

# **16 级计科 7 班: 操作系统原理实验 #5**

Due on Friday, April 13, 2018

凌应标 周一 9-10 节

**颜彬**

**16337269**

# Content

	Page
<b>1 实验目的</b>	<b>3</b>
<b>2 实验方案</b>	<b>3</b>
2.0.1 中断调用原理 . . . . .	3
2.0.2 软硬中断简介 . . . . .	3
2.0.3 用户自定义中断 . . . . .	3
2.1 实验环境 . . . . .	4
2.1.1 系统与虚拟机 . . . . .	4
2.1.2 相关工具、指令 . . . . .	4
<b>3 实验过程</b>	<b>4</b>
3.0.1 GCC 语言拓展的运用 . . . . .	4
3.1 修改时钟中断 . . . . .	6
3.2 修改键盘中断 . . . . .	6
3.3 安装用户自定义中断 . . . . .	7
<b>4 实验结果</b>	<b>7</b>
4.1 终端界面 . . . . .	7
4.2 用户程序 . . . . .	7
4.3 特色中断 . . . . .	7
<b>5 实验总结</b>	<b>8</b>
5.1 亮点介绍 . . . . .	8
5.1.1 极其详尽的文档和代码开源 . . . . .	8
5.1.2 ASM 拓展语法 . . . . .	8
5.1.3 格式化 ls 输出 . . . . .	8
5.1.4 光标滚屏自主控制 . . . . .	8
5.2 更新细节 . . . . .	8
5.2.1 printf 进一步完善 . . . . .	8
5.2.2 目录树结构变更 . . . . .	8
5.3 心得体会 . . . . .	8
5.4 BUG 汇总 . . . . .	8
<b>A 参考文献</b>	<b>8</b>
<b>B 其他代码</b>	<b>9</b>

## 1 实验目的

掌握 PC 微机的实模式硬件中断系统原理和中断服务程序设计方式, 实现对时钟、键盘/鼠标等硬件中断的简单服务处理程序编程和调试, 让你的原型操作系统在运行以前已有的用户程序时, 能对异步事件正确地捕捉和响应。

掌握操作系统的系统调用原理, 实现原型操作系统中的系统调用框架, 提供若干简单功能的系统调用。

学习掌握 C 语言库的实际方法, 为自己的原型操作系统配套一个 C 程序开发环境, 实现用自建的 C 语言开发简单的输入/输出的用户程序, 展示封装的系统调用。

## 2 实验方案

### 2.0.1 中断调用原理

在实模式下, 内存 0x0000 起始处维护着一张中断服务程序入口的表。每个表项站 4 字节, 共同代表服务程序的入口地址。其中的低 16 位代表代码段地址, 高 16 位代表段内偏移地址。

当调用中断时, 中断号代表着服务程序地址在地址表中的索引。所以当执行 `int id` 时, 会将 `0x0000 : id×4` 起始的 32 bits 作为入口地址。一般寄存器 `ah` 中的值会作为中断的“功能号”。其他寄存器的值依据文档, 相应地作为参数或作为返回值。

当实模式下触发中断时, 会首先将 `FLAGS` 寄存器压入栈中 (32 位模式下是 `EFLAGS`), 随后压入 `cs:ip`, 并根据地址表中的信息, 跳转到相应地址。在中断返回时, 要首先向硬件端口输出相应参数, 告知中断处理硬件中断完成。随后, `iret` 指令将把 `FLAGS` 和 `cs:ip` 出栈, 并跳转回相应的用户程序地址。

这些中断调用的原理就给了自定义中断的可能。用户只需修改中断地址表, 并按照中断的行为书写自己的代码, 就能实现自己的中断。

### 2.0.2 软硬中断简介

软件和硬件都能引发中断。其中软件中断又称为系统调用。其与调用一个函数 (在程序员可见角度) 类似, 有输入的参数和相应的返回值。程序员在系统调用前应手动保存需要保存的寄存器, 并能预计到中断会修改相应寄存器的值。系统调用都是不可屏蔽的。

