指向链表下一个结点的指针、一个指向指针的指针、两个进程描述信息（PID）、进程控制信息 env_status&&env_runs 以及地址空间 env_pgdir&&env_cr3。其中，寄存器结构信息是在 int/trap.h 中规定的；进程有三个状态，在 kern/env.h 中规定：FREE，RUNNABLE，NOT_RUNNABLE；进程的地址空间 env_pgdir 保存进程页表的虚拟地址，env_cr3 保存进程页表的物理地址。

另外，系统还为进程管理定义了一个链表头，同实验 2 中 3.1.1 中结构相同，使用相同的一套宏将系统所有的 PCB 组织在一起，并把它们放在内存的固定区域，即 PAGES 页面管理结构所占用的空间之后的 4KB 空间。

该链表头的定义是通过两个宏来完成的，它们分别是 kern/env.h 中的：

```
LIST_HEAD(Env_list, Env);        // Declares 'struct Env_list'
```

和 queue.h 中的：

```
#define    LIST_HEAD(name, type)                                    \
struct name {                                        \
```

struct type *lh_first;   /* first element */   \
}
以及在 env.c 中定义的全局变量：
static struct Env_list env_free_list;// Free list
通过分析，我们发现 Page_list 结构实际上可以写成：
struct Env_list {
struct Env *lh_first;
};
此链表结构是和 Page 结构一样的，只包含了一个指向 Env 结构的指针 lh_first。同时，env_free_list 这个全局变量实际上就是传说中的指向进程管理双向链表的头结构了，其保存没有运行的进程。

同时，系统还定义了一些宏来对这个链表头进行操作，这些宏已经在实验 2 的 3.1.1 中进行了描述，此处就不重复了。

进程管理链表主要就通过一系列宏搭建起来，如图 5-2 所示：

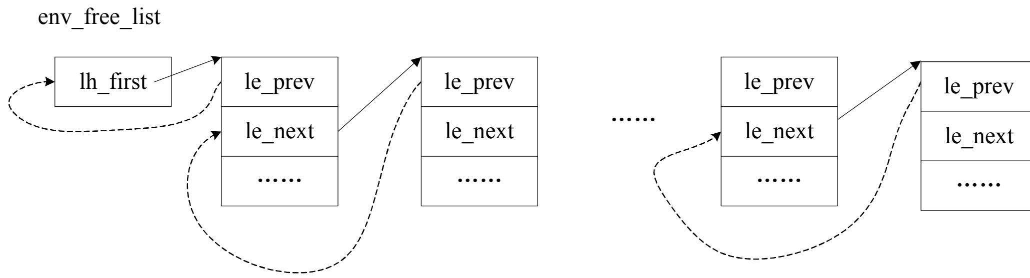

图 5-2. 进程管理链表结构

注意，图中实线和虚线表示的意义是不完全一样的。

5.3.1.2 进程管理链表在内存中的存储和放置

实验 2 中，在 i386_vm_init(void)函数中，我们通过 boot_alloc 函数为页面管理链表中的众多管理结点分配空间，同样，我们也用这个函数为进程管理链表中的众多管理结点分配空间。

static void* boot_alloc(uint32_t n, uint32_t align)
在调用这个函数的时候，传入的参数为：
n = ROUNDUP((sizeof(struct Env) * NENV), PGSIZE)
align = PGSIZE (4KB);

即在全局变量 boot_freemem 等于 Pages 装入后的最末尾的位置，然后将它往上（高地址）移到 align（这里是 4KB）对齐的位置，并从这里分配空间，分配完了以后，boot_freemem 继续望上移，并将前一个 align 对齐的位置返回回去。所以，我们的进程管理链表实际上存储在紧接着 Pages 的内存的“相对高端”的地方。这个空间的位置如图 5-3 所示。

同时，需要说明的是，系统定义了两个全局变量：
struct Env *envs = NULL;   /* All environments */
struct Env *curenv = NULL;   /* the current env */
boot_alloc()函数返回的地址就赋值到 envs 变量里。

