第13章 软件中断

本章主要涉及文件“ sig.c ”的内容。“ sig.c ”文件引进了进程之间通信的一种新机制。特别地，这种新机制提供了一种使“ 用户态 ”进位可被中断、转移或终止的方法，而造成这种中断、转移或终止的原因可以是另一个进程的动作、击错或操作员的操作。

在本章的讨论中，我们经慎重考虑特地用“ 软件中断 ” (software interrupt)代替术语“ 信号 ” (signal)。“ 信号 ”在UNIX环境中的涵义与普通英语中的用法有相当区别，所以为了不引起误解，我们避免使用该术语。

UNIX识别20种(见0113行的NSIG定义)不同类型的软件中断，如果读者仔细阅读 UPM的SIGNAL(II)，那么可以发现其中的13种具有标准名字和含意。中断类型＃0被解释为“ 无中断 ” (no interrupt)。

在每个进程的ppda区中有一整型数组“ u.u_signal ”，其长度为“ NSIG ”。其中每一项对应于一个不同类型的软件中断，并定义了该进程遇到此种软件中断时应采取的动作 (如表13-1所示)。

表13-1 软件中断的种类及涵义

u_signal〔n〕	当＃n中断发生时
0	此进程将终止自身
奇数，非0	忽略此软件中断
偶数，非0	该值为用户空间中一过程的起始地址，应立即执行该过程

中断类型＃9(SIGKIL)是非常特殊的一种， UNIX保证“ u.u_signal〔9〕”的值一直为0，所以若进程由于“ SIGKIL ”而中断，则它总是终止自身。

13.1 设置期望动作

每个进程都可设置它自己的“ u.u_signal〔 〕”数组中各项(除u.u_signal〔9〕以外)的值，以指定将来发生相应类型中断时应采取的动作。用户程序的编写人员用“ signal ”系统调用实现这一类(参见UPM中的SIGNAL(II))。

例如，若程序员希望忽略＃2软件中断 (用户在终端上击删除键，则产生此软件中断)，那么他应当在“ C ”程序中，用下列系统调用将“ u.u_signal〔2〕”设置为1：

“ `signal(2,1);` ”

13.2 对进程造成中断

在进程“ proc ”结构中包含有“ p_sig ”项(0363)，将其值设置为某一中断类型编号 (亦即在1至NSIG－1之间的一个值)，那么对该进程就造成了一个中断。

即使所影响进程及其ppda区已换出至磁盘上，也总是可以直接存取“ p_sig ”。显然，这种机构只允许每一个进程在任一时刻仅有一个中断等待处理。该等待处理的中断总是“ p_sig ”即最近一次设置的值所对应的中断，唯一的例外是类型为 9的中断，一旦将“ p_sig ”设置为9，那么后面发生中断时就不再能重新设置“ p_sig ”，也就是说＃ 9 中断是占绝对优势的。

13.3 作用

软件中断的作用不会立即产生。如果受影响的进程当前正在运行，那么经短时间延迟后，软件中断的作用就可能产生。如果受影响的进程已被挂起或者其映像已被换出到磁盘上，那么软件中断就可能经过相当长的时间后才能起作用。

对于软件中断的规定动作总是由受影响的进程自己执行并施加到自身的，因此只有当所影响的进程是活动进程时，软件中断的作用才可能产生。

如果软件中断的相应动作是执行一个用户定义的过程，那么为了实现这一点，核心态进程要对用户态栈作调整使其符合下列状态：刚刚进入该软件中断对应的用户定义过程，在执行其第1条指令之前立即被中断(硬件类型)。于是当系统从核心态以一般方式返回至用户态时，由于用户栈调整后的作用，下一条执行的用户态指令就是指定过程的第 1条指令。

13.4 跟踪

UNIX在软件中断设施基础上稍加扩充又提供了一种非常强有力但效率在某种程度上较低的机构，使用这种机构父进程可以监控其一个或多个子进程的进展状况。

13.5 过程

因为与软件中断相关的各过程之间的关系初看起来会
们之前，先对它们作一些简要说明。

13.5.1 期望动作的设置

“ ssig ”(3614)实现＃48系统调用，其功能主要是设置数

13.5.2 造成软件中断

“ kill ”(3630)实现＃37系统调用，其功能是对指定标识
“ signal ”(3949)对由一个指定的终端所控制的或启动的
“ psignal ”(3963)的作用是实际设置“ p_sig ”，它由“ k
(trap(2793、2818)和pipe(7828)也调用psignal)。

13.5.3 作用

“ issig ”(3991)，现活动进程用其测试是否接到一个等
“ sleep ”(2073、2085)、“ trap ”(2821)和“ clock ”(3826)调
“ psig ”(4043)实现由软件中断触发的动作，当“ issig

