第12章 陷入与系统调用

本章说明系统处理陷入的一般方式以及处理系统调用的特殊性。

有很多条件会造成处理机“陷入”。其中有些显然是出错条件，例如硬件或电源故障，UNIX并不企图采用任何相当复杂的方法进行故障恢复。

我们的注意力集中于“trap.c”文件中的主要过程。

12.1 trap(2693)

调用此过程的方法在第10章中已经说明。用汇编语言编写的“trap”例程执行某些基础性的内务处理，建立起了核心栈中的有关部分，所以当调用此过程时，似乎一切都已准备妥当。

“trap”过程可视为是由另一个“C”过程带7个参数以一般方式调用的，这7个参数及它们的顺序是：

dev,sp,r1,nps,r0,pc,ps

顺便要在这里提及的是一个特殊之处。通常，传送给“C”过程的所有参数都采用传值方式。如果被调用的过程在执行中更改了参数值，这不会直接影响到调用过程。但是若“trap”或中断处理程序更改了它们的参数值，那么这些新值在恢复“前状态”寄存器时都可以取得。

在陷入后立即取得处理机状态字的值，然后掩蔽除最后5位的其他各位，这样就可获得“dev”的值。在进行掩蔽处理之前，用相应矢量单元中的原型值以及陷入前的处理机状态设置处理机状态字。若矢量单元第2个字的值是“br7+n；”(0516行)，那么“dev”的值将是n。

2698：“savfp”保存浮点寄存器(对于PDP11/40来说，这不执行任何操作!)。

2700：若前状态是“用户态”，则对“dev”的值进行修改，使其加8进制值020(2662)。

2701：将栈中存放r0的单元地址存放在“u.u_ar0”中以供以后引用。(结果是：各寄存器值可用u.u_ar0〔Rn〕引用。)

2702：这是一个依赖于“dev”值的多路选择语句。

到此为止我们可以观察到，按照处理机前状态及陷入原因UNIX将陷入分成三类：

1) 核心态。

2) 用户态，并非由“trap”指令造成的陷入。

3) 用户态，由于“trap”指令造成的陷入。

12.2 核心态陷入

除一个例外，这种陷入是不能预期的。对此种陷入的处理是执行“panic”。所执行的代码是“switch”语句的“default”部分：

2716：打印：

• 第7核心态段地址寄存器的当前值(亦即当前的ppda(每个进程数据区)的地址)。

• 在核心态栈中存放“ ps ”单元的地址。

• 陷入类型编号。

2719：调用“ panic ”，该过程并不返回。

浮点操作仅由用户态程序使用，核心态程序不使用浮点操作。因为在 PDP11/45和PDP11/70中浮点操作是以异步方式执行的。所以当一个浮点异常发生时，处理机可能已切换至核心态进行中断处理。于是，在核心态下发生的浮点异常陷入可以判断为是由当前用户程序造成的。

2793：调用“ psignal ”(3963)，它设置一标志表示已发生浮点异常。

2794：返回。

在这里可以提示一个有趣的问题：“ 为什么对核心态和用户态下发生的浮点异常的处理会稍有不同呢？”

12.3 用户态陷入

首先考虑并非由执行“ trap ”指令而产生的陷入。这种陷入是由可能的出错造成的，操作系统对这种出错没有明确规定的处理方法 (除可能需要的内存映像转储外)，而用户程序本身可能对此已预作准备并包含有适当的处理方法。这方面的内容将在下一章中展开。在本章中主要说明的是：如何“ 通知 ”(signal)已发生此种陷入。

2721：在进程处于用户态运行时发生一总线错，将“ SIGBUS ”值(0123)赋予变量i。

2723：“ break ”语句的作用是从“ switch ”语句中跳出并转移至2818行。

2733：除源代码注释中指出的一个特殊情况外，对非法指令的处理在此阶段与对总线错的处理相同。

2739：

2743：

2747：

2796：参见2721行的注释。

注意：此“ switch ”语句(2702)中的2个“ case ”——“
(程序中断)由“ default ” case(2715行)处理。

2810：这一case(9+USER)要求操作系统提供帮助以扩
汇编语言例程“ backup ”(1012)用于重新构造在执行
用户栈的实际扩展则由“ grow ”(4136)实现。

“ backup ”过程相当繁琐，它必须仔细地考虑到 PDP11
留给对此有兴趣的读者研究分析。

正如在注释(2800、2809)中对PDP11/40所说明的那样
息以解决各种可能发生的情况，所以“ backup ”并不保证
2818：调用“ psignal ”(3963)以设置适当的“ 信号 ”。
句的多case才转移到此条语句。)