硬件中断有可屏蔽和不可屏蔽两种。其中时钟中断 (`watchdog`) 是不可屏蔽中断。由于硬件中断可以发生在任何一个时间点, 所以硬件中断应能保证绝对不修改任何寄存器的值。即硬件中断对程序员是透明的。

### 2.0.3 用户自定义中断

用户自定义中断可以采用如下的方式。首先查阅中断地址表, 找到未被占用的中断号 (例如本项目使用的 `0x2B` 号)。将地址表的相应表项修改用户自定义的地址。则当中断发生时, 程序控制流可以到达用户自定义地址。

标准的中断实现中, 用户自定义中断应该提取 `ah` 作为功能号, 建立中断自己的功能地址表, 根据 `ah` 的值在功能地址表中提取相应功能的中断入口地址, 并跳转到该地址。

中断返回时, 要保证将栈退到“刚进入中断”时的状态, 向硬件端口传送一些数据, 并用 `iret` 指令返回。

## 2.1 实验环境

### 2.1.1 系统与虚拟机

- 操作系统  
本实验在 Linux 下完成。采用 Ubuntu 16.04
- 虚拟机  
bochs. 它是一款开源且跨平台的 IA-32 模拟器。

### 2.1.2 相关工具、指令

- 汇编器  
NASM. NASM 是一个轻量级的、模块化的 80x86 和 x86-64 汇编器。它的语法与 Intel 原语法十分相似, 但更加简洁和易读。它对宏有十分强大的支持。
- 编译器  
GCC. GNU/GCC 是开源的 C 语言编译器。其产生的伪 16 位代码可以与 NASM 结合, 混编生成伪 16 位程序。
- 镜像文件产生工具  
bxiimage. 该命令允许生成指定大小的软件镜像。
- 二进制写入命令  
dd. dd 允许指定源文件和目标文件, 将源文件的二进制比特写入目标文件中的指定位置。
- 二进制文件查看命令  
xxd. xxd 允许将二进制文件中的内容按地址顺序依次输出, 可读性强
- 反汇编器  
objdump. objdump 可以查看目标文件和二进制文件的反汇编代码, 还能指令 intel 或 at&t 格式显示。
- 代码生成脚本  
makefile. makefile 脚本具有强大的功能, 其可以识别文件依赖关系, 自动构筑文件, 自动执行 shell 脚本等。

## 3 实验过程

### 3.0.1 GCC 语言拓展的运用

为了简化代码的书写, 本项目进一步地使用了 GCC 的语言拓展, `__asm__` 语法。代码1举出了两个比较有说明性的道理, 列举其用法。内嵌汇编的初衷是, 若 C 语言不得不调用汇编, 而汇编语句又很短 (10 行以内), 为了汇编而新建一个文件并编译链接显得很麻烦。内嵌汇编可以做到“细粒度”的 C 与汇编交互功能。

`volatile` 语句的作用是“禁止 GCC 优化 (删除) 掉汇编语句”。由于 GCC 在编译时无法解析汇编的内容, 汇编的内容是在 C 语言代码完全编译后, 再由内置汇编器解释的。故 GCC 无法在编译期确定内嵌的汇编代码是否有副作用。若 GCC (错误地) 认为这些汇编代码没有副作用, 会把代码完全优化 (删

除) 掉。volatile 可以禁止这种优化。

整个拓展语法为 `__asm__ volatile ( assembly : output : input : clobber );`

其中 assembly 为字符串表示的汇编代码。以回车结尾以分隔汇编代码。output 显式地指明了汇编代码会“输出”到 C 变量, 即汇编代码会修改某个 C 变量的值。input 显式地指明了汇编代码会从 C 语言“输入”值, 即把 C 的某个变量赋给寄存器。clobber 显式地指明了汇编语句会修改到哪些内容 (如标志寄存器, 内存等)。如果不指明, C 语言会 (为了速度) 而不刷新缓存, 导致汇编语句的修改没有更新到 C 语言环境中。

