日曜研究室

技術的な観点から日常を綴ります

[xv6 #28] Chapter 2 – Traps, interrupts, and drivers – Code: Interrupts

テキストの37〜39ページ

本文

マザーボード上のデバイスは、割り込みを生成することが出来、xv6はそれらの割り込みを制御するためにハードウェアをセットアップしなければならない。
デバイスのサポート無しでは、xv6は使い物にならないだろう。
ユーザはキーボードでタイプ出来ないし、ファイルシステムはディスクにデータを保管することが出来ないし、他にもたくさんの問題が起きる。
幸いにも、割り込みを追加したり、単純なデバイスをサポートする事は、難しくない。
今まで見てきたように、割り込みは、システムコールと例外のためのコードと同じもの流用することが出来る。

割り込みは、いつでもデバイスに生成され得るという事を除いて、システムコールに似ている。
マザーボード上には、デバイスに注意が必要なとき（例えばユーザがキーボードの文字をタイプしたとき）にCPUへ信号を送るハードウェアがある。
我々は、デバイスに割り込みを生成させ、CPUが受け取る割り込みを整備するためにプログラムしなければならない。

タイマーデバイスとタイマー割り込みを見てみよう。
我々は、カーネルが時間の経過を認識出来るようにするため、カーネルが複数のプロセスの間で時分割出来るようにするため、タイマーハードウェアに1秒間に100回の割り込みを生成して欲しい。
1秒間に100回という選択は、プロセッサが割り込みの制御に忙殺されてない間、適正な対話パフォーマンスを可能にする。

x86プロセッサそれ自身のように、PCのマザーボードは進化してきた。
そして、提供される割り込みの方法も同じく進化してきた。
初期のボードは、単純なプログラム可能な割り込みコントローラ（PICと呼ばれる）を持っていた。

それを管理するためのコードをpicirq.cで見つけることが出来るだろう。

picirq.c（一応載せときます）

```
01 // Intel 8259A programmable interrupt controllers.
02
03 #include "types.h"
04 #include "x86.h"
05 #include "traps.h"
06
07 // I/O Addresses of the two programmable interrupt controllers
08 #define IO_PIC1         0x20    // Master (IRQs 0-7)
09 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
10
11 #define IRQ_SLAVE       2       // IRQ at which slave connects to
   master
12
13 // Current IRQ mask.
14 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
15 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
16
17 static void
18 picsetmask(ushort mask)
19 {
20   irqmask = mask;
21   outb(IO_PIC1+1, mask);
22   outb(IO_PIC2+1, mask >> 8);
23 }
24
25 void
26 picenable(int irq)
27 {
28   picsetmask(irqmask & ~(1<<irq));
29 }
30
31 // Initialize the 8259A interrupt controllers.
32 void
33 picinit(void)
34 {
35   // mask all interrupts
36   outb(IO_PIC1+1, 0xFF);
37   outb(IO_PIC2+1, 0xFF);
38
39   // Set up master (8259A-1)
40
41   // ICW1:  0001g0hi
42   //    g:  0 = edge triggering, 1 = level triggering
43   //    h:  0 = cascaded PICs, 1 = master only
44   //    i:  0 = no ICW4, 1 = ICW4 required
45   outb(IO_PIC1, 0x11);
46
47   // ICW2:  Vector offset
48   outb(IO_PIC1+1, T_IRQ0);
49
50   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
51   //        (slave PIC) 3-bit # of slave's connection to master
52   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
53
54   // ICW4:  000nbmap
55   //    n:  1 = special fully nested mode
56   //    b:  1 = buffered mode
57   //    m:  0 = slave PIC, 1 = master PIC
```

```
58    //      (ignored when b is 0, as the master/slave role
59    //      can be hardwired).
60    //    a:  1 = Automatic EOI mode
61    //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
62    outb(IO_PIC1+1, 0x3);
63
64    // Set up slave (8259A-2)
65    outb(IO_PIC2, 0x11);                  // ICW1
66    outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
67    outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
68    // NB Automatic EOI mode doesn't tend to work on the slave.
69    // Linux source code says it's "to be investigated".
70    outb(IO_PIC2+1, 0x3);                 // ICW4
71
72    // OCW3:  0ef01prs
73    //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific
      mask
74    //    p:  0 = no polling, 1 = polling mode
75    //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
76    outb(IO_PIC1, 0x68);             // clear specific mask
77    outb(IO_PIC1, 0x0a);             // read IRR by default
78
79    outb(IO_PIC2, 0x68);             // OCW3
80    outb(IO_PIC2, 0x0a);             // OCW3
81
82    if(irqmask != 0xFFFF)
83      picsetmask(irqmask);
84  }
```

