XV6 中断异常机制 阅读报告

本次代码阅读主要对 XV6 中的中断异常机制进行调研，主要涉及的代码有：traps.h，trap.c，trapsasm.S，vectors.pl，syscall.h，syscall.c，sysproc.c，usys.S。其中，traps.h 和 syscall.h 主要声明了与中断、异常有关的一些宏数值，其他文件为核心的处理程序。下面我将逐个剖析，以阐释 XV6 的中断异常机制。

注：代码阅读基于 XV6 的 x86 体系结构版本，对应于网站上提供的 xv6-rev7.pdf 的手册版本，可在 https://github.com/panks/Xv6 中找到源代码。（后续维护版本可能在函数封装上略有不同）

一、IDT 表的构建与初始化：

1.概述：

中断异常机制中，一个十分重要的数据结构便是中断向量表，它指明了中断处理程序的入口地址。XV6 中，采用 IDT 表项来维护这一重要内容。当系统初始化时（执行 main.c），系统会调用 trap.c 中的 tvinit 函数和 idtinit 函数（在 main.c 中的 mpmain 函数中），来构建 IDT 表，维护 IDT 寄存器（指向表头位置）。

2.中断向量的生成：

在进行上述初始化之前，我们首先应该获取 IDT 中的所有向量值，这一过程在 vectors.pl 中完成。这个文件是一个 perl 脚本，它生成 vectors.S 汇编文件，定义了每个中断的入口程序和入口地址，储存于 trap.c 中的 vectors 数组中。vectors.pl 的代码如下：

```
8  print "# generated by vectors.pl - do not edit\n";
9  print "# handlers\n";
10 print ".globl alltraps\n";
11 for(my $i = 0; $i < 256; $i++){
12     print ".globl vector$i\n";
13     print "vector$i:\n";
14     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
15         print "  pushl \$0\n";
16     }
17     print "  pushl \$$i\n";
18     print "  jmp alltraps\n";
19 }
20
21 print "\n# vector table\n";
22 print ".data\n";
23 print ".globl vectors\n";
24 print "vectors:\n";
25 for(my $i = 0; $i < 256; $i++){
26     print "  .long vector$i\n";
27 }
```

我们来看对每一个 vector 的处理。每一个向量对应于这样的一串汇编代码：首先，12 行中的.globl vector 告诉我们，每一个 vector 都是一个全局变量；13 行中指明标签 vector i；14-16 行是判断错误代码是否压入栈中（因为中断可以分为压入错误编码和不压入错误编码的两大类），如果属于不压入错误编码的中断，则把 0 作为替代压入栈中；然后，vectors[i]对应的汇编代码会把中断号压入栈（17 行）；最后，所有的中断都由统一的 alltraps 入口来进入（位于 trapasm.S 中）。

在声明完每一个 vector 跳转到中断入口之前要做的事情后，程序还会声明，每一个 vector 都是 long vector（21-27 行）。这样，所有中断向量的构造便执行完毕，入口标签的

地址偏移被储存在 vectors 数组中。

3.IDT 表项的生成：

接下来，系统会利用上述的向量内容，进一步构造 IDT 表。首先，在 tvinit 函数中，系统会基于 vectors 数组构建 IDT 表项。大致代码如下：

```
17  void
18  tvinit(void)
19  {
20    int i;
21
22    for(i = 0; i < 256; i++)
23      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25
26    initlock(&tickslock, "time");
27  }
```

这个函数利用一个循环，逐个生成 IDT 表项，维护在 idt 数组中。这个数组是一个 gatedesc 的结构数组，它声明于 mmu.h 中，具体内容如下：

```
147 // Gate descriptors for interrupts and traps
148 struct gatedesc {
149   uint off_15_0 : 16;   // low 16 bits of offset in segment
150   uint cs : 16;         // code segment selector
151   uint args : 5;        // # args, 0 for interrupt/trap gates
152   uint rsv1 : 3;        // reserved(should be zero I guess)
153   uint type : 4;        // type(STS_{IG32,TG32})
154   uint s : 1;           // must be 0 (system)
155   uint dpl : 2;         // descriptor(meaning new) privilege level
156   uint p : 1;           // Present
157   uint off_31_16 : 16;  // high bits of offset in segment
158 };
```