代码1中, “c”, “D” 分别代表寄存器 CX 和 DX, “CC” 表示标志寄存器, 其他标志可从字面意思看懂。这些标志与架构密切相关, 可以从 GCC 官网的 ASM 语言拓展查到详细的使用手册。

代码1的两个函数中, 第一个函数通过中断的方式在屏幕的某个特定的位置显示字符。C 语言对它做了便利的封装。第二个函数用于得到光标位置, 并自动导出到 cursor\_pos 中。

代码 1: GCC 内嵌汇编拓展语法详解

```
static inline int16_t _draw_char(char ch, int offset, uint8_t style) {
2   uint16_t written_data = ch | (style << 8);
   __asm__ volatile (
4       "movb $1, %%ah\n"
       "int $0x2B\n"
6       : /* no output */
       : "c"(written_data), "D"(offset)
8       : "cc", "ax", "memory"
   );
10  return 1;
}
12 static inline uint16_t get_cursor(){
   uint16_t cursor_pos;
14   __asm__ volatile(
       "pushw %%ax\n"
16       "pushw %%bx\n"
       "movb $0x03, %%ah\n"
18       "movb $0, %%bh\n"
       "int $0x10\n"
20       "popw %%bx\n"
       "popw %%ax\n"
22       : "=d"(cursor_pos)
       : /* no input */
24       : "cc", "cx"
   );
26  return cursor_pos;
}
```

### 3.1 修改时钟中断

本步骤的重点在于，保证时钟中断结束后不修改任何寄存器的值，且进入时钟中断的时立即将段寄存器初始化到正确的状态。

由于时钟中断可以发生在代码执行的任何一个时刻，用户程序在执行到任何一个状态时，都有可能被突然间中断。如果时钟中断没有保存（哪怕一个）寄存器的值，当中断返回时，用户程序会直接崩溃。更可怕的是，有可能用户程序不立即崩溃，带着错误的状态继续执行下去，错误像滚雪球般地越积越多。当程序错误地退出时，很难定位到 BUG 的具体发生位置。

更需要注意的是，在时钟中断进入后，要尽快地将 ds 的值设置为 cs 的值。虽然本项目使用平坦模型 (flatten model)，段寄存器的值都是 0，但是由于时钟中断可以发生在任何时候（哪怕当前正在运行另外一个中断）。而另外的中断（例如 int 16H）的代码中很可能临时地修改段寄存器。在时钟中断突然来临后，ds 的值将是错误的值。这同样会导致程序的异常出现。

代码2展示了自定义时钟中断的实现。假设代码已经成功地将中断向量表地址修改为 timeOut 汇编函数的首地址。timeOut 后，程序会先保存一些寄存器的值，将 ds 同步为 cs，随后使用 call 指令调用 C 的真正处理函数。当从 C 函数中返回后，恢复相关寄存器，随后跳转到原时钟中断 (0F000H:0FEA5H) 的地址。由于不清楚原时钟中断做了什么，所以最好在程序最后跳转回原中断，让它完成它想完成的事情。

代码 2: 自定义时钟中断的部分代码

```
timeOut:
2
4     pusha
4     push gs
4     push ds
6     push es

8     mov ax, cs
8     mov ds, ax
10    mov es, ax

12    push cs
12    call timeout ; timeout is a C function
14

16    pop es
16    pop ds
16    pop gs
18    popa

20    jmp 0F000H:0FEA5H ; 为了展示目的，这里采用硬编码的方式
```

### 3.2 修改键盘中断

键盘中断的实现和时钟中断十分类似。首先修改中断向量表，保证中断发生时控制权限能到达用户自定义程序。只要保证能恢复到中断前的状态，程序一般就能正确运行。

本个实现的小技巧是，采用 `pushf` 和 `call far` 的方式先调用旧键盘中断处理程序。由于旧中断处理程序的最后一条指令必为 `iret`，刚好可以把 `pushf` 和 `call far` 中压栈的内容正确地恢复到原有的寄存器中。随后自定义中断服务程序再通过一个 `iret` 把栈完全复原。