マルチプロセッサPCボードの登場によって、新しい割り込みの制御方法が必要とされた。
なぜなら、どのCPUも割り込みを制御するための割り込みコントローラを必要とし、プロセッサに対するルーチン割り込みの為の手法が存在すべきだからである。
この方法は、2つの部分から成る。
1つは、I/Oシステム部分（IO APIC, ioapic.c）、もう1つは、それぞれのプロセッサにアタッチする部分（ローカルAPIC, lapic.c）である。
xv6は、マルチプロセッサ用のボードのためにデザインされ、そしてどのプロセッサも割り込みを受け取るためにプログラムされる。

ioapic.c

```
01  // The I/O APIC manages hardware interrupts for an SMP system.
02  // http://www.intel.com/design/chipsets/datashts/29056601.pdf
03  // See also picirq.c.
04
05  #include "types.h"
06  #include "defs.h"
07  #include "traps.h"
08
09  #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
10
11  #define REG_ID     0x00  // Register index: ID
12  #define REG_VER    0x01  // Register index: version
13  #define REG_TABLE  0x10  // Redirection table base
14
15  // The redirection table starts at REG_TABLE and uses
16  // two registers to configure each interrupt.
17  // The first (low) register in a pair contains configuration bits.
18  // The second (high) register contains a bitmask telling which
```

```
19 // CPUs can serve that interrupt.
20 #define INT_DISABLED   0x00010000  // Interrupt disabled
21 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
22 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
23 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC
   ID)
24
25 volatile struct ioapic *ioapic;
26
27 // IO APIC MMIO structure: write reg, then read or write data.
28 struct ioapic {
29   uint reg;
30   uint pad[3];
31   uint data;
32 };
33
34 static uint
35 ioapicread(int reg)
36 {
37   ioapic->reg = reg;
38   return ioapic->data;
39 }
40
41 static void
42 ioapicwrite(int reg, uint data)
43 {
44   ioapic->reg = reg;
45   ioapic->data = data;
46 }
47
48 void
49 ioapicinit(void)
50 {
51   int i, id, maxintr;
52
53   if(!ismp)
54     return;
55
56   ioapic = (volatile struct ioapic*)IOAPIC;
57   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
58   id = ioapicread(REG_ID) >> 24;
59   if(id != ioapicid)
60     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
61
62   // Mark all interrupts edge-triggered, active high, disabled,
63   // and not routed to any CPUs.
64   for(i = 0; i <= maxintr; i++){
65     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
66     ioapicwrite(REG_TABLE+2*i+1, 0);
67   }
68 }
69
70 void
71 ioapicenable(int irq, int cpunum)
72 {
73   if(!ismp)
74     return;
75
76   // Mark interrupt edge-triggered, active high,
77   // enabled, and routed to the given cpunum,
78   // which happens to be that cpu's APIC ID.
79   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
80   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
81 }
```

