日曜研究室

技術的な観点から日常を綴ります

[xv6 #26] Chapter 2 – Traps, interrupts, and drivers – Code: C trap handler

テキストの36ページ

本文

我々は前節で、どのハンドラもトラップフレームをセットアップし、そしてCの関数であるtrapを呼ぶところを見てきた。
trap関数は、なぜ呼ばれ何をすべきか決定するためにハードウェアトラップ番号tf->trapnoを見る。
もしそのトラップがT_SYSCALLなら、trap関数はシステムコールのハンドラであるsyscall関数を呼ぶ。
我々は、第4章で2つのproc->killed（原文ではcp->killedになってるけど多分間違い）について再検討する予定である。

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  }
```

システムコールの為のチェックの後、trap関数はハードウェア割り込みを見る。（これについては後で説明する。）
予期されたハードウェアデバイスに加えて、trap関数は擬似割り込み（不必要なハードウェア割り込み）によっても引き起こされる。

もしトラップがシステムコールやハードウェアデバイスの割り込みでなければ、trap関数はそれをトラップの前に実行されてたコードの一部によるおかしな処理（例えばゼロ除算）によって引き起こされたものだと仮定する。
もし、そのトラップを引き起こしたコードがユーザプログラムなら、xv6は詳細を印字し、そしてそのユーザプロセスをクリーンアップする事を忘れないようにするためにproc->killed（原文ではcp->killed）をセットする。
我々は、第4章でxv6がどうやってこのクリーンアップを実行するかを見る予定である。

もし、それがカーネルの実行中なら、カーネルのバグということになる。
trap関数は、その驚きについて詳細を印字し、そしてpanic関数を呼ぶ。

感想

trap関数ではtf->trapnoで処理を割り振るという話です。

cp->killedとproc->killedについては、gitのログを調べたところ2009/8/31にrename c/cp to cpu/procというコメントを含んだコミットがありそこでcpからprocに変更されていたので、本文が間違ってる（古いまま残ってる）のだろうと思います。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/2 金曜日 [http://peta.okechan.net/blog/archives/1390] |