### 3.3 安装用户自定义中断

为了让用户自定义中断的行为和系统自带的中断保持类似，自定义中断也采用 `ah` 作为功能号标志。具体实现如代码3所示。

`global_custom_int_install` 函数安装用户自定义中断，把 `2BH` 号中断链接到 `custom_int_handler` 函数中。该函数是个中断处理函数，它负责解析功能号 (`ah`) 的值，在一个向量表中查询服务程序的地址，并将控制权移交给服务程序。值得注意的是，所有的服务程序都通过 `retf` 的方式先返回到 `custom_int_handler`，由其统一地使用 `iret` 结束中断。也就是 `custom_int_handler` 是一个统一的入口和出口，做了封装和“分发处理”。

`custom_int_table` 地址表。功能号 `ah` 代表着地址在表中的位置。本项目定义了三个简单的中断程序。第一个叫 `testcase`，它做了若干入栈和出栈操作。它用来检测“从中断安装到中断执行”的过程中，栈是否正确地返回了。`_int_draw_char` 是重写的底层输出函数。所有的 IO 函数几乎都依赖于这一中断调用。见//TODO: `_draw_char` 的介绍。`_int_draw_my_info` 中断会在程序的特定位置显示 3-D 的学号。视觉效果较好。

## 4 实验结果

### 4.1 终端界面

图1展示了终端界面的外观。//TODO: 由于四周围的边框需要有闪动的字符，所以终端的所有提示信息都应自动地为边框腾出空间。为了完成这个效果，本项目结合了滚屏中断和光标设置中断，实现了一个特殊的字符输出中断，原本写好的 IO 程序可以不加修改，只修改他们底层最终依赖的中断，即可自动地为边框腾出空间。

图1还同时展示了改进版的 `ls` 指令。该指令会用对齐的方式输出根目录的所有文件，文件的大小以及文件所在的首扇区。对齐的输出使用了新增的 `printf` 定宽输出功能。见//TODO:

文件系统其实还实现了输出文件所有簇编号这一功能。只是由于内核程序太大，所占的簇数太多，导致输出的画面显得混乱。所以在这个版本中我把这个功能砍掉了。但我保留了一张截图，如??所示。

### 4.2 用户程序

图3实现了带按键反馈的用户程序。在进入用户程序时，安装自定义中断，键盘对每次按下都会输出“OUCH!”，而在离开用户程序后，中断复原，不再对键盘按下响应“OUCH!”。

### 4.3 特色中断

本实现实现了特色中断，可以以 3D 的形式输出本人的学号。如图4 所示。

## 5 实验总结

### 5.1 亮点介绍

#### 5.1.1 极其详尽的文档和代码开源

本次项目为所有文件添加了足够的注释, 并自动生成了说明文档。说明文档可以见网页

<https://yanb25.github.io/OperatingSystem/files.html>。

其上极其详细地为几乎所有函数增加说明, 列举了函数间的依赖关系以及调用方法等。

#### 5.1.2 ASM 拓展语法

本实验在上个实验的基础上, 更深入地使用了 GCC 的 ASM 语言拓展。为 C 和汇编的交互带来了更大的便利。见 3.0.1 节。

#### 5.1.3 格式化 ls 输出

本实验实现了格式化的 ls 输出, 在输出文件名的同时, 格式化地输出文件大小和文件首簇位置。本实验还实现了函数, 可以将文件的所有簇按序输出。见图??和图1。其他寄存器

#### 5.1.4 光标滚屏自主控制

为了完成终端中闪烁的边框, 本项目将 IO 函数的一个底层中断重写了一遍。并结合了光标控制和自动滚屏。见//TODO: 所示。

### 5.2 更新细节

#### 5.2.1 printf 进一步完善

实现了定宽输出的功能。即支持 `printf("%4d", 1)` 功能, 在输出的 1 前输出 3 个空格。//TODO: 同时, 所有的 IO 函数重构成返回 `int16_t` 类型, 代表他们输出到屏幕的字符数。这为格式化输出提供了巨大的帮助。