的例外出现在“ sleep ”，其中的处理稍复杂一些。

“ core ”(4094)实现进程内存映像的转存，如果需要为一个终止进程进行映像转存，则“ psig ”调用此过程。

“ grow ”(4136)扩展用户态栈区，在需要此功能时“ psig ”调用此过程。

“ exit ”(3219)终止当前活动进程。

13.5.4 跟踪

“ ptrace ”(4165)实现＃26“ ptrace ”系统调用。

“ stop ”(4016)，一个要求父进程跟踪的进程在“ issig ”中检测到一信号后立即调用此过程，使监视父进程有机会查看此子进程。

“ procxmt ”(4204)由“ stop ”(4028)调用，被跟踪子进程藉此按父进程的命令执行与跟踪相关的操作。

13.6 ssig(3614)

此过程实现“ signal ”系统调用。

3619：若软件中断源超出了范围或者等于“ SIGKIL ”(9)，则出错退出。

3623：取“ u.u_signal〔a〕”的原来值，在“ signal ”系统调用结束时将其作为执行结果返回。

3624：将“ u.u_signal〔a〕”设置为所希望的值。

3625：若当前本进程已接到一个类型为“ a ”的软件中断并等待处理，则取消该软件中断。

半不清楚，你有什么建议吗 ?)

更对另一个指定的进程造成一指定类型的软件中断。

数为“ a ”的进程是要被中断的。若“ a ”为0，那么与当都是要被中断的。

每一项，若该项满足下列条件之一，则跳过：

，而且该项所代表的进程并不与当前进程具有同一控制终(亦即＃0、＃1进程)之一(3644)。

user)，而且该项所代表进程的用户标识数 (p_uid)并不与当6)。

跳过)的任一进程调用“ psignal ”以更改“ p_sig ”。

” 表示)控制的进程调用“ psignal ”。

13.9 psignal(3963)

3966：若“ sig ”太大则立即返回 (但是为什么对 sig是负值的情况不进行检查呢 ?在将该参数传向 psignal之前， kill并不对其进行检查。我们实际上认可 kill命令只会使 sig的值为正)。

3971：若“ p_sig ”的当前值不是“ SIGKIL ”，则设置新值(亦即，一旦进程已被杀死，那么就无法使其复生)。

3973：这里似乎是一个错误，不是“ p_stat ”而应当是“ p_pri ”。如果此进程的优先权不高，则提高其优先权(将其设置为较其原先值小的 PUSER)。

3975：若该进程正等待一非内核事件，亦即该进程以正优先数值调用“ sleep ”(2066)，则再一次将其设置为运行态。

13.10 issig(3991)

3997：若“ p_sig ”非0，则……

3998：若“ 跟踪 ”标志已设置，则调用“ stop ”(以后还将继续此话题)。

4000：若“ p_sig ”为0，则返回0(因为stop在执行时作为一副作用可以清 p_sig，所以这一条看起来冗余的测试是必须的)。

4003：若“ u.u_signal ”数组中相应元素的值为偶数 (可以是0)，则返回一非0值。

4006：否则返回0。

从3983至3985行的注释与调用“ issig ”的频度有关，对此需要作些说明。每次执行“ trap ”时至少调用一次“ issig ”，唯一的例外是“ clock ”中断例
在“ pri ”为正值时，“ sleep ”在调用“ swtch ”的前、后各i

13.11 psig(4043)

若“ issig ”测试到“ u.u_signal〔n〕”的值为偶数，则
一情况。若此值非0(4051)，则将其视为应执行的用户态地址

4054：除非此软件中断是由一条非法指令或一跟踪陷入
设置为0。

4055：计算用户空间中较当前栈顶低两个字的地址，从

4056：调用“ grow ”，它先检查当前用户态栈的长度，

4057：将处理机状态寄存器和程序计数器的值送入用户
件中断 ”时获取并存入核心栈的，然后更新核心栈中存放的
的处理后，在返回到用户态时将从指定过程的开始处恢复
先被中断的过程将恢复执行。

4066：若“u.u_signal〔n〕”为0，则对于4066行至4074
产生进程内存映像转储。

4079：在“ u.u_arg〔0〕”中存入一值，该值是由两部

放的r0的低字节，另一部分是“ n ”，在“ n ”中记录了软件中断的类型以及是否执行成功一内存映像转储操作。

4080：调用“ exit ”终止本进程。

13.12 core(4094)

此过程将可换进换出的程序映像复制到位于用户当前目录的“ core ”文件中。为了详细解释此过程，需先了解本书第三、四部分中与输入 /输出和文件系统有关的内容。

13.13 grow(4136)