可以看到，XV6 沿用了 x86 的门描述符结构，维护了偏移量的高 16 位、低 16 位、门描述符 DPL，以及其他信息。

在生成 IDT 时，函数调用了 SETGATE 宏，它位于 mmu.h 中，内容如下：

```
160 // Set up a normal interrupt/trap gate descriptor.
161 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
162 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
163 // - sel: Code segment selector for interrupt/trap handler
164 // - off: Offset in code segment for interrupt/trap handler
165 // - dpl: Descriptor Privilege Level -
166 //        the privilege level required for software to invoke
167 //        this interrupt/trap gate explicitly using an int instruction.
168 #define SETGATE(gate, istrap, sel, off, d)                \
169 {                                                         \
170   (gate).off_15_0 = (uint)(off) & 0xffff;                \
171   (gate).cs = (sel);                                      \
172   (gate).args = 0;                                        \
173   (gate).rsv1 = 0;                                        \
174   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
175   (gate).s = 0;                                           \
176   (gate).dpl = (d);                                       \
177   (gate).p = 1;                                           \
178   (gate).off_31_16 = (uint)(off) >> 16;                  \
179 }
180
181 #endif
182
```

其中，gate 代表 idt[i]，为待建立的中断门；istrap 用于区分中断与异常（陷入）；sel 为代码段选择器的索引值，这里用 SEG_CODE（值为 1）左移 3 位，表示内核代码段的一个选择器；off 为偏移量，储存在 vectors 数组中；d 代表权限等级，对于中断，级别为 0，表示内核程序才能引发这个门。

由于系统调用属于异常的处理，且可以由用户调用，循环结束后，系统调用对应的表项被进一步修饰。而其他的中断由系统管理，故创建表项时，dpl 位设置为 0（内核态）。T_SYSCALL 为系统调用号，值为 64，声明于 traps.h 中。

在 IDT 表项（门描述符数组）生成完毕后，函数还会调用 initlock 函数，为 IDT 表生成一个互斥锁 tickslock。互斥锁的结构在 spinlock.h 中声明，内容如下：

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

tvinit 生成的互斥锁的标志名称为“time”，状态为 0，表示这个锁被占用，cpu 为 0，表示占用这个锁的 cpu 标号。这个锁始终用来处理中断。初始化互斥锁完成后，整个 IDT 表项的构建也宣告结束。

4.IDTR 寄存器的初始化：

在构造完成 IDT 表后，我们还需要维护 IDTR 寄存器，以确定 IDT 表的起始地址，这一过程由 trap.c 的 idtinit 完成，在 main.c 的 mpmain 函数中被调用。它的内容如下：

```
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

这个函数调用了 x86.h 中的 lidt 函数，其内容如下：

```
static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

实际上，lidt 在汇编中是一个装载 LDTR 的指令，格式为：“lidt　48 位伪描述符”。这里的伪描述符包含 3 部分内容：size-1，p 的低 16 位，p 的高 16 位。在把它们存储到 pd 数组中后，函数调用内联汇编语言，把这些内容装载到 LDTR 中。

到现在，所有的初始化工作便全部完成，下面便可以开始执行中断和异常的处理了。

二、中断与异常的统一入口：

1.概述：

在 XV6 中，中断和异常的处理都会统一经过一个入口，再跳转到对应的处理程序处。这个入口是 trapasm.S 中的汇编代码，与 trap.c 中的 trap 函数。下面就让我来对这两部分代码进行简要的分析。

2.trapasm.S——汇编代码保存现场与恢复现场：

当中断或异常触发时，在进入这段汇编代码之前，系统往往会做出一些保护措施。以 int n 这一系统调用指令为例，这个指令会进行以下的操作：

（1）从 IDT 中获得第 n 个描述符，n 为汇编语言的参数

（2）检查%cs 中的 CPL，应<=门描述符 DPL

（3）如果目标段的选择符的 DPL<CPL，在 CPU 内部的寄存器中保存%esp 和%ss 的值

（4）从一个任务段描述符中加载%ss 和%esp

（5）将%ss、%esp、%eflags、%cs、%eip 分别压栈

（6）清除%eflags 的一些位

（7）设置%cs、%eip 为描述符中的值

上述过程结束后，内核栈的情况如下：

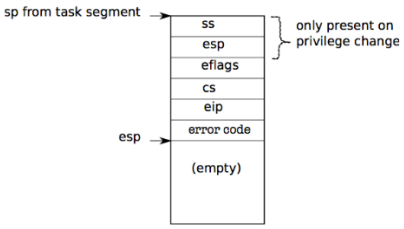

Figure 3-1. Kernel stack after an int instruction.