lapic.c

```
001 // The local APIC manages internal (non-I/O) interrupts.
002 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
003
004 #include "types.h"
005 #include "defs.h"
006 #include "memlayout.h"
007 #include "traps.h"
008 #include "mmu.h"
009 #include "x86.h"
010
011 // Local APIC registers, divided by 4 for use as uint[] indices.
012 #define ID      (0x0020/4)   // ID
013 #define VER     (0x0030/4)   // Version
014 #define TPR     (0x0080/4)   // Task Priority
015 #define EOI     (0x00B0/4)   // EOI
016 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
017   #define ENABLE     0x00000100   // Unit Enable
018 #define ESR     (0x0280/4)   // Error Status
019 #define ICRLO   (0x0300/4)   // Interrupt Command
020   #define INIT       0x00000500   // INIT/RESET
021   #define STARTUP    0x00000600   // Startup IPI
022   #define DELIVS     0x00001000   // Delivery status
023   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
024   #define DEASSERT   0x00000000
025   #define LEVEL      0x00008000   // Level triggered
026   #define BCAST      0x00080000   // Send to all APICs, including
    self.
027   #define BUSY       0x00001000
028   #define FIXED      0x00000000
029 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
030 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
031   #define X1         0x0000000B   // divide counts by 1
032   #define PERIODIC   0x00020000   // Periodic
033 #define PCINT   (0x0340/4)   // Performance Counter LVT
034 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
035 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
036 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
037   #define MASKED     0x00010000   // Interrupt masked
038 #define TICR    (0x0380/4)   // Timer Initial Count
039 #define TCCR    (0x0390/4)   // Timer Current Count
040 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
041
042 volatile uint *lapic;  // Initialized in mp.c
043
044 static void
045 lapicw(int index, int value)
046 {
047   lapic[index] = value;
048   lapic[ID];  // wait for write to finish, by reading
049 }
050 //PAGEBREAK!
051
052 void
053 lapicinit(int c)
054 {
055   if(!lapic)
056     return;
057
058   // Enable local APIC; set spurious interrupt vector.
059   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
060
```

```
061   // The timer repeatedly counts down at bus frequency
062   // from lapic[TICR] and then issues an interrupt.
063   // If xv6 cared more about precise timekeeping,
064   // TICR would be calibrated using an external time source.
065   lapicw(TDCR, X1);
066   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
067   lapicw(TICR, 10000000);
068
069   // Disable logical interrupt lines.
070   lapicw(LINT0, MASKED);
071   lapicw(LINT1, MASKED);
072
073   // Disable performance counter overflow interrupts
074   // on machines that provide that interrupt entry.
075   if(((lapic[VER]>>16) & 0xFF) >= 4)
076     lapicw(PCINT, MASKED);
077
078   // Map error interrupt to IRQ_ERROR.
079   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
080
081   // Clear error status register (requires back-to-back writes).
082   lapicw(ESR, 0);
083   lapicw(ESR, 0);
084
085   // Ack any outstanding interrupts.
086   lapicw(EOI, 0);
087
088   // Send an Init Level De-Assert to synchronise arbitration ID's.
089   lapicw(ICRHI, 0);
090   lapicw(ICRLO, BCAST | INIT | LEVEL);
091   while(lapic[ICRLO] & DELIVS)
092     ;
093
094   // Enable interrupts on the APIC (but not on the processor).
095   lapicw(TPR, 0);
096 }
097
098 int
099 cpunum(void)
100 {
101   // Cannot call cpu when interrupts are enabled:
102   // result not guaranteed to last long enough to be used!
103   // Would prefer to panic but even printing is chancy here:
104   // almost everything, including cprintf and panic, calls cpu,
105   // often indirectly through acquire and release.
106   if(readeflags()&FL_IF){
107     static int n;
108     if(n++ == 0)
109       cprintf("cpu called from %x with interrupts enabled\n",
110         __builtin_return_address(0));
111   }
112
113   if(lapic)
114     return lapic[ID]>>24;
115   return 0;
116 }
117
118 // Acknowledge interrupt.
119 void
120 lapiceoi(void)
121 {
122   if(lapic)
123     lapicw(EOI, 0);
124 }
```

```
125
126 // Spin for a given number of microseconds.
127 // On real hardware would want to tune this dynamically.
128 void
129 microdelay(int us)
130 {
131 }
132
133 #define IO_RTC  0x70
134
135 // Start additional processor running entry code at addr.
136 // See Appendix B of MultiProcessor Specification.
137 void
138 lapicstartap(uchar apicid, uint addr)
139 {
140   int i;
141   ushort *wrv;
142
143   // "The BSP must initialize CMOS shutdown code to 0AH
144   // and the warm reset vector (DWORD based at 40:67) to point at
145   // the AP startup code prior to the [universal startup
    algorithm]."
146   outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
147   outb(IO_RTC+1, 0x0A);
148   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
149   wrv[0] = 0;
150   wrv[1] = addr >> 4;
151
152   // "Universal startup algorithm."
153   // Send INIT (level-triggered) interrupt to reset other CPU.
154   lapicw(ICRHI, apicid<<24);
155   lapicw(ICRLO, INIT | LEVEL | ASSERT);
156   microdelay(200);
157   lapicw(ICRLO, INIT | LEVEL);
158   microdelay(100);    // should be 10ms, but too slow in Bochs!
159
160   // Send startup IPI (twice!) to enter code.
161   // Regular hardware is supposed to only accept a STARTUP
162   // when it is in the halted state due to an INIT.  So the second
163   // should be ignored, but it is part of the official Intel
    algorithm.
164   // Bochs complains about the second one.  Too bad for Bochs.
165   for(i = 0; i < 2; i++){
166     lapicw(ICRHI, apicid<<24);
167     lapicw(ICRLO, STARTUP | (addr>>12));
168     microdelay(200);
169   }
170 }
```

