日曜研究室

技術的な観点から日常を綴ります

[xv6 #25] Chapter 2 – Traps, interrupts, and drivers – Code: Assembly trap handlers

テキストの34〜36ページ

本文

xv6は、プロセッサにトラップを引き起こさせるようなint命令に遭遇したとき、目的に叶った何らかの処理を実行するために、x86のハードウェアをセットアップしなければならない。
x86は256個の個別の割り込みを扱うことが出来る。
割り込み0-31は、除算エラーやおかしなメモリアドレスにアクセスを試みたときのような、ソフトウェア例外のために定義されている。
xv6は、32個のハードウェア割り込みを32-63の領域に割り当てていて、64をシステムコール用の割り込みに割り当てている。

tvinit関数は、main関数から呼ばれ、idtテーブルの中の256個のエントリをセットアップする。
割り込みiは、vectors[i]の中のアドレスにあるコードによって制御される。
エントリポイントは全て違う。
なぜならxv6は割り込みハンドラに対してトラップ番号を提供しないからである。
（ここ訳があやしい。because the x86 provides does not provide the trap number to the interrupt handler.）
256個の個別のハンドラを使うということは、256個の場合を見分ける唯一の方法である。
（実際に割り込みハンドラが実行されるときにはそのハンドラからはトラップ番号を得る方法がないということかな。）

trap.cのtvinit関数付近

```
01 // Interrupt descriptor table (shared by all CPUs).
02 struct gatedesc idt[256];
03 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
04 struct spinlock tickslock;
05 uint ticks;
06
07 void
```

```
08 tvinit(void)
09 {
10   int i;
11
12   for(i = 0; i < 256; i++)
13     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
14   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL],
   DPL_USER);
15
16   initlock(&tickslock, "time");
17 }
```

main.cのmain関数

```
01 // Bootstrap processor starts running C code here.
02 // Allocate a real stack and switch to it, first
03 // doing some setup required for memory allocator to work.
04 int
05 main(void)
06 {
07   kvmalloc();      // kernel page table
08   mpinit();        // collect info about this machine
09   lapicinit(mpbcpu());
10   seginit();       // set up segments
11   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
12   picinit();       // interrupt controller
13   ioapicinit();    // another interrupt controller
14   consoleinit();   // I/O devices & their interrupts
15   uartinit();      // serial port
16   pinit();         // process table
17   tvinit();        // trap vectors
18   binit();         // buffer cache
19   fileinit();      // file table
20   iinit();         // inode cache
21   ideinit();       // disk
22   if(!ismp)
23     timerinit();   // uniprocessor timer
24   startothers();    // start other processors (must come before
   kinit)
25   kinit();         // initialize memory allocator
26   userinit();      // first user process  (must come after kinit)
27   // Finish setting up this processor in mpmain.
28   mpmain();
29 }
```

tvinitは、ユーザのシステムコールをトラップするためにだけにT_SYSCALLを制御する。
それは、2番目の引数に1という値を渡す事によって”trap”タイプのゲートを指定する。
トラップゲートはFLフラグをクリアせず、システムコールハンドラ中に他の割り込みを許可する。

カーネルもまた、ユーザプログラムが直接的なint命令によってトラップを生成できるよう、システムコールゲートの権限をDPL_USERにセットする。
xv6は、プロセスにint命令を使って他の割り込み（例えば、デバイスの割り込み）を引き起こすことを許可しない。
もしプロセスがそれを試みたら、一般保護違反に遭遇するだろう。
（そしたらvectors[13]に処理が移る）

ユーザモードからカーネルモードに保護レベルを変更したとき、ユーザプロセスのスタックを使わないようにすべきである。
なぜなら、その内容がおかしくなってしまうかもしれないからである。
ユーザプロセスは、そのプロセスのユーザメモリじゃない部分を指すようなアドレスを%espにセットするようなバグや悪意を持ってるかもしれない。
xv6は、ハードウェアがスタックセグメントセレクタと%espのための新しい値を読み込む事を通して、タスクセグメントディスクリプタをセットアップする事によって、トラップ中にスタック切り替えを実行させるよう、x86ハードウェアをプログラムする。
switchuvm関数は、ユーザプロセスのカーネルスタックのトップのアドレスを、タスクセグメントディスクリプタに保存する。

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);
07   cpu->gdt[SEG_TSS].s = 0;
08   cpu->ts.ss0 = SEG_KDATA << 3;
09   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
10   ltr(SEG_TSS << 3);
11   if(p->pgdir == 0)
12     panic("switchuvm: no pgdir");
13   lcr3(v2p(p->pgdir));  // switch to new address space
14   popcli();
15 }
```

トラップが発生したとき、プロセッサのハードウェアは以下の事を行う。
もしプロセッサがユーザモードで実行中だったら、タスクセグメントディスクリプタから%espと%ssをロードし、ユーザの%ssと%espを新しいスタックにプッシュする。
もしプロセッサがカーネルモードで実行中だったら、上記のような事は何も行わない。
そしたらプロセッサは%eflagsと%csと%eipレジスタをプッシュする。
いくつかのトラップでは、プロセッサはエラー語（error word）もプッシュする。
そしたら、プロセッサは適切なIDTエントリから%eipと%csを読み込む。

xv6は、IDTエントリポイントへのエントリポイントを生成するためにPerlスクリプトを使う。
それぞれのエントリは、プロセッサがやらなかったらエラーコードをプッシュし、割り込み番号をプッシュし、そしてalltrapsにジャンプする。

vectors.pl