接下来，根据 vectors.pl 中的脚本输出对应的汇编代码，系统会把中断号也压入栈中，然后跳转到 trapasm.S 的 alltraps 标号下：

```
3   # vectors.S sends all traps here.
4 .globl alltraps
5 alltraps:
6   # Build trap frame.
7   pushl %ds
8   pushl %es
9   pushl %fs
10  pushl %gs
11  pushal
```

alltraps 继续保存处理器中的寄存器，它压入%ds、%es、%fs、%gs，以及通用寄存器。于是，内核栈中被压入了 trap frame（中断帧）结构，它包含了用于恢复用户进程的所有信息。内容如下：（其中，处理器压入%ss、%esp、%eflags、%cs、%eip，alltraps 压入余下内容）

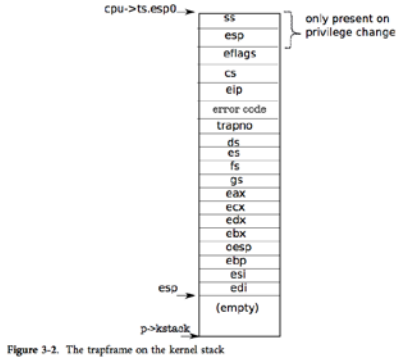

Figure 3-2. The trapframe on the kernel stack

接下来，汇编代码开始设置处理器，准备执行内核的 C 代码。alltraps 让%ds、%es 包含指向数据段的指针值；让%fs、%gs 包含指向 SEG_KCPU（CPU 数据段选择符）的值。

```
13    # Set up data and per-cpu segments.
14    movw $(SEG_KDATA<<3), %ax
15    movw %ax, %ds
16    movw %ax, %es
17    movw $(SEG_KCPU<<3), %ax
18    movw %ax, %fs
19    movw %ax, %gs
```

上述准备工作都进行完毕后，汇编程序便可调用 trap 函数了。它首先压入%esp 作为 trap 的参数（是刚建立好的中断帧的栈顶位置），然后调用 trap。trap 返回后，alltraps 弹出栈顶元素，然后执行后续的处理代码，为中断异常处理收尾。

```
21    # Call trap(tf), where tf=%esp
22    pushl %esp
23    call trap
24    addl $4, %esp
```

收尾工作是 trapret 标签下的代码。由于调用 trap 函数之前进行了额外的压栈操作，首先应该把对应的数据弹回原来的位置。然后把栈指针+8，这等价于把陷入好和错误代码同时弹出栈，最后执行 iret 命令，跳回用户空间，继续原来的程序执行。

```
26    # Return falls through to trapret...
27  .globl trapret
28  trapret:
29    popal
30    popl %gs
31    popl %fs
32    popl %es
33    popl %ds
34    addl $0x8, %esp  # trapno and errcode
35    iret
36
```

3.trap() in trap.c——C 程序执行中断/异常处理程序的跳转：

上面我们讲述了发生中断/异常时，处理器与汇编语言对现场（寄存器）的保护与恢复措施，那么接下来我们来一起走进 trap 函数，来观察系统是如何继续跳转到每个异常/中断所对应的特定入口的。

trap 函数要求传入 trap frame 的指针，这个结构在 x86.h 中被定义，其内容如下。可以看到，储存的内容即为上文图中的各寄存器值。

```
148  // Layout of the trap frame built on the stack by the
149  // hardware and by trapasm.S, and passed to trap().
150 struct trapframe {
151    // registers as pushed by pusha
152    uint edi;
153    uint esi;
154    uint ebp;
155    uint oesp;      // useless & ignored
156    uint ebx;
157    uint edx;
158    uint ecx;
159    uint eax;
160
161    // rest of trap frame
162    ushort gs;
163    ushort padding1;
164    ushort fs;
165    ushort padding2;
166    ushort es;
167    ushort padding3;
168    ushort ds;
169    ushort padding4;
170    uint trapno;
171
172    // below here defined by x86 hardware
173    uint err;
174    uint eip;
175    ushort cs;
176    ushort padding5;
177    uint eflags;
178
179    // below here only when crossing rings, such as from user to kernel
180    uint esp;
181    ushort ss;
182    ushort padding6;
183  };
```