ユニプロセッサ上でもちゃんと動くようにするために、xv6は、プログラム可能な割り込みコントローラ（PIC）をプログラムする。

（picirq.cのpicinit関数）

どのPICも最大で8個の割り込み（例えば複数のデバイス）を制御出来、そしてそれらをプロセッサの割り込みピンへ多重送信する。

8個以上のデバイスに対応するため、複数のPICはカスケード接続され、典型的なボードなら最低でも2つのPICを持つ。

inb命令とoutb命令を使うことで、xv6は、マスタにIRQ 0からIRQ 7を生成させるため、スレーブにIRQ 8からIRQ 15（原文では16になってる）を生成させるために、それらをプログラムする。

最初に、xv6はPICに全ての割り込みをマスクさせるためにプログラムする。

timer.cのコードで、タイマーを1にセットしPIC上のタイマー割り込みを有効にしている。
（timer.cのtimerinit関数）
今は、PICのプログラミングに関して詳細をいくつか省いている。
それらの詳細（PIC、IOAPIC、LAPIC）は、このテキストでは重要ではないが、興味を持った読者はそれらのソースファイルで参照されているそれぞれのデバイスのマニュアルを調べる事ができる。

timer.c

```
01 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
02 // Only used on uniprocessors;
03 // SMP machines use the local APIC timer.
04
05 #include "types.h"
06 #include "defs.h"
07 #include "traps.h"
08 #include "x86.h"
09
10 #define IO_TIMER1       0x040           // 8253 Timer #1
11
12 // Frequency of all three count-down timers;
13 // (TIMER_FREQ/freq) is the appropriate count
14 // to generate a frequency of freq Hz.
15
16 #define TIMER_FREQ      1193182
17 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
18
19 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
20 #define TIMER_SEL0      0x00    // select counter 0
21 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
22 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
23
24 void
25 timerinit(void)
26 {
27   // Interrupt 100 times/sec.
28   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
29   outb(IO_TIMER1, TIMER_DIV(100) % 256);
30   outb(IO_TIMER1, TIMER_DIV(100) / 256);
31   picenable(IRQ_TIMER);
32 }
```

マルチプロセッサの場合、xv6は、それぞれのプロセッサ上でIOAPICとLAPICをプログラムしなければならない。
IO APICはテーブルを持ち、プロセッサはinb命令やoutb命令を使う代わりに、メモリマップドI/Oを通してそのテーブルのエントリをプログラムすることが出来る。
初期化中、xv6は、割り込み0をIRQ 0に対応付けるようプログラムする。（1以降も同様に）
しかし、それら全てを無効化する。
特定のデバイスは、特有の割り込みを有効にし、その割り込みを担当すべきであるとプロセッサに伝える。
例えば、xv6はキーボード割り込みをプロセッサ0に送る。
（console.cのconsoleinit関数）
後で見るが、xv6はディスク割り込みをそのシステムで一番番号が大きいプロセッサに割り当てる。

console.cのconsoleinit関数

```
01 void
02 consoleinit(void)
03 {
04   initlock(&cons.lock, "console");
05   initlock(&input.lock, "input");
06
07   devsw[CONSOLE].write = consolewrite;
08   devsw[CONSOLE].read = consoleread;
09   cons.locking = 1;
10
11   picenable(IRQ_KBD);
12   ioapicenable(IRQ_KBD, 0);
13 }
```

