2020 年秋操作系统 xv6 源码阅读报告 1
中断与异常

黎善达

18000129610@pku.edu.cn

2020 年 10 月 23 日

1 关键代码阅读与分析

xv6 源码中,涉及中断、异常和系统调用的主要文件包括 `traps.h`,`vectors.pl`,`trapasm.S`,`trap.c`,`syscall.c`，`sysproc.c` 等；同时涉及一些其它文件中的函数。

`traps.h`

主要是一些宏的定义。

x86 体系结构中一共允许有 256 类中断和异常，依次编号为 0 到 255。在 xv6 中，0 到 31 号为异常（软件中断），32 到 63 号为中断（硬件中断），64 号用作系统调用。`traps.h` 即为各种中断/异常分配中断号，其前若干行如下：

```
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
```

`vectors.pl`

这是一个脚本文件，用于生成 `vectors.S`。生成的 `vectors.S` 包含了各个中断处理程序的入口。在每个入口，对于某些特定类型的中断（`vectors.pl` 中第 14 行的特判），CPU 会将一个 0 压栈；然后对所有中断，CPU 都将中断类型号压栈；最后，所有中断统一跳转到 `alltraps`。事实上，对于某些

特定类型的中断 CPU 将一个 0 压栈，是因为这些中断发生时硬件并不自动产生错误码，因此以软件的方式将其压栈。alltraps 包含了所有中断都需要进行的一些操作，在 trapasm.S 中出现。

vectors.S 还声明了中断向量表 vectors 数组，事实上就是中断程序的入口地址。trap.c 的 tvinit 函数会用该数组对门描述符进行初始化。

trapasm.S

该文件是汇编代码，包含两个部分：alltraps 和 trapret，分别是进入中断和退出中断的一些保存现场、恢复现场的工作。

在 alltraps 中主要进行以下工作：

- **保存现场**：将%ds,%es,%fs,%gs 寄存器压栈，调用 pushal 将%eax，%ecx 等寄存器压栈。xv6 的注释将该过程称为建立**中断帧** (trap frame)。
- **设置段寄存器**：将 SEG_KDATA<<3 存入数据段寄存器%ds、附加段寄存器%es，使用内核对应的数据。
- **调用 trap 函数**：trap 函数所需的参数 tf 为指向中断帧的指针，即为当前的栈指针%esp。trap 函数是中断处理的核心函数，将在稍后说明之。

trapret 的主要工作是恢复现场，即恢复各个通用寄存器和段寄存器，再将栈指针增加 8 以将错误码和中断类型号弹栈，最后调用 iret 返回。

trap.c

这是中断处理的核心文件。

该文件声明了所有 CPU 共用的中断描述符表。

tvinit 函数初始化了 256 个门描述符。该函数在文件 main.c 的 main 函数中被调用，是系统启动时初始化工作的一部分。该函数将 vectors.S 中的 vectors 数组填入门描述符表，同时设置了门类型、权限等信息。值得注意的是，**系统调用对应的中断向量被特殊处理**：系统调用采用陷阱门（从而允许其他中断打断），且在用户权限下即可访问。系统调用是唯一的权限为 DPL_USER 的中断，用户模式下调用 int 指令，只能以系统调用的中断号作为参数，否则将因为权限不匹配产生一般保护异常 (general protection exception)。

idtinit 函数调用了 lidt 函数，其行为是将这里填好的门描述符表的指针存入硬件中的寄存器 IDTR。该函数在文件 main.c 的 mpmain 函数中被调用，是 CPU 启动时工作的一部分。

trap 函数是该文件中最主要的函数。前文中已经看到，所有中断在执行完 alltraps 后都会调用该函数。该函数根据得到的中断帧进行中断处理，其逻辑可以梳理如下：