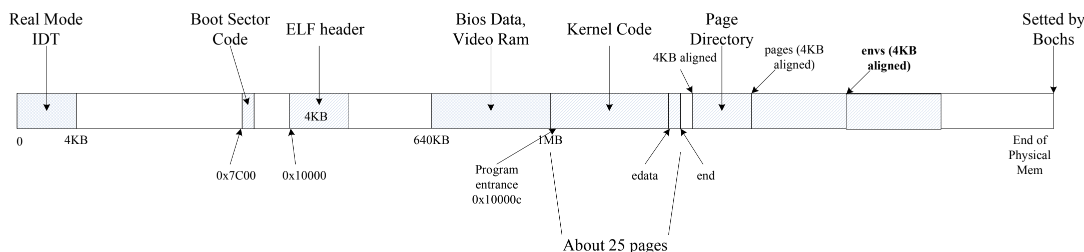

图 5-3. 页面管理空间的放置

对于 envs 的使用，与实验 2 中 pages 的使用方法是一样的，即对于进程管理和组织至少有 2 种方法：其 1 是通过 envs[下标]，另一种是通过双向链表。不清楚的可以参考实验 2 中 3.1.2。对于 curenv 它总是指向当前运行的进程，在进程切换时需要用到。

注意，一般操作系统中都可以多进程并发执行的，这取决与 PCB 表的大小，在 JOS 系统中，evns 数组就等价于 PCB 表，其共有 1024（NENV）个表项，即 JOS 系统并发度为 1024。其相关宏在 inc/Env.h 中定义：

```
#define LOG2NENV          10
#define NENV              (1 << LOG2NENV)
#define ENVX(envid)       ((envid) & (NENV - 1))
```

其中 NENV 的宏定义为 1 左移 10 位，即 2^10 为 1024，用来说明系统中最多控制 1024 个用户进程。

5.3.2 进程初始化

在本节中，主要讨论 JOS 系统在运行用户进程前所做的准备工作。

首先，在 3.1.2 中为进程管理分配完内存后，需要将进程地址按 JOS 的线性地址规划进行定位，在 kern/Pmap.c 中使用函数 **boot_map_segment**，将 envs 结构体的物理地址映射到从 UENV 所指向的线性地址空间，此时因为该线性地址空间用户允许访问，所以页面权限被标记为只读。即

[UENVS, sizeof(envs)] => [envs, sizeof(envs)]

这里 UENVS 代表进程管理结构所占用的空间；

5.3.2.1 进程映像的说明

在 JOS 中，每个进程都有代码段、数据段、用户栈、进程属性，由于当前系统还没有文件系统，系统将用户进程编译链接成原始的 ELF 二进制映像内嵌到内核中，所以系统加载一个用户进程对应的代码和数据时读取的对象是 ELF 格式文件。其编译命令可以参看 kern/Makefrag 文件。

5.3.2.2 进程的实现

在 JOS 中，每个进程都有 4GB 的虚拟地址空间，其中的内核部分都是相同的。通过以上我们对 JOS 中进程的了解，系统仅仅为进程分配了内存空间，并没有实际的运行用户进程，我们需要继续完成 JOS 系统中对进程进行操作的所有函数，其中包括：

env_init()

env_setup_vm()

segment_alloc()

load_icode()
env_create()
env_run()

下面对这些函数进行说明。

void **env_init**(void)

该函数主要用来初始化所有 NENV 个 Env 结构体，将它们加入到空闲队列中，作为可用进程控制块，具体操作和实验 2 的 page_free_list 类似，不赘述。

注意：该函数将结构体插入空闲队列时，以反序的方式插入，即从高到低的顺序！

static int **env_setup_vm**(struct Env *e)

该函数主要用来为当前用户进程环境 e 分配控制结构的页目录。通过 pgdir_walk（实验 2 中有说明）分配一页，然后将当前进程的 env_cr3 字段指向该页的物理地址，将当前进程的 env_pgdir 字段指向该页的逻辑地址。

注意：分配完页面之后，要对该页的逻辑地址所指向的内存清 0，因为这块地址就是用来控制该进程所有内存页式分配的页目录。另外，对于 UTOP 以上的虚拟地址空间对于所有的进程来说都是相同的，用户进程无法访问和使用这一部分虚拟地址空间，所以只要将这部分地址空间的映射关系由 boot_pgdir[]中完全拷贝到 e->env_pgdir[]中即可。

static void **segment_alloc**(struct Env *e, void *va, size_t len)