进入 trap 函数后，函数首先判断传入的是中断还是系统调用。若为系统调用（T_SYSCALL），在真正地执行系统调用之前，函数会对当前进程的 kill 情况进行检测。如

果被 kill 掉，那么函数立即调用 exit 方法终止当前进程。系统调用 syscall 函数将在后面详细介绍。

```
39  if(tf->trapno == T_SYSCALL){
40    if(proc->killed)
41      exit();
42    proc->tf = tf;
43    syscall();
44    if(proc->killed)
45      exit();
46    return;
47  }
```

如果不是系统调用，那么接下来函数将根据中断的类型进行对应的中断处理。中断类型的宏定义数值储存于 traps.h 中。如下图所示：

```
1   // x86 trap and interrupt constants.
2
3   // Processor-defined:
4   #define T_DIVIDE         0      // divide error
5   #define T_DEBUG          1      // debug exception
6   #define T_NMI            2      // non-maskable interrupt
7   #define T_BRKPT          3      // breakpoint
8   #define T_OFLOW          4      // overflow
9   #define T_BOUND          5      // bounds check
10  #define T_ILLOP          6      // illegal opcode
11  #define T_DEVICE         7      // device not available
12  #define T_DBLFLT         8      // double fault
13  // #define T_COPROC      9      // reserved (not used since 486)
14  #define T_TSS           10      // invalid task switch segment
15  #define T_SEGNP         11      // segment not present
16  #define T_STACK         12      // stack exception
17  #define T_GPFLT         13      // general protection fault
18  #define T_PGFLT         14      // page fault
19  // #define T_RES        15      // reserved
20  #define T_FPERR         16      // floating point error
21  #define T_ALIGN         17      // aligment check
22  #define T_MCHK          18      // machine check
23  #define T_SIMDERR       19      // SIMD floating point error
24
25  // These are arbitrarily chosen, but with care not to overlap
26  // processor defined exceptions or interrupt vectors.
27  #define T_SYSCALL       64      // system call
28  #define T_DEFAULT      500      // catchall
29
30  #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
31
32  #define IRQ_TIMER        0
33  #define IRQ_KBD          1
34  #define IRQ_COM1         4
35  #define IRQ_IDE         14
36  #define IRQ_ERROR       19
37  #define IRQ_SPURIOUS    31
```

XV6 把中断类型筛选实现为 switch 语句的格式。我们可以根据代码格式把这些处理大致分为四大类：

（1）时钟中断的处理：

```
50  case T_IRQ0 + IRQ_TIMER:
51    if(cpu->id == 0){
52      acquire(&tickslock);
53      ticks++;
54      wakeup(&ticks);
55      release(&tickslock);
56    }
57    lapiceoi();
58    break;
```

当遇到时钟中断时，程序首先申请获取互斥锁，获取成功后，程序把 ticks 变量加 1，然后调用 wakeup 函数对休眠状态的进程进行检查。如果某些进程休眠时间已结束，那么 wakeup 函数会将其唤醒（从 SLEEPING 状态转换为 RUNNABLE 状态）。执行完毕后，释放互斥锁。最后调用 lapic.c 中的 lapiceoi 函数，以告知系统已经收到并处理完毕当前中断。

（2）一些主要中断的处理

```
59    case T_IRQ0 + IRQ_IDE:
60      ideintr();
61      lapiceoi();
62      break;
63    case T_IRQ0 + IRQ_IDE+1:
64      // Bochs generates spurious IDE1 interrupts.
65      break;
66    case T_IRQ0 + IRQ_KBD:
67      kbdintr();
68      lapiceoi();
69      break;
70    case T_IRQ0 + IRQ_COM1:
71      uartintr();
72      lapiceoi();
73      break;
```

在 XV6 源代码里，除时钟中断外，还有 4 种常见的异常被单独处理。它们分别是：磁盘外部中断、键盘外部中断、串口外部中断。三种中断的处理函数分别为 ide.c 的 ideintr 函数、kbd.c 的 kbdintr 函数、uart.c 中的 uartintr 函数。在分别调用完这几个函数后，程序仍然会调用 lapiceoi 函数，来告知中断处理完毕。