#### 5.2.2 目录树结构变更

将 C 程序库进一步细分, 分成了 `utilities.h`, `stdio.h`, `ctype.h`, `mystring.h` 等基本库。`utilities.h` 负责内核所需的最基本的操作, 例如扇区读写和光标控制等。其他头文件作用类似与 ANSI C 的实现。

### 5.3 心得体会

### 5.4 BUG 汇总

## 附录 A 参考文献

sasadf

1. <https://blog.csdn.net/longintchar/article/details/50602851>  
16 位和 32 位汇编指令的不同 (尤其是 push 指令)



2. <https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s1>  
GCC 内嵌汇编的书写。
3. <http://blog.51cto.com/dengqi/1349327> FAT32 文件系统讲解
4. <https://blog.csdn.net/yeruby/article/details/41978199> FAT16 文件系统讲解

## 附录 B 其他代码

代码 3: 用户自定义中断的实现

```
global custom_int_install:
2   push ds
   mov ax, 0
4   mov ds, ax

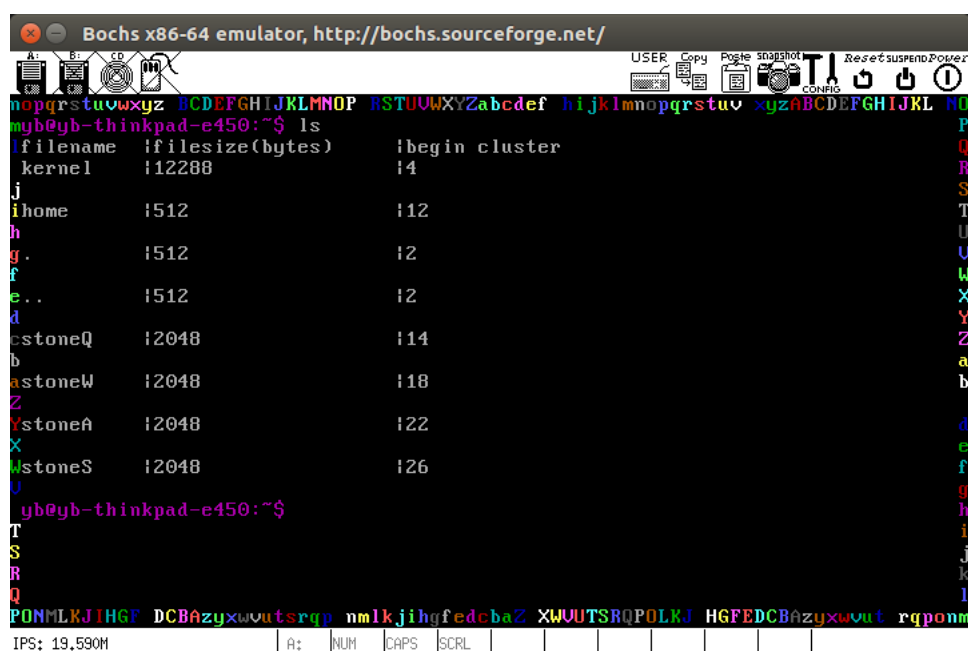
   mov word [ds:2BH * 4], custom_int_handler
   mov word [ds:2BH * 4 + 2], cs
8
   pop ds
10  retl
custom_int_handler:
12  ; ah=01H: _draw_char
   push bx
14  push ax
   push dx
16
   mov dl, 4
18  mov al, ah
   xor ah, ah
20  mul dl
   add ax, custom_int_table
22  mov bx, ax

   pop dx
24  pop ax

   call far [bx]
28
   pop bx
30  iret

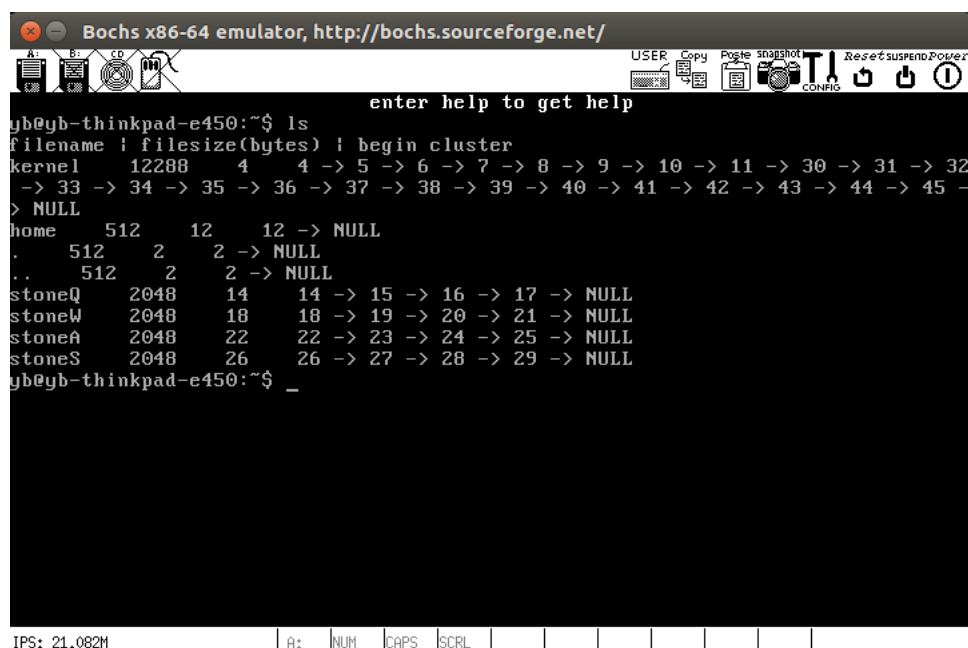
32  testcase:
   ...
34
   retf
36  _int_draw_char:
   ...
38
   retf
40
_int_draw_my_info:
42  ...

44  retf
custom_int_table:
46  dw testcase
   dw 0x0000
48  dw _int_draw_char
   dw 0x0000
50  dw _int_draw_my_info
   dw 0x0000
```