该函数主要用于给进程分配内存空间，用于存放进程运行所需资源。只要调用实验 2 中的 page_insert 函数就可以实现，代码简单，不赘述。

注意：此处分配的空间可以是根据 env_pgdir[]页目录来控制分配，可以由虚拟地址找到分配的页面，而不是根据 boot_pgdir 在 kernel 的地址空间中分配。

static void **load_icode**(struct Env *e, uint8_t *binary, size_t size)

该函数比较复杂，也是初始化用户进程空间的最重要的函数之一，该函数的主要功能是将给定的 2 进制形式存储的 elf 文件重新映射到进程空间中分配好的那一部分内存空间里，可参照 boot/Main.c 中 bootmain 函数完成。

具体代码如下：

```
struct Elf *env_elf;
struct Proghdr *ph, *eph;
env_elf = (struct Elf*)binary;
ph = (struct Proghdr*)((uint8_t*)(env_elf) + env_elf->e_phoff);
eph = ph + env_elf->e_phnum;
for (; ph < eph; ph++)
{
	if(ph->p_type == ELF_PROG_LOAD)
	{
		segment_alloc(e,(void*)ph->p_va,ph->p_memsz);
		memmove((void*)ph->p_va,binary+ph->p_offset,ph->p_filesz);
		memset((void*)(ph->p_va + ph->p_filesz),0,ph->p_memsz-ph->p_filesz);
	}
}
```

e->env_tf.tf_eip = env_elf->e_entry;

其中，从 uint8_t*结构的 binary 指针加上 env_elf 给定的偏移量中可以计算出 binary 中的所有段被加载到内存中的起始虚拟地址 va：ph->p_va，该地址是起始地址，所以在之后首先对于当前进程在该虚拟地址处分配空闲的物理页面，然后将指定大小的数据复制到从这一地址开始的内存中，然后同时必须改变当前进程的 eip 指针，将其指向这一片读入的代码的入口地址：env_elf->e_entry，这样才能根据设置好的 CS 与新的偏移量 eip 找到用户进程需要执行的代码。

void **env_create**(uint8_t *binary, size_t size)

该函数的任务非常明确，只要先调用 env_alloc 创建好进程的地址空间页目录，然后再调用 load_icode 将进程运行所需要的代码装入进程的地址空间中即可。

void **env_run**(struct Env *e)

该函数的主要功能是运行初始化完毕的进程，所以在加载新的进程的cr3 寄存器之前（重定位 CS 与 eip），必须要将原先设置好的 es、ds 和 esp 入栈，防止在此过程中被破坏（调用函数 **env_pop_tf**()，完成进程上下文切换），加载完毕之后重置这些寄存器，然后用户进程就在新的代码段中开始执行用户的程序了。

注意：运行到这里，用户进程试图访问一个系统调用，因为我们还没有完成系统调用，这里系统会 crash。

5.3.2.3 用户进程运行前的调用图

JOS 的启动流程如下所示：

- start (kern/entry.S)
- i386_init
  - cons_init
  - i386_detect_memory
  - i386_vm_init
  - page_init
  - env_init
  - idt_init (still incomplete at this point)
  - env_create
  - env_run
    - env_pop_tf

图 5-4. 用户进程运行前的调用图

以上是用户代码执行前的系统流程。

5.4. 中断与异常管理

在系统运行到 env_run 之后，用户进程会遇到 int $0x30 系统调用，开始中断之旅。在 JOS 系统中，我们发现，为了保证大家自己写的程序段的正确性，该实验安排了一些用户进程函数，以进行阶段性地检查，如果这些检查函数发现大家写的程序不符合实验原来的设想的话，就会提前垮掉，一般是出现 GP 错误。每完成一小段程序，用相应的用户进程对程序进行一下检查，以确保走的是正确的路。其用户函数主要在 **user** 目录下。