```
01 #!/usr/bin/perl -w
02
03 # Generate vectors.S, the trap/interrupt entry points.
04 # There has to be one entry point per interrupt number
```

```
# since otherwise there's no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector$i\n";
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
for(my $i = 0; $i < 256; $i++){
    print "  .long vector$i\n";
}

# sample output:
#   # handlers
#   .globl alltraps
#   .globl vector0
#   vector0:
#     pushl $0
#     pushl $0
#     jmp alltraps
#   ...
#
#   # vector table
#   .data
#   .globl vectors
#   vectors:
#     .long vector0
#     .long vector1
#     .long vector2
#   ...
```

alltrapsは、プロセッサのレジスタの保存を続ける。
%ds, %es, %fs, %gs、それと汎用的なレジスタをプッシュする。（alltrapsのpushl %dsからpushalの部分）
この努力の結果、カーネルスタックに、トラップの瞬間のプロセッサのレジスタを含むtrapframe構造体が含まれる。（図2–2を参照）
プロセッサは、%ss, %esp, %eflags, %cs, %eipをプッシュする。
プロセッサもしくはトラップベクタはエラー番号をプッシュし、alltrapsは残りをプッシュする。
トラップフレームは、カーネルから現在のプロセスに戻ったときにユーザモードのプロセッサのレジスタを復元するのに十分な全ての情報を含む。
その結果、プロセッサはトラップが開始した時点とちょうど同じ状態で続行することができる。
第1章を思い出すと、userinitはこのゴールを達成するために、手動でトラップフレームを構築している。
（図1-3参照）

図2-2 カーネルスタック上のトラップフレーム

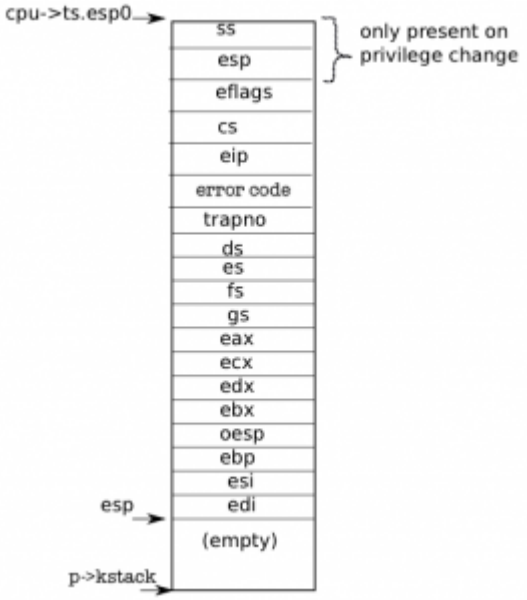
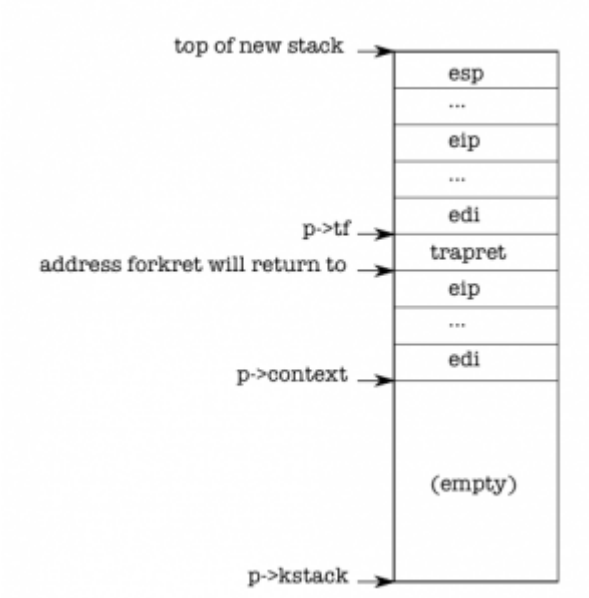


図1-3 新しいカーネルスタックのセットアップ（再掲）



trapasm.S

```
01 #include "mmu.h"
02
03   # vectors.S sends all traps here.
04 .globl alltraps
05 alltraps:
06   # Build trap frame.
07   pushl %ds
08   pushl %es
09   pushl %fs
10   pushl %gs
11   pushal
12
```

```
  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

x86.hのtrapframe構造体

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to
kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

最初のシステムコールの場合、保存された%eipは、int命令の直後の命令のアドレスである。
%csはユーザコードのセグメントセレクタである。
%eflagsはint命令を実行した瞬間のeflagsレジスタの内容である。
保存された汎用レジスタの一部のように、alltrapsは、カーネルが後で調べるためのシステムコール番号を含む%eaxもまた保存する。