2821：“ issig ”检查现在或稍早一些时间是否将一“ 信号 ”送给了此用户程序，并且对此“ 信号 ”尚未进行适当处理。

2822：“ psig ”执行适当的处理动作。(在下一章将详细讨论issig和psig。)

2823：重新计算当前进程的优先数。

12.4 系统调用

“ trap ”指令是系统调用机构的重要组成部分，用户态程序用该指令要求操作系统提供帮助。

因为“ trap ”指令可以有很多种样式，所以在“ trap ”指令中可以包含所要求帮助的具体类型。

参数是系统调用的组成部分，可以用下列几种方式将用户程序中说明的参数传送给操作系统：

1) 通过专用寄存器r0。

2) 跟随在“ trap ”指令之后作为程序正文的组成部分。

3) 作为程序数据区的一部分(这是“ 间接 ”调用)。

间接系统调用的开销比直接系统调用高。当在编译时不能决定参数值时需要使用间接系统调用。如果只有一个依赖于数据的参数，那么可以通过 r0传递该参数，于是也就可以避免使用间接系统调用。在选择哪一个参数用 r0传送方面，UNIX系统的设计者是按照他们自己的经验行事的，我们很难对此作进一步的讨论。

“ C ”编译程序对系统调用并不进行特殊的识别，只是将它们视为一般的过程而进行同样的处理。当装入程序解决未确定的引用时，它使用相应的库例程，而库例程中则包含实际的“ trap ”指令。

2752：重置错指示器。

2754：取得造成陷入的用户态指令，除最后 6位外掩蔽该指令字中的其他各位。按该操作所得的结果(亦即指令字中的最后 6位)从“ sysent ”结构数组中选择一项，并将所选项的地址存放在“ callp ”中。

2755：＃0系统调用是“ 间接 ”系统调用，经其传送的参数是位于用户程序数据空间中一个系统调用及其参数序列的起始地址。

注意，程序中分别使用了“ fuword ”和“ fuiword ”。这两者之间的区别在 PDP11/40中是不重要的，但是对于指令空间和数据空间分开的机型而言，则是非常重要的。

2760：“ i=077 ”对应于最后一个系统调用 (2975)，实际上没有这种系统调用，其结果是调用“ nosys ”(2855)，它构造一种对用户态程序而言通常是致命性的出错条件。

2762：

2765：在“ sysent ”中指定的参数数是用户程序员实际提供的参数数，或者如果一个参数是通过r0传送的，则“ sysent ”中指定的参数数较用户程序员提供的参数数减 1。将各参数从用户数据或指令区复制到“ u.u_arg ”数组中，该数组长度为 5(从sysent中可见，该数组只要有4项就足够了，这可能是为了将来有一定的扩充余地)。

2770：第1个参数值被复制到“ u.u_dirp ”中，“ u.u_dirp ”的主要作用似乎是一个使用方便的临时存储单元。

2771：调用“ trap1 ”，其参数是所希望的系统例程的地址。注意从 2828行开始的注释。

2776：发生一错误时，设置原处理机状态字的“ C位 ”(见2658)，并通过 r0返回出错编号。

12.5 系统调用处理程序

在“ sysent.c ”文件中可以看到全套系统调用，而在 UPM的第Ⅱ部分则对它们进行详细说明。

处理系统调用的各过程绝大部分在“ sys1.c ”、“ sys2.c ”、“ sys3.c ”和“ sys4.c ”中。

另外两个重要过程是“ nullsys ”(2855)和“ nosys ”(2864)，它们在文件“ trap.c ”中。

12.6 文件sys1.c

此文件包含5个系统调用的处理过程，其中 3个在此处说明，另外 2个(rexit和wait)在下一章讨论。

本文件中的第1个过程，也是我们曾涉及过的第 1个系统调用是“ exec ”。

12.6.1 exec(3020)

本系统调用的编号是 11，它将执行某一个程序的进程更改为执行另一个不同程序的进程。请参见UPM中的EXEC(II)部分。exec执行的代码最长也是最重要的系统调用之一。

3034：“ namei ”(6618)(在第19章将对 namei进行详细讨论)将第1个参数(它是一个字符串指针，该字符串定义了新程序的名字)变换成一个“ inode ”引用。(inode是文件引用机构的关键部分。)

3037：若当前正执行中的“ exec ”数太大，则等待。(见3011行的注释。)

3040：“ getblk(NODEV) ”的作用是在缓存池中分配一个 512字节的缓存。用此缓存在内存中临时存放当前正在用户数据区中且启动新程序需要的信息。注意，在“ u.u_arg ”中的第2个参数是指向此信息的指针。