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
USER Copy Paste Snapshot Reset suspend Power
noprstuvwxyz BCDEFGHIJKLMNOP RSTUUVWXYZabcdef hijklmnopqrstuvwxyz ABCDEFGHIJKL NO
myb@yb-thinkpad-e450:~$ ls
filename      filesize(bytes)  begin cluster
kernel        12288            14
j
ihome         1512            12
h
g.            1512            12
f
e..           1512            12
d
cstoneQ       12048           14
b
astoneW       12048           18
Z
ystoneA       12048           22
X
stoneS        12048           26
yb@yb-thinkpad-e450:~$
T
S
R
Q
POMMLKJIHGFI DCBAzyxwvutsrqpn mlkjihgfedcbaZ XWUUTSRQPOLKJ HGFEDCBAzyxwvut rqpomn
IPS: 19,590M | A: | NUM | CAPS | SCRL | | | | | | | | | |
```

图 1: 带运动边框的终端界面



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
USER Copy Paste Snapshot Reset suspend Power
enter help to get help
yb@yb-thinkpad-e450:~$ ls
filename      filesize(bytes)  begin cluster
kernel        12288            4  4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 30 -> 31 -> 32
-> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 41 -> 42 -> 43 -> 44 -> 45 -> NULL
home          512            12  12 -> NULL
.             512            2  2 -> NULL
..            512            2  2 -> NULL
stoneQ        2048            14  14 -> 15 -> 16 -> 17 -> NULL
stoneW        2048            18  18 -> 19 -> 20 -> 21 -> NULL
stoneA        2048            22  22 -> 23 -> 24 -> 25 -> NULL
stoneS        2048            26  26 -> 27 -> 28 -> 29 -> NULL
yb@yb-thinkpad-e450:~$ _
IPS: 21,082M | A: | NUM | CAPS | SCRL | | | | | | | | | |
```