我们为了完成基本的中断和系统调用，让处理器从用户态回到内核态，要用到保护控制转移机制，介绍如下：

5.4.1 保护控制转移机制

一个进程的执行不能对内核和其他进程产生干扰，当用户进程需要执行特权指令或者内核功能时，需要一个系统调用，使处理器从用户态切换到内核态，处理完毕返回用户态。中断和异常可以完成这些功能，他们是保护控制转移机制。在 x86 体系下，保护控制转移由两个特殊的机制实现：**IDT（中断描述符表）**和 **TSS（任务状态段）**。

5.4.1.1 IDT（中断描述符表）

IDT 是将每一个中断向量和一个描述符对应起来的一个门描述符，就是每一个中断或异常处理程序的入口地址。中断向量到中断处理程序的对应过程如图 5-5 所示：

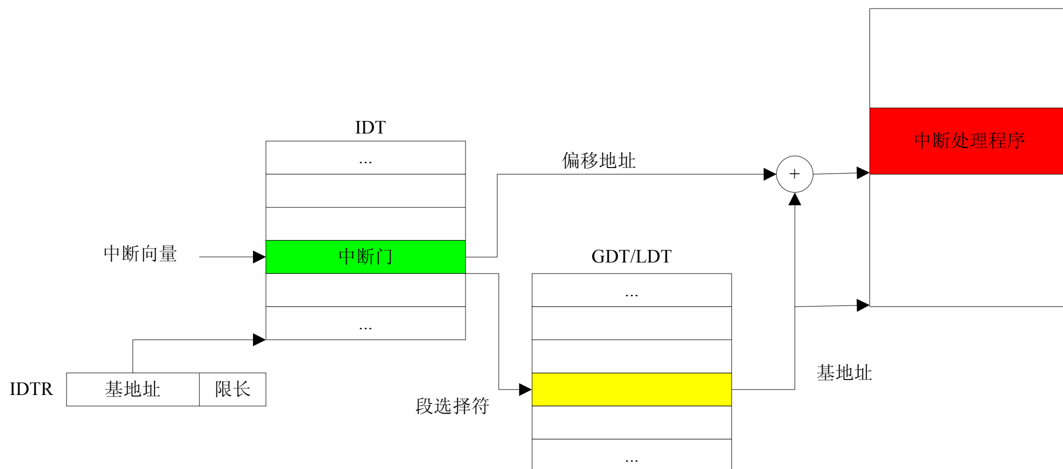

图 5-5. 中断处理过程

x86 允许 **256** 种不同的中断或者异常入口，每个中断和异常都由一个唯一的整数值表示，称之为**中断向量（Interrupt Vector）**。中断向量被 CPU 用来作为 IDT 的索引访问对应的门描述符，而中断门用来指向目标代码，即 Segment Selector + Offset，其中断门结构如图 5-6 所示：

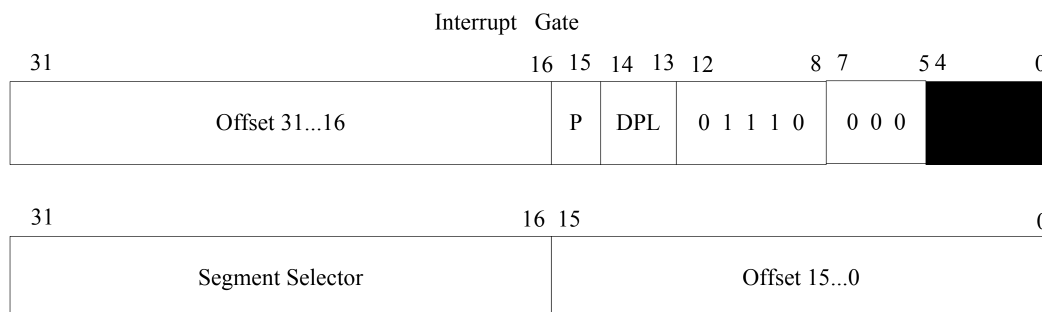

图 5-6. 中断门

其结构 **struct Gatedesc** 是在 inc/mmu.h 中定义的。由其结构可知，中断或异常也有特权级别，由中断门描述符中的 **DPL** 约束，门描述符中的段选择子中的 CPL 说明中断处理程序运行的特权级别。