除此之外，对于 IRQ_IDE+1 的中断号，XV6 认为这是生成的一个 IDE1 伪中断，于是不做任何处理直接退出 switch 语句。

（3）非法中断的处理：

```
74    case T_IRQ0 + 7:
75    case T_IRQ0 + IRQ_SPURIOUS:
76      cprintf("cpu%d: spurious interrupt at %x:%x\n",
77              cpu->id, tf->cs, tf->eip);
78      lapiceoi();
79      break;
```

对于中断号 7，以及大于 31 的中断号，XV6 认为这些中断号为伪中断号，并打印相应的中断信息：CPU 的 id、%cs 的值、%eip 的值。

（4）其他中断的处理

```
82  default:
83    if(proc == 0 || (tf->cs&3) == 0){
84      // In kernel, it must be our mistake.
85      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
86              tf->trapno, cpu->id, tf->eip, rcr2());
87      panic("trap");
88    }
89    // In user space, assume process misbehaved.
90    cprintf("pid %d %s: trap %d err %d on cpu %d "
91            "eip 0x%x addr 0x%x--kill proc\n",
92            proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
93            rcr2());
94    proc->killed = 1;
95  }
```

首先检查是否在内核态中，为内核本身出错，如果是，程序打印错误信息（trap 号、CPU 的 id、栈中的%eip 值，以及%cr2 的寄存器值），然后调用 panic 函数，结束进程。

如果不是内核出错，那么 XV6 系统假定目前处在用户空间的进程出现了错误行为，在打印错误信息后设置进程 killed 的值，准备杀死当前进程。

在完成上述的处理以后，函数在退出之前，还会执行一些检查工作。

```
97    // Force process exit if it has been killed and is in user space.
98    // (If it is still executing in the kernel, let it keep running
99    // until it gets to the regular system call return.)
100   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
101     exit();
102
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
106     yield();
107
108   // Check if the process has been killed since we yielded
109   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
110     exit();
111 }
```

首先，如果进程被杀死（killed 值为 1）且处在用户空间中，那么杀死当前进程。但如果当前进程运行在内核态，让它运行至当前的系统调用结束。其次，如果当前进程为运行状态，且中断类型为时钟中断，那么调用 yield 函数，让当前进程放弃 CPU。最后，如果刚才的 yield 调用后进程被杀死了（killed 值为 1），那么杀死当前进程。

当上面的流程都执行完毕后，trap 函数退出，控制跳转回 trapasm.S 中的汇编语句，执行返回用户进程前的收尾处理。

三、系统调用的处理：

1.概述：

在前一部分中，我们看到，trap 函数调用 syscall 函数来执行系统调用。这一部分的代码主要被组织在 syscall.h、syscall.c、sysproc.c 这三个文件中。

其中，syscall.h 中包含着 XV6 中的 21 个系统调用的调用号，如下图所示。

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

syscall.c 包含执行系统调用的主体函数。在 XV6 中，系统调用由 int T_SYSCALL 指令执行，系统调用号应存放于%eax 中，参数位于栈中。这个文件中包含着 syscall 函数，它是所有系统调用的统一入口，而系统调用的函数体放于 sysproc.c 中。除了 syscall 函数之外，syscall.c 中还提供了一些工具函数，供一些系统调用的函数体和 sysfile.c 中的部分文件处理函数使用。另外，上面我们提到，在进入 trapasm.S 之前，系统会进行部分的寄存器保存，对于系统调用，由 int 指令来实现，XV6 把这些指令封装在了 usys.S 汇编文件中。下面让我们来逐个分析一下这些函数。

2.系统调用的入口——syscall 函数

```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

syscall 函数将系统调用分配到不同的处理函数。可以看到，系统首先从%eax 寄存器中获取系统调用号，然后判断调用号是否合法。若合法，执行系统调用数组 syscalls 中的对应条目，返回值存放于%eax 中；若不合法，输出错误信息，返回值为-1，存放于%eax 中。

syscalls 为 syscall.c 中存放的一个函数指针数组，其中的函数指针指向对应系统调用函数的入口地址，数组下标与系统调用号（in syscall.h）相对应。