今、ユーザモードのプロセッサのレジスタは保存され、alltrapsはプロセッサにカーネルのCコードを実行させるためのセットアップを完了することが出来る。
プロセッサは、ハンドラに入る前にセレクタ%cs, %ssをセットする。
alltrapsは%dsと%esをセットする。（alltrapsのmovw $(SEG_KDATA<<3), %axからmovw %ax, %esの部分）
それから、CPUごとのデータセグメントであるSEG_KCPUを指すために%fsと%gsをセットする。（alltrapsのmovw $(SEG_KCPU<<3), %axからmovw %ax, %gsの部分）

一度セグメントが適切にセットされたら、alltrapsは、Cで書かれたトラップハンドラであるtrap関数（trap.c）を呼ぶことが出来る。
alltrapsは、構築されたばかりのトラップフレームを指す%espを、trapへの引数としてスタックにプッシュする。（alltrapsのpushl %espの部分）
そしてtrapを呼ぶ。（alltrapsのcall trapの部分）
trap関数から返ってきた後、alltrapsは、スタックポインタに加算することによってスタックの引数を取り除き、そして、trapretとラベルづけされたコードの実行を開始する。
我々は、第1章で、最初のユーザプロセスがユーザ空間に抜けるためにtrapretを実行したとき、このコードを追った。
同じ処理がここで起きた。
トラップフレームを通じてポップする事は、ユーザモードのレジスタを復元し、そしてiretはユーザ空間に戻す。

trap.cのtrap関数

```
01 void
02 trap(struct trapframe *tf)
03 {
04   if(tf->trapno == T_SYSCALL){
05     if(proc->killed)
06       exit();
07     proc->tf = tf;
08     syscall();
09     if(proc->killed)
10       exit();
11     return;
12   }
13
14   switch(tf->trapno){
15   case T_IRQ0 + IRQ_TIMER:
16     if(cpu->id == 0){
17       acquire(&tickslock);
18       ticks++;
19       wakeup(&ticks);
```

```
20        release(&tickslock);
21      }
22      lapiceoi();
23      break;
24    case T_IRQ0 + IRQ_IDE:
25      ideintr();
26      lapiceoi();
27      break;
28    case T_IRQ0 + IRQ_IDE+1:
29      // Bochs generates spurious IDE1 interrupts.
30      break;
31    case T_IRQ0 + IRQ_KBD:
32      kbdintr();
33      lapiceoi();
34      break;
35    case T_IRQ0 + IRQ_COM1:
36      uartintr();
37      lapiceoi();
38      break;
39    case T_IRQ0 + 7:
40    case T_IRQ0 + IRQ_SPURIOUS:
41      cprintf("cpu%d: spurious interrupt at %x:%x\n",
42              cpu->id, tf->cs, tf->eip);
43      lapiceoi();
44      break;
45
46    //PAGEBREAK: 13
47    default:
48      if(proc == 0 || (tf->cs&3) == 0){
49        // In kernel, it must be our mistake.
50        cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
51                tf->trapno, cpu->id, tf->eip, rcr2());
52        panic("trap");
53      }
54      // In user space, assume process misbehaved.
55      cprintf("pid %d %s: trap %d err %d on cpu %d "
56              "eip 0x%x addr 0x%x-kill proc\n",
57              proc->pid, proc->name, tf->trapno, tf->err, cpu->id,
tf->eip,
58              rcr2());
59      proc->killed = 1;
60    }
61
62    // Force process exit if it has been killed and is in user space.
63    // (If it is still executing in the kernel, let it keep running
64    // until it gets to the regular system call return.)
65    if(proc && proc->killed && (tf->cs&3) == DPL_USER)
66      exit();
67
68    // Force process to give up CPU on clock tick.
69    // If interrupts were on while locks held, would need to check
nlock.
70    if(proc && proc->state == RUNNING && tf->trapno ==
T_IRQ0+IRQ_TIMER)
71      yield();
72
73    // Check if the process has been killed since we yielded
74    if(proc && proc->killed && (tf->cs&3) == DPL_USER)
75      exit();
76  }
```

ここまでの解説は、ユーザモードから発生したトラップについて話してきた。
しかしトラップはカーネルを実行してるときでも発生する。

そのような場合、ハードウェアはスタックを切り替えたり、スタックポインタやスタックセグメントセレクタを保存したりはしない。
しかしながら、ユーザモードからのトラップと同じ手順が発生し、そして同じxv6のトラップを制御するコードを実行する。
iretの後、カーネルモードの%csを復元し、プロセッサはカーネルモードでの実行を続ける。

感想

ユーザモードでシステムコールしたときに発生するトラップで、どうやってカーネルモードに移行してどうやって戻るかの話ですね。

本文にも書いてあるけど、vectors.Sはvectors.plから生成するようになってるので、一度makeとかしないと存在しません。

タスクセグメントディスクリプタ周りの話がよくわかりません。
プロテクトモード - Wikipediaを読めば少しは分かるかな。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/1 木曜日 [http://peta.okechan.net/blog/archives/1383] |