5.4.1.2 TSS（任务状态段）

在现在操作系统中，TSS 是一个特殊的数据结构，一个任务的所有状态信息存储在其中。在处理器处理进程切换时需要 TSS 来保存旧的处理器状态，以便在中断返回时恢复以前的进程。当 x86 处理器发生中断或陷入时，切换到内核区域的一个堆栈也是由 TSS 指出的。TSS 结构是在 mmu.h 中规定，其结构可由图 5-7 表示：

I/O Map Base Address	T
	LDT Segment Selector
	GS
	FS
	DS
	SS
	CS
	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
CR3	
	SS2
ESP2	
	SS1
ESP1	
	SS0
ESP0	
	Previous Task Link

Reserved

图 5-7. TSS 结构

5.4.1.3 中断映射布局

在 x86 体系下，中断向量 0-31 为内部处理器异常，31 以上的中断仅被用于软件中断或者异步硬件中断。在文件 inc/trap.h 中规定了中断向量的类型，如下所示：

```
// Trap numbers
// These are processor defined:
#define T_DIVIDE     0      // divide error
#define T_DEBUG      1      // debug exception
#define T_NMI        2      // non-maskable interrupt
#define T_BRKPT      3      // breakpoint
#define T_OFLOW      4      // overflow
#define T_BOUND      5      // bounds check
#define T_ILLOP      6      // illegal opcode
#define T_DEVICE     7      // device not available
#define T_DBLFLT     8      // double fault
```

```
/* #define T_COPROC    9 */// reserved (not generated by recent processors)
#define T_TSS          10          // invalid task switch segment
#define T_SEGNP        11          // segment not present
#define T_STACK        12          // stack exception
#define T_GPFLT        13          // general protection fault
#define T_PGFLT        14          // page fault
/* #define T_RES       15 */   // reserved
#define T_FPERR        16          // floating point error
#define T_ALIGN        17          // aligment check
#define T_MCHK         18          // machine check
#define T_SIMDERR      19          // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL      48          // system call
#define T_DEFAULT      500         // catchall
```

由系统规定中断向量类型可知，JOS 系统中系统调用（T_SYSCALL）为 **int $0x30.**

5.4.1.4 中断实例分析

在这个例子中，我们分析发生在用户模式下的除零中断（int 0）。首先，我们应该注意的是：堆栈的切换。在内核模式下的异常处理中，需要将异常参数压到当前堆栈；而在用户模式下的异常/中断处理中，需要切换到 TSS 中 SS0，ESP0 所指的堆栈。

在除零中断中，我们需要切换到 **TSS 中 SS0（GD_KD），ESP0（KSTACKTOP）**所指的堆栈，然后在内核堆栈压入必要的信息，其结构如图 5-8 所示：

```
+--------------------+ KSTACKTOP
| 0x00000   old SS   |    " – 4
|      old ESP       |    " – 8
|    old EFLAGS      |    " – 12
| 0x00000 | old CS   |    " – 16
|      old EIP       |    " - 20 <---- ESP
+--------------------+
```

图 5-8. 压入信息之后的内核堆栈结构

然后设置 CS：EIP 使其指向 IDT 中 0 号中断对应中断门中保存的中断处理函数地址，其过程如 5.4.1.1 中的图 5 所示，在除零中断处理函数处理完该中断后返回用户进程。

对于某些 x86 异常，如缺页异常（page fault），除了上述 5 个字段外，有时会加上一个 **error code**，此时就会在内核堆栈多压入一个字 error code，其结构如图 9 所示：

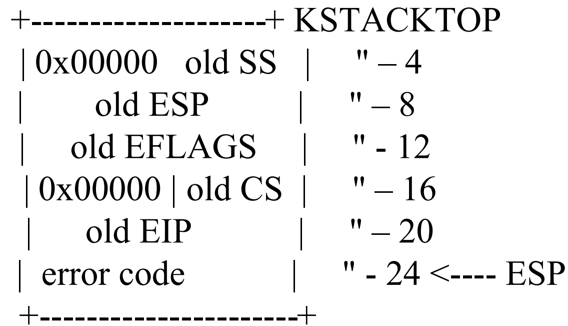

图 5-9. 带 error code 的内核堆栈结构

5.4.2 中断与异常的实现

在本节中，主要讨论 JOS 系统中中断与异常的具体实现，其代码主要集中在 trapentry.S 和 trap.c 文件。首先，我们先看一下整个中断过程的控制流，如图 5-10 所示：