```
102 static int (*syscalls[])(void) = {
103 [SYS_fork]    sys_fork,
104 [SYS_exit]    sys_exit,
105 [SYS_wait]    sys_wait,
106 [SYS_pipe]    sys_pipe,
107 [SYS_read]    sys_read,
108 [SYS_kill]    sys_kill,
109 [SYS_exec]    sys_exec,
110 [SYS_fstat]   sys_fstat,
111 [SYS_chdir]   sys_chdir,
112 [SYS_dup]     sys_dup,
113 [SYS_getpid]  sys_getpid,
114 [SYS_sbrk]    sys_sbrk,
115 [SYS_sleep]   sys_sleep,
116 [SYS_uptime]  sys_uptime,
117 [SYS_open]    sys_open,
118 [SYS_write]   sys_write,
119 [SYS_mknod]   sys_mknod,
120 [SYS_unlink]  sys_unlink,
121 [SYS_link]    sys_link,
122 [SYS_mkdir]   sys_mkdir,
123 [SYS_close]   sys_close,
124 };
```

```
80  extern int sys_chdir(void);
81  extern int sys_close(void);
82  extern int sys_dup(void);
83  extern int sys_exec(void);
84  extern int sys_exit(void);
85  extern int sys_fork(void);
86  extern int sys_fstat(void);
87  extern int sys_getpid(void);
88  extern int sys_kill(void);
89  extern int sys_link(void);
90  extern int sys_mkdir(void);
91  extern int sys_mknod(void);
92  extern int sys_open(void);
93  extern int sys_pipe(void);
94  extern int sys_read(void);
95  extern int sys_sbrk(void);
96  extern int sys_sleep(void);
97  extern int sys_unlink(void);
98  extern int sys_wait(void);
99  extern int sys_write(void);
100 extern int sys_uptime(void);
```

3.部分系统调用的具体实现——sysproc.c

上述列举的部分系统调用函数的实现存放于 sysproc.c 中，这里主要是与进程相关的系统调用的实现，与文件相关的系统调用位于 sysfile.c 中，等待阅读文件系统代码时再去分析它们的功能。下面让我们来逐个分析它们的功能。

（1）sys_fork

fork 函数的包装。收到该异常后，创建一个新子进程，并返回其 pid。

```
9  int
10 sys_fork(void)
11 {
12   return fork();
13 }
```

（2）sys_exit

exit 函数的包装。收到该异常后，调用 exit 函数，退出当前进程。

```
15 int
16 sys_exit(void)
17 {
18   exit();
19   return 0;  // not reached
20 }
```

（3）sys_wait

wait 函数的包装。收到该异常后，调用 wait 函数，等待子进程终止。

```
22 int
23 sys_wait(void)
24 {
25   return wait();
26 }
```

（4）sys_kill

kill 函数的包装。收到该异常后，首先检查 pid 是否小于 0。若小于 0，返回-1，认为

出错；否则，调用 kill 函数，杀死 pid 对应的进程。

```
28  int
29  sys_kill(void)
30  {
31    int pid;
32
33    if(argint(0, &pid) < 0)
34      return -1;
35    return kill(pid);
36  }
```

（5）sys_getpid

收到该异常后，返回当前进程的 pid。

```
38  int
39  sys_getpid(void)
40  {
41    return proc->pid;
42  }
```

（6）sys_sbrk

收到该异常后，系统准备扩展堆。首先检查扩展大小 n 是否小于 0，若小于 0 认为出错，返回-1，接下来用 addr 变量保存原来的堆指针，然后做 n 字节的扩展。若失败（返回值小于 0）则返回-1，否则返回原来的堆指针。

```
44  int
45  sys_sbrk(void)
46  {
47    int addr;
48    int n;
49
50    if(argint(0, &n) < 0)
51      return -1;
52    addr = proc->sz;
53    if(growproc(n) < 0)
54      return -1;
55    return addr;
56  }
```

（7）sys_sleep

收到该信号后，系统准备让当前进程睡眠。首先检查睡眠时长 n 是否大于 0，若小于 0，认为出错，返回-1.然后获取互斥锁，记录原来的 ticks 值（时长值），然后利用 while 循环等待时长 n。在这个过程中，如果进程被杀死，那么释放互斥锁，返回-1，否则持续调用 sleep 函数进行睡眠。结束 while 循环后，再释放互斥锁，正常返回 0。