图 2: 带有文件所有扇区列表的 ls 指令

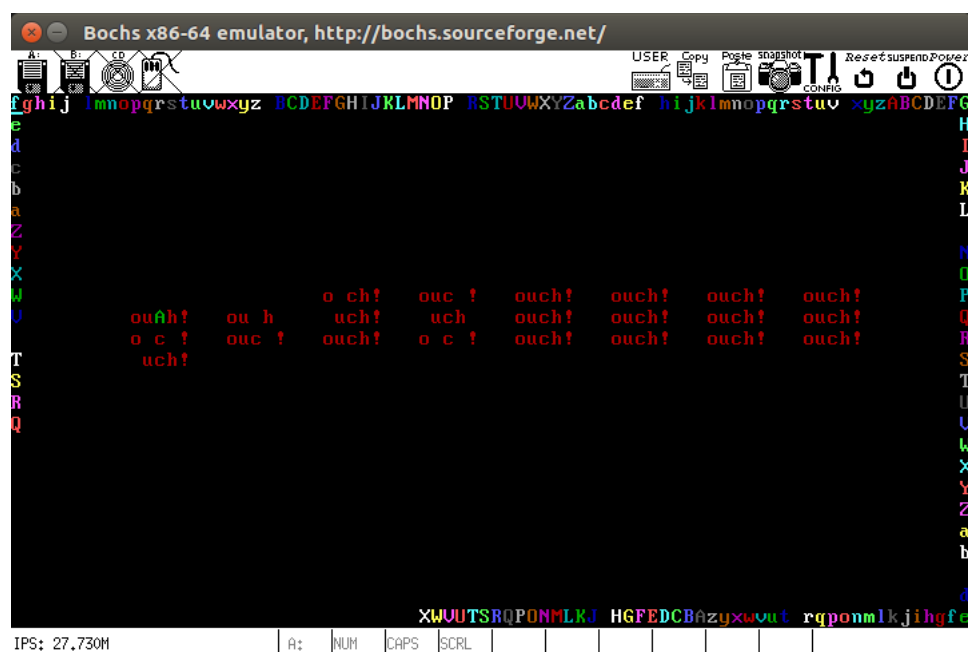


图 3: 带按键反馈的用户程序

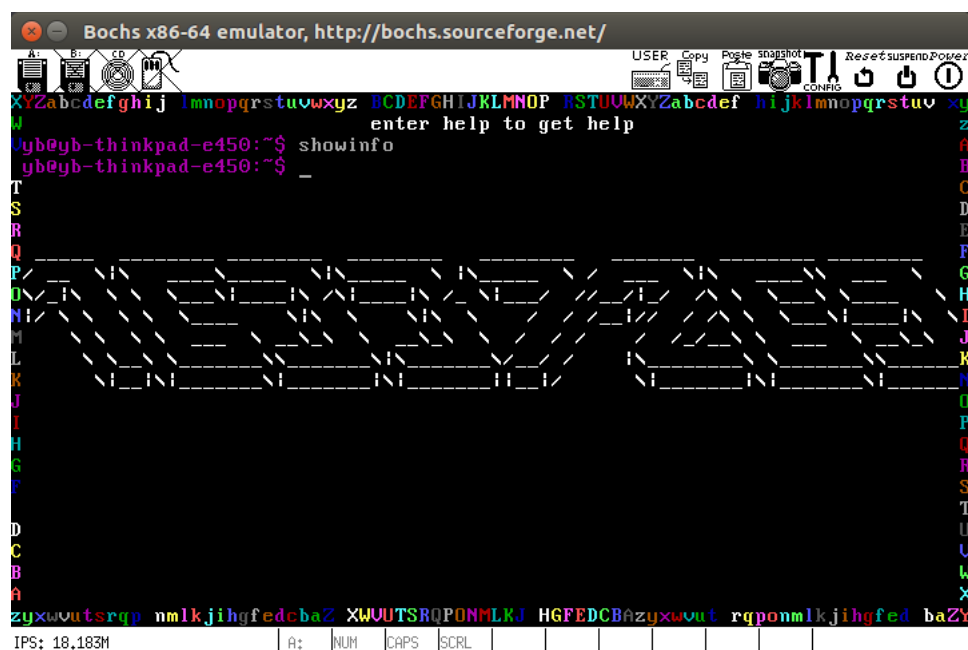


图 4: 特色中断: 个人学号的 3D 显示