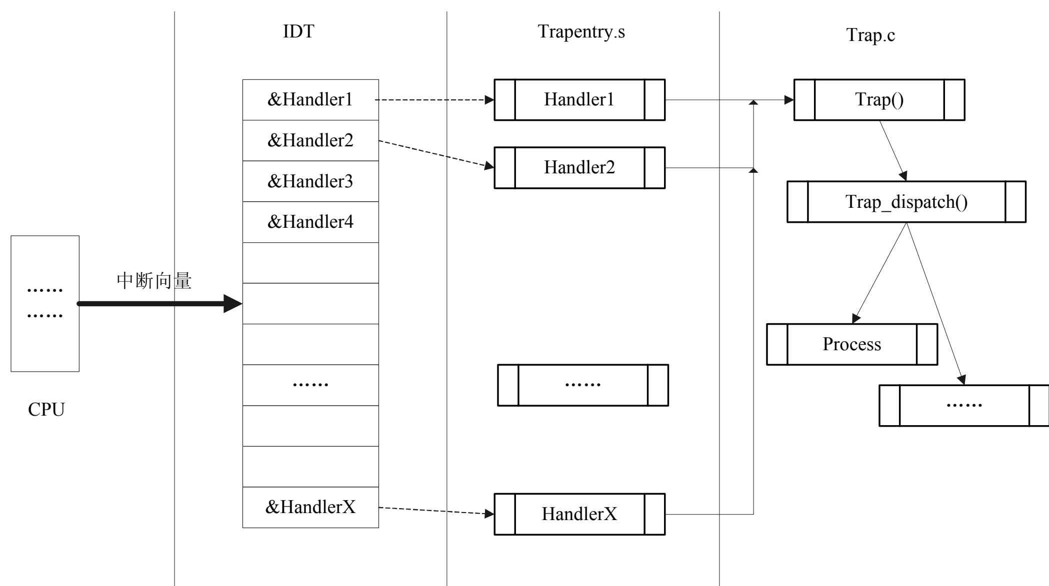

图 5-10. JOS 系统中断过程的控制流

我们根据上图来实现整个中断管理。

5.4.2.1 中断的初始化

现在我们来讨论如何初始化中断。首先，在 trapentry.S 中，我们需要判断出每一种 exception/interrupt 是否需要压入 error code，然后完成标号_alltraps 的部分。

关于中断的类型，需要阅读 Intel 手册中 0-31 号系统预留的中断类型，其类型如图 5-11 所示：

Vector	Description	Type	ErrorCode
0	Divide Error	Fault	No
1	Debug Exception	Fault/Trap	No
2	NMI Interrupt	Interrupt	No
3	Breakpoint	Trap	No
4	Overflow	Trap	No
5	Bound Check	Fault	No
6	Illegal Opcode	Fault	No
7	Device Not available	Fault	No
8	Double Fault	Abort	Yes
9	Reserved		
10	Invalid TSS	Fault	Yes
11	Segment Not Present	Fault	Yes
12	Stack Exception	Fault	Yes
13	General Protection Fault	Fault	Yes
14	Page Fault	Fault	Yes
15	Reserved		
16	Floating Point Error	Fault	No
17	Alignment Check	Fault	Yes
18	Machine Check	Abort	No
19	Simd Floating Point Error	Fault	No

图 5-11. 系统预留中断类型

由此可以得知需要所有中断的类型以及是否需要错误码，在 trapentry.S 中，有两个重要的宏可以帮助我们，对于需要错误码的中断采用宏 TRAPHANDLER 处理，该宏中会由 cpu 自动压入错误码；对于不需要错误码的中断采用宏 TRAPHANDLER_NOEC，在宏中压入常数 0 作为错误码。这两种宏的其他操作一致，向堆栈压入中断向量之后，跳转至标号为_alltraps 处继续执行余下指令。