```
58  int
59  sys_sleep(void)
60  {
61    int n;
62    uint ticks0;
63
64    if(argint(0, &n) < 0)
65      return -1;
66    acquire(&tickslock);
67    ticks0 = ticks;
68    while(ticks - ticks0 < n){
69      if(proc->killed){
70        release(&tickslock);
71        return -1;
72      }
73      sleep(&ticks, &tickslock);
74    }
75    release(&tickslock);
76    return 0;
77  }
```

（8）sys_uptime

收到该异常后，系统请求时间互斥锁，得到当前状态下的 clock tick 值，然后释放互斥锁，返回这个值。

```
// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
```

4.syscall.c 中的其他工具函数

（1）argint 与 fetchint

argint 返回 32 位系统中的第 n 个参数，由于调用时返回地址要入栈，因此额外+4 字节，越过返回地址。fetchint 函数在检查地址的合法性后向指针指向的地址写入对应的参数值。因为用户和内核共享一个页表，所以 fetchint 可以直接将地址值转换为指针。但内核要检验地址的合法性，以及是否在用户区内，地址超出用户区域便返回-1。这两个函数结合起来可以获取整数参数。

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
  if(addr >= proc->sz || addr+4 > proc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
```

（2）argptr

这个函数把第 n 个参数值转换为一个指针的值。首先，调用 argint 获取整数值，如果失败则返回-1。接下来检查这个整数值作为地址是否在用户区内，如果不是则返回-1；如果是则把这个地址存放到*pp 下，返回 0。

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

（3）argstr 与 fetchstr

argstr 函数把第 n 个参数转换为一个字符串指针。首先，函数调用 argint，获取第 n 个参数的整数值，若失败则返回-1。接下来调用 fetchstr 函数，获取字符串指针。

在 fetchstr 函数中，首先检查刚才获得的整数值作为地址是否在用户空间内，不在则返回-1。然后开始检查这个指针是否可以作为一个字符串指针。函数从这个指针出发，检查到用户区域的地址最大值，如果找到 0 值（字符串结尾的'\0'），就认为这是个合法的字符串，返回字符串长度，否则返回-1。虽然指针被赋值，但可以用返回值判断其合法性。

```
// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
```

```
// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= proc->sz)
    return -1;
  *pp = (char*)addr;
  ep = (char*)proc->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

5.usys.S——用于激发系统调用的入口

XV6 中的系统调用由 int 指令激发，这些指令存放于 usys.S 文件中。可以看到，文件声明了一个宏定义的汇编函数，其中的##代表连接。每段汇编代码首先把系统调用号移至%rax 中，然后调用 int 指令触发系统调用。int 触发后，会进行寄存器保护，然后跳转至 trapasm 入口执行。执行完毕后，ret 回原来的位置。

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

四、一些主要中断的处理：

在 trap 函数中，XV6 系统对时钟中断、磁盘中断、键盘中断、串口中断进行了处理，下面让我们来简要分析一下它们的处理方法。

1.时钟中断

XV6 把时钟芯片放置于局部 APIC 中，因此每个处理器可以独立地接收时钟中断。

当遇到时钟中断时，trap 函数会在把 clock tick 加 1 后调用 wakeup 函数。这个函数位于 proc.c 中。它获取互斥锁后，把进程表中所有到达睡眠时间（检查 proc 结构中的 chan 变量）的进程唤醒（从 SLEEPING 变成 RUNNABLE），然后释放互斥锁（进程表和时钟 clock tick 的两个锁），结束处理。

```
377  //PAGEBREAK!
378  // Wake up all processes sleeping on chan.
379  // The ptable lock must be held.
380  static void
381  wakeup1(void *chan)
382  {
383    struct proc *p;
384
385    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
386      if(p->state == SLEEPING && p->chan == chan)
387        p->state = RUNNABLE;
388  }
389
390  // Wake up all processes sleeping on chan.
391  void
392  wakeup(void *chan)
393  {
394    acquire(&ptable.lock);
395    wakeup1(chan);
396    release(&ptable.lock);
397  }
```

2.磁盘中断

当磁盘中断发生时，trap 会调用 ide.c 中的 ideintr 函数来处理它。这个函数查询队列中的第一个缓冲区，看它正在发生什么操作。如果该缓冲区正在被读入并且磁盘控制器有数据在等待，ideintr 就会调用 insl 将数据读入缓冲区。缓冲区就绪后，ideintr 设置 B_VALID，清除 B_DIRTY，唤醒等待这个缓冲区就绪的进程。最后，ideintr 将下一个等待中的缓冲区传递给磁盘，结束整个处理过程。