タイマーチップはLAPICの中にあり、そのおかげでどのプロセッサも個別にタイマー割り込みを受け取ることが出来る。
xv6は、それをlapic.cのlapicinit関数でセットアップしている。
重要な行は、タイマーをプログラムしてる部分である。
（lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));の部分)
この行は、LAPICにIRQ_TIMERで割り込みを生成させる事を伝えている。
（traps.hに#define IRQ_TIMER 0とありIRQ 0にあたる）
lapicinitの最後の行（lapicw(TPR, 0);）は、ローカルプロセッサに割り込みを運ぶためにCPUのLAPIC上の割り込みを有効にする。

プロセッサは、eflagレジスタの中のIFフラグを通して、割り込みを受け取りたいかどうかを制御することが出来る。
cli命令は、IFをクリアする事によってそのプロセッサへの割り込みを無効化し、sti命令はそのプロセッサへの割り込みを有効化する。
xv6は、起動中は、メインCPUと他のプロセッサに対する割り込みをcli命令を使って無効化する。
（スペースの都合とその内容は今は重要じゃないのでソースは載せませんがそれぞれ、bootasm.S、entryother.Sでcli命令が使われています。）
それぞれのプロセッサのスケジューラが、割り込みを有効にする。
（こちらもソースは載せませんが、proc.cのscheduler関数でsti命令が使われています。）
特定のコードの断片を割り込みされないよう制御するため、xv6はそれらのコードの断片の間、割り込みを無効化する。
（例えば、switchuvmを見よ）
（pushcli関数、popcli関数の中でそれぞれcli命令、sti命令が呼ばれます。）

vm.cのswitchuvm関数

```
01 // Switch TSS and h/w page table to correspond to process p.
02 void
03 switchuvm(struct proc *p)
04 {
05   pushcli();
06   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1,
```

```
    0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir));  // switch to new address space
  popcli();
}
```

タイマー割り込みは、ベクタ32（xv6がIRQ 0を制御するために選んだ）を通り、xv6はidtinit関数の中でセットアプする。
（idtinit関数は最終的にlidt命令を実行する）
ベクタ32とベクタ64（システムコール用）の違いは、ベクタ32はトラップゲートではなく割り込みゲートであるという事だけである。
割り込みを受けたプロセッサが、現在の割り込みを制御している間に割り込みを受けないようにするために、割り込みゲートはIFフラグをクリアする。
ここからtrap関数に至るまで、割り込みは、システムコールや例外や、トラップフレームの構築と同じコードパスをたどる。

main.cのmpmain関数（idtinit関数を呼び出す部分）

```
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting\n", cpu->id);
  idtinit();       // load idt register
  xchg(&cpu->started, 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

タイマー割り込みで呼ばれたとき、trap関数は、2つの事だけを行う。
変数ticksのインクリメント。
wakeup関数の呼び出し。
第4章で見るが、後者は違うプロセスへ戻る割り込みを引き起こす。

trap.cのtrap関数のタイマー関連の処理の部分

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpu->id == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

感想

タイマー割り込みを例にした割り込みの説明といったところですかね。
タイマー割り込みもかなり重要なので、例というのはちょっと言い過ぎですね。

ベクタというのは、前々々回Code: Assembly trap handlersの節の最初に出てきた、xv6における割り込みの区別の為の概念ですね。
実際はただの関数の配列のようなものでした。

個別の節の理解度は少しづつ上がってきてる気がします。
それぞれの関連や全体の中でどこに位置するのかがピンと来るようになればもっと理解が深まるんじゃないかと思います。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/4 日曜日 [http://peta.okechan.net/blog/archives/1397] |