- 检查中断号，判断是系统调用，中断（硬件中断），还是异常（软件中断），并分别处理之。

  - **系统调用**：首先检查当前进程是否被杀死，若被杀死，则退出程序。通常情况下（当前进程未被杀死），会进一步调用系统调用的处理函数 `syscall`。处理完毕后，将再次检查当前进程是否被杀死，若被杀死，则退出程序；否则正常返回。
  - **硬件中断**：通过 `switch` 语句进一步判断硬件中断的类型，并相应处理。任何类型的硬件中断处理完毕后，总是会调用 `lapiceoi` 函数，该函数在 `lapic.c` 中定义，行为是将局部高级可编程中断控制器 (Local APIC) 中的 EOI 寄存器置 0，若不如此做，该硬件中断将无法得到再次触发。
  - **软件中断**：软件中断一般由程序中的错误行为导致（如除零、溢出等）。出现软件中断时，xv6 先检查此时是在内核态还是用户态。若在内核态，表明是操作系统造成的错误，因此程序打印错误信息，并调用 `panic` 函数；若在用户态，则打印错误信息，并将当前进程的 `killed` 位置 1，从而稍后系统会清理该进程。
- 分情况处理完毕后，进行一些对特殊情况的额外处理。
  - 若当前进程的权限为 `DPL_USER` 且 `killed` 位已被置 1，则退出程序。
  - 若中断号对应时钟中断，调用 `yield` 函数令当前进程让出 CPU。
  - 因为先前调用了 `yield` 函数，所以需要再次检查当前进程是否被杀死：若当前进程的权限为 `DPL_USER` 且 `killed` 位已被置 1，则退出程序。

syscall.h

主要是一些宏的定义，定义了各个系统调用对应的系统调用号。

syscall.c

该文件中首先定义了一系列辅助函数 (helper function)，如 `fetchint`，`fetchstr`，`argint`，`argptr` 等等，用于取出用户程序在进行系统调用时准备的参数。这些函数的一大特点是进行了显示的访问权限判断，以 `argptr` 为例：

```
1  // Fetch the nth word-sized system call argument as a pointer
2  // to a block of memory of size bytes. Check that the pointer
3  // lies within the process address space.
4  int
5  argptr(int n, char **pp, int size)
6  {
7    int i;
8    struct proc *curproc = myproc();
```

```
9
10      if(argint(n, &i) < 0)
11        return -1;
12      if(size<0 || (uint)i>=curproc->sz || (uint)i+size>curproc->sz)
13        return -1;
14      *pp = (char*)i;
15      return 0;
16    }
```

该程序首先调用 argint 取出一个整形的参数，若 argint 函数调用出错，即访问了用户权限下不可访问的地址，则 argptr 返回 −1 表示出错。其次再对取出的整形参数作为检查，该整形参数是一个指针，如果它指向的空间包含用户权限不可访问的地址，则 argptr 返回 −1 表示出错。以上两次判断结果均为合法，才能最终完成 argptr。这里需要进行多次权限检查是因为：用户程序进行系统调用时，只能访问其权限对应的数据，否则将会造成异常（软件中断）并杀死当前进程；但此时由于系统调用进入了内核态，权限发生了改变，所以必须显示地检查访存的合法性。这充分体现出为确保系统安全而带来的复杂性。

该文件其次定义了一个函数指针组成的数组，其中包含各个系统调用对应的服务程序，按中断类型号排列。

该文件的最后是 syscall 函数。该函数较为简单：判断当前系统调用号是否合法，若合法则执行相应服务程序，若不合法则输出错误信息。此外，为了给用户程序在%eax 寄存器中提供返回值，系统调用会将返回值存放在当前中断帧对应的%eax 位置。这样，在最终进行 trapret 时，返回值会被正确地弹出到 texttt%eax 寄存器中。

sysproc.c

实现了各个系统调用服务程序。由于具体实现与中断异常的机制无关，故不详细分析。

2 小结

该部分代码实现的中断与异常机制可分为初始化和中断异常处理两部分；与课程中介绍的中断与异常机制总体相同，

该部分代码较为突出的特点有：

- 对权限进行精细的检查以确保安全性。课程中介绍的中断异常机制对访问权限的介绍相对较少，而在具体实现中，随处可见对访存权限进行的细节处理：例如将系统调用设置为唯一的权限为 DPL_USER 的中断，在系统调用的服务程序中显示检查用户的访存权限。

- 涉及硬件方面的细节。课程中对硬件层面的介绍较少，但在 xv6 的具体实现中涉及了中断控制器 APIC、体系结构提供的 IDTR 寄存器等的操作，`lapic.c` 等文件就包含了大量与硬件交互的接口函数，此部分也调用了其中部分函数。
- 高度模块化。本部分源码中进行中断、异常和系统调用处理的核心函数是 `trap` 函数，但该函数并不长；其它文件通过宏定义、辅助函数定义等使此部分更加易读。
- 该部分代码用脚本文件生成了 `vectors.S`，免去了枯燥重复的汇编指令编写。