```
90   // Interrupt handler.
91   void
92   ideintr(void)
93   {
94     struct buf *b;
95
96     // First queued buffer is the active request.
97     acquire(&idelock);
98     if((b = idequeue) == 0){
99       release(&idelock);
100      // cprintf("spurious IDE interrupt\n");
101      return;
102    }
103    idequeue = b->qnext;
104
105    // Read data if needed.
106    if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
107      insl(0x1f0, b->data, 512/4);
108
109    // Wake process waiting for this buf.
110    b->flags |= B_VALID;
111    b->flags &= ~B_DIRTY;
112    wakeup(b);
113
114    // Start disk on next buf in queue.
115    if(idequeue != 0)
116      idestart(idequeue);
117
118    release(&idelock);
119  }
```

3.键盘中断和串口中断

对于这两个中断，trap 分别会调用 kbdintr 函数和 uartintr 函数来处理，它们分别位于 kbd.c 和 uart.c 中。这两个函数的流程大致相当，都是调用 console.c 中的 consoleintr 函数来与控制台做交互处理。

```
73  void
74  uartintr(void)
75  {
76    consoleintr(uartgetc);
77  }
78
```

```
46  void
47  kbdintr(void)
48  {
49    consoleintr(kbdgetc);
50  }
51
```

4.中断控制器

在 XV6 中，系统提供了一个可编程的中断控制器（位于 picirq.c 中），还提供了 I/O 系统的 APIC（ioapic.c）和每个处理器上的局部 APIC（lapic.c）。可以看出，XV6 是为能够搭载多核处理器的主板设计的，每一个处理器都需要编程接收中断。

为了在单核处理器上也能够正常运行，XV6 也为 PIC 编程。在 picrq.c 中，程序使用 inb 和 outb 指令，来配置主 PIC 产生 IRQ 0 到 7，从 PIC 产生 IRQ 8 到 16。初始化时，xv6 配置 PIC 屏蔽所有中断。

在多核处理器上，xv6 必须编写 IOAPIC 和每一个处理器的 LAPIC。IOAPIC 维护了一张表，处理器可以通过内存映射 I/O 写这个表的表项，而非使用 inb 和 outb 指令。在初始化的过程中，xv6 将第 0 号中断映射到 IRQ 0，以此类推，然后把它们都屏蔽掉。不同的设备自己开启自己的中断，并且同时指定哪一个处理器接受这个中断。

五、小结：

通过以上的源代码分析，对于 XV6 系统的中断异常处理，我们可以归纳如下：

1. 在系统启动时，会调用 tvinit 函数和 idtinit 函数生成 IDT 表，填充 IDTR 寄存器，为后续的中断异常处理提供入口，这些入口信息由 vectors.pl 脚本生成的 vectors.s 来提供。

2.当系统遇到中断或者异常时，首先会进行一部分的现场保存（把部分寄存器的值压入栈，对于系统调用，由 int 指令执行，它们储存于 usys.S 中），然后跳转到 trapasm.S 汇编代码处，这段代码首先进一步进行寄存器压栈操作，生成 trap frame 栈帧结构，然后设置数据段和 per-CPU 段。当这些准备工作全部完成后，调用 trap 函数这一统一入口。系统调用结束后，trapasm.S 会恢复栈结构，然后返回用户进程。

3.如果传入的是系统调用，trap 函数会进一步转到 syscall 函数这一处理系统调用的统一入口，syscall 函数会根据异常号来选择下一步要调用的具体函数。

4.如果传入的是中断，XV6 会根据中断的类型，进行 switch 的处理。具体来说，XV6 明确了时钟中断、磁盘中断、键盘中断、串口中断的处理办法，对于其他中断，XV6 要么不作处理，要么打印信息。在最后，函数还会检查当前进程的状态，来决定是否杀死进程，或者是否让出处理器给其他进程。