3041：若此文件是不可执行的，则“ access ”返回一非0值。第2个条件测试该文件是否是一个目录或一个特殊字符文件。(早一点进行这种测试，例如在 3036行之后立即进行测试似乎会改善这段代码的运行效率。)

3052：将位于用户空间的参数集复制到临时缓存中。

3064：如果由于参数字符串太长而不能全部存放在缓存中，则出错返回。

3071：如果参数字符串中的字符数是奇数，则增加一个 null字符。

3090：将命名文件的前4个字(8个字节)读入“ u.u_arg ”。从3076行开始的注释对这些字的意义进行了说明，更详细完整的解释请见 UPM的A.OUT(V)部分。

注意，设置“ u.u_base ”、“ u.u_count ”、“ u.u_offset ”和“ u.u_segflg ”，为读操作做好准备。

3095：如果无需对正文段进行保护，则将正文区的长度加至数据区长度，并将前者设置为0。

3105：检查该程序是否有一“ 纯 ”正文区，若不包含纯正文区，则进一步检查该程序是否已被其他程序作为一个数据文件而打开。如果是这样，则出错终止。

3127：到达此点时，执行此新程序的决定不再能逆转，也就是不再有机会设置好一个出错标志并返回到原先执行的程序。

3129：“ 扩展 ”在这里实际上意味着收缩至只保留 ppda区。

3130：“ xalloc ”分配正文区(若需要)，并与其相连接。

3158：将存放在缓存区中的信息复制至新程序的用户数据区的栈中。

3186：将核中栈中包含用户态寄存器先前值的各单元清 0，唯一例外是栈指针 r6，它已在3155行中设置。

3194：使对该inode结构的引用数减1。

3195：释放临时缓存。

3196：唤醒在3037行等待的进程。

12.6.2 fork(3322)

在调用“ exec ”之前经常先调用“ fork ”。“ fork ”的大部分工作是由“ newproc ”(1826)完成的，而在调用“ newproc ”之前，“ fork ”在“ proc ”数组中先搜寻一空闲项，然后用“ p2 ”记住该项的位置(3327)。

“ newproc ”也独自搜寻“ proc ”。可以推测它找到的空闲项与“ fork ”找到的是同一项。(为什么不会产生混乱呢?)

3335：对于新进程，“ fork ”返回父进程标识的值，并为各个计数参数赋初值 0。

3344：对于父进程，“ fork ”返回新创建的子进程标识的值，并使用户态程序计数器 (pc)值加2(一个字)。

注意，最后返回给“ C ”程序的值与上面所说明的是稍有不同的。请参阅 UPM的FORK(II)部分。

12.6.3 sbreak(3354)

此过程实现＃17系统调用，该系统调用在 UPM的“ BREAK(II) ”中说明。此过程开始部分的注释使很多读者感到迷惑：在“ C ”程序中使用标识符“ break ”，其作用是跳出一个程序循环；在这里，“ sbreak ”的作用是更改程序数据区的长度。这两者是完全不同的。

“ sbreak ”与过程“ grow ”(4136)非常相似但又有很多不同之处。“ sbreak ”只能显式调用，而且事实上它不但可以扩展，同样也可以收缩进程数据区 (取决于所希望的新长度)。

3364：计算数据区所需的新长度 (单位：32字块)。

3371：检查新长度是否满足对存储段的限制。

3376：数据区正在缩小。将栈区向下复制到以前的数据区中。调用“ expand ”裁去多余部分。

3386：调用“ expand ”增加总的区域。将栈区向上复制到新增部分，并清除该栈以前占用的存储区。

下列过程也包含在“ sys1.c ”中，在第13章中将对它们进行说明：

rexit(3205) **wait(3270)**

exit(3219)

12.7 文件sys2.c和sys3.c

sys2.c和sys3.c主要与文件系统及输入/输出有关。这两个文件位于操作系统源代码的第四部分。

12.8 文件sys4.c

此文件中的所有过程实现系统调用。下列过程在第 13章中说明：

ssig(3614) **kill(3630)**

下列过程简单明了，留给读者阅读并从中得到欢乐和教益：

```
getswit (3413)    sync    (3486)
gtime   (3428)    getgid  (3472)
stime   (3429)    getpid  (3488)
smtaid  (3439)    nice    (3493)
gmtaid  (3452)    times   (3656)
getgid  (3468)    profil  (3667)
```

下列过程与文件系统有关，将在后面的有关章节中对它们进行说明：

unlink(3510) **chown (3375)**

chdir (3538) **smdate(3595)**

chmod (3560)