对于_alltrap 处的处理，这是所有 trap handler 所共同执行的代码，在调用中断处理程序之前必须向栈中压入一个 struct Trapframe*变量，所以根据 Trapframe 的结构按照倒序向栈中压入所需的寄存器，注意到函数调用的时候对于结构体成员的寻址是按照首地址加上偏移量进行的，此处偏移量按照定义顺序依次增大，而向栈中压入变量时栈从高地址向低地址生长，所以倒序压入正好满足函数寻址时 Trapframe 结构的格式要求。其过程如图 5-12 所示：

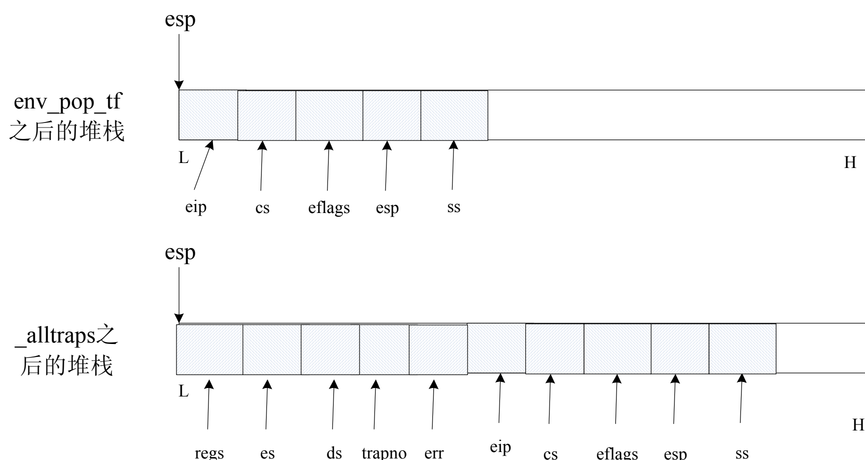

图 5-12. _alltraps 之后的堆栈

由图 5-10 我们知道，JOS 中处理中断时，每个中断都有一个 Handler，在_alltraps 之后，栈中所存的数据的格式满足了 Trapframe 的结构，就方便了 Handler 统一调用 trap（）函数；每个 Handler 会根据对应的中断是否有 error code 入栈来压入一个伪 error code 来保证压入堆栈的数据满足统一格式。

注意：完成这一步之后，中断初始化还没有完成，我们还要在 idt_init()中设置中断门描述符，建立 IDT，此处使用 int/mmu.h 中的宏 **SETGATE** 可以帮助我们完成。

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, dpl)
```

注意：因为中断都是在内核中完成的，其 sel 应设置为内核代码段。还有 JOS 只使用了 TSS 来设置正确堆栈：

```
// Setup a TSS so that we get the right stack
// when we trap to the kernel.
ts.ts_esp0 = KSTACKTOP;
ts.ts_ss0 = GD_KD;
```

5.4.2.2 缺页中断和断点异常

在 4.2 的开始，我们已经介绍了中断流程，在初始化完中断之后，就需要对相关中断进行处理了。

下面对主要函数进行说明。

static void **trap_dispatch**(struct Trapframe *tf)

该函数检查 Trapframe 结构中的中断向量号将中断分发到相应的中断处理过程。

在 **trap_dispatch** 函数中，遇到缺页中断（14 号中断 T_PGFLT）就分发到

page_fault_handler（）函数来处理；遇到断点异常（3 号中断 T_BRKPT）就分发到 **monitor()** 函数来处理。

5.4.2.3 系统调用

在 JOS 系统中，我们使用 **int $0x30** 指令引起处理器中断，完成系统调用。通过系统调用用户进程可以要求内核为其完成一些功能（只能内核才能完成的），当内核执行完相应代码返回用户进程继续执行。JOS 已经在 int/trap.h 中定义了常量 T_SYSCALL 为 0x30，我们需要将其加入到中断描述符中。

注意：我们设置 idt[0x30]的 **dpl** 时必须为 3，这样才允许用户调用。在 **int n** 的软中断调用时会检查 CPL 和 DPL 是否满足相应的级别关系，只有当前的 CPL 比要调用的中断的 DPL 大或者相等（数值上小）时才可以调用，否则就会产生 General Protection 中断。而我们当前处于用户状态，也就是说 CPL 此时为 3，为了能使用系统调用，需将系统中断的 DPL 定为 3。

JOS 系统的系统调用主要是在 kern/syscall.c 中函数 **syscall()**实现的。
int32_t **syscall**(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)

对于函数 syscall()，只要根据传入的参数中第一个参数 syscallno 来判定系统调用号，进而根据该号调用相应的程序。注意：在 inc/syscall.h 中定义了一个枚举结构，共有 SYS_cputs, SYS_cgetc, SYS_getenvid, SYS_env_destroy 这几种调用号，只要根据调用号调用正确的程序即可。在 syscall 函数中传入的参数总共有 6 个，第一个是调用号，后面 5 个是参数，调用相应的函数时各自需要的参数是按照顺序从这 5 个参数中依次选择的。

剩下的是，在 kern/trap.c 中的 trap_dispatch 函数，当发现当前中断向量为 SYS_CALL 时，在调用 syscall()函数之前，要传递相应的参数，仿照 lib/syscall.c 中的汇编代码，依次将当前进程控制块记录的 eax、edx、ecx、ebx、esi 与 edi 寄存器的值压入函数堆栈，然后将返回值刷新当前进程控制块中记录的 eax 值即可。

5.4.2.4 用户模式的开启

我们操作系统最终是要运行用户进程的，在 JOS 中，用户进程是 lib/entry.S 开始运行的，然后转入函数 **libmain()**，这部分内容中唯一需要注意的事情是，在 lib/libmain.c 中对于 env 的设置，在 entry.S 中 env 指向 UENV，现在需要更正为指向当前进程控制块的地址，对于全局数组 envs[]，我们只要根据当前进程的进程编号在 envs 里面进行搜索就可以找到当前的进程控制块。在进行搜索时要用宏 ENVX 对当前进程号进行处理，该宏的作用是保留进程号的低 10 位，以防在不同的进程控制块分配回收后进程编号超过了 NENV，造成数组越界。

直到这里，我们的进程 user/hello 才可以正常运行。

注意：内存保护可以确保用户进程中的 bugs 不能破坏其他进程或者内核。当用户进程试图访问一个无效的或者没有权限的地址时，处理器就会中断进程陷入到内核，如果错误可以修复，内核就修复它并让用户进程继续执行；如果无法修复，那么用户进程就不能继续执行。而在系统调用中导致了内存保护。许多系统调用接口运行把指针传给 kernel，这些指针指向用户 buffer，这就要防止一些恶意用户程序破坏内核，需要内核对用户传递的指针进行权限检查。对每个指针进行检查，由 **user_mem_check()**和 **user_mem_assert()**实现。检查用

户进程访存权限，并检查是否越界。

5.5. 常见错误

在完成实验 3 的过程中，我们遇见了许多错误，现在总结一下，希望以后做这个实验的人注意一下！！

1. 在实验 2 中的 i386_vm_init()函数中，当我们在将堆栈地址线性地址映射到物理地址时，其权限设置应该是 **PTE_U**，因为后续的用户进程也可以使用堆栈，如果换为其他，后面实验会引起不确定的错误！！
2. 也是实验 2 中页表权限的问题。在 pmap.c 的 pgdir_walk（）中，这个函数会为不存在的线性地址创建页表项，这时就需要注意权限设置，一定要或上 PTE_U!!，否则后面调试用户进程时，会出现 GP 错误！其代码如下所示。
   //ok, new physical page is allocated as well as the page structure
   pgdir[PDX(la)] = page2pa(free_page) | PTE_W | PTE_U | PTE_P;
   在实验 2 中，即使权限设置的有问题也不影响实验 2 的结果。
3. 在实验中，一定要仔细看说明，在 env_init()初始化时，其 env 结构是按倒序插入 evns[]结构数组的，在做实验时一定要细心，后面出错了就不好查了。
4. 在实验 3 的后半部分，设置中断时也要注意，在设置 IDT 时根据 Intel 手册，使用宏 SETGATE，其参数如下：
   //SETGATE(gate,istrap,sel,off,dpl);
   根据宏的定义，sel 一定要设置成 GD_KT，如果设置成 GD_KD,因为在 JOS 系统初始化的时候,把 GD_KD 设置成不可执行的!!! 所以即使设置了中断门，系统也不认为中断门所指向的代码是可执行的!如果设置不对，常会出现 GP 错误。