本过程的参数“ sp ”定义了用户态栈中应当包括的一个字的地址。

4141：若此栈长度足够大，则以0值立即返回。

注意，此测试依赖于2的补码运算的特异性。若满足下列两个条件时，作出扩展栈的决定则会是错误的：

- ｜sp｜＞2＾15
- ｜u.u_size×64｜＞2＾15

4143：计算所需的栈长度增量以包括参数“ sp ”指向的栈单元以及20×32个字的余量。

4144：检查此值确定是正值，亦即，我们没有处理 4141行测试的一种失误。

4146：检查扩展后新栈的长度并不与存储段长度的限制相冲突 (若产生冲突，则estabur设置u.u_error),然后清段寄存器原型。

4148：得到一个新的经扩展后的数据区，将栈段复制至新数据区的高地址端 (每次32个字),然后清除栈扩展部分。

4156：更新栈长度“ u.u_ssize ”，返回表示成功执行的结果。

13.14 exit(3219)

当一个进程要终止自身时，调用本过程。

3224：清除“ 跟踪 ”标识。

3225：将数组“ u.u_signal ”中的所有元素值(包括u.u_signal〔SIGKIL〕）设置为1以便在将来执行“ issig ”时不会随之执行“ psig ”。

3227：调用“ closef ”(6643)以关闭该进程已打开的所有文件。 (closing的大部分工作只是简单地将打开文件的引用计数器减 1)。

3232：将当前目录的引用计数减 1。

3233：对进程所使用的正文段有关信息进行处理。

3234：需要找到一个区域以存放“ 每个进程 ”（per process)信息，这样，父进程以后能查看到这些信息。在磁盘可交换区中为此分配一块 (256字)是合适之举。

3237：找到一合适的缓存(256字)，并且……

3238：将“ u ”结构的前半部分复制到该缓存中。

3239：将该缓存中的内容写至磁盘交换区。

3241：将该进程占用的内存空间释放回空间列表。自然，该内存区仍可使用。当其他进程从空间表中分配到这一内存区后，其中的信息也就会改变而不再能使用。这种情况是不会立即发生的，其原因是“ getblk ”和“ bwrite ”在执行过程中会调用“ sleep ”(后面会进一步说明这一点)，而在此进程睡眠过程中，什么事情都可能发生。观察到所有这些后，那么在 3226 行之后插入下列语句是合理的：

“expand(USIZE)”；

3243：将进程状态设置为“ 等待善后处理 ” (zombie)。

3245：余下部分的代码在“ proc ”数组中搜寻父进程，找到后立即唤醒它。如果子进程处于等待善后处理状态，则父进程进行必要的处理；若子进程位于暂停状态以便跟踪，则释放它们。最后，此代码包含了本进程对“ swtch ”的最后一次调用。

在分析说明与跟踪有关的程序之前，先分析两个与“ exit ”密切相关的例程。

13.15 rexit(3205)

此过程实现＃1系统调用“ exit ”。它先取得用户提供参数的低位字节，然后将该字节存入“ u.u_arg〔0〕”，而“ u.u_arg〔0〕”位于“ u ”结构的前半部分，而该部分在进程进入“ zombie ”状态时被写至磁盘交换区。

13.16 wait(3270)

对应于每一次“ exit ”调用，都应当有一个由其父进程或祖先进程发出的与其匹配的“ wait ”调用。实现“ wait ”系统调用过程的主要功能是找到并处理“ zombie ”子进程。

“ wait ”也有一个辅助功能，它搜寻因“ 跟踪 ”而暂停的子进程（“ 跟踪 ”是下一个要谈论的话题)。

3277：搜索整个“ proc ”数组，寻找子进程(如果没有子进程，则出错终止(3317))。

3280：如果子进程是一个“ zombie ”进程：

- 保存子进程标识数，以便父进程以后需要时取用。
- 从磁盘交换区读回256字记录，然后释放该磁盘交换区。
- 再初始化该“ proc ”数组项。
- 累计各种时间计算项。
- 也保存“ u_arg〔0〕”值，以便父进程以后需要时取用。

3298：最后，释放缓存区。

3300：检查子进程是否是一个处于“ 暂停 ”状态的进程。 (如果是这样，则对其的处理方式将在下面讨论)。

3313：如果找到1个或多个子进程，但它们都不是“ zombie ”或“ 暂停 ”进程，则调用“ sleep ”进入睡眠状态，以后被唤醒后再次搜寻。

13.17 跟踪

UNIX对软件中断设施进行修改和扩充，在此基础上提供了跟踪设施。简而言之，如果一

父进程正在跟踪子进程的进展，那么当子进程每次遇到一软件中断时，作为对该软件中断总响应的一部分，父进程就得到插入进行干预的机会。

父进程的干预可以包括对子进程数据区中各个值的查询，该区中包含有 ppda区。在受到某些限制的前提下，父进程也可以更改该区中的值。

软件中断源可以是父进程、用户自身 (例如，在其终端上输入 kill命令或击“ 删除 ”键)或者子进程自身(例如，若它执行非法指令或其他造成出错的操作)。

父、子进程之间的通信是一种礼仪性的规范化活动。

1) 子进程遇到一软件中断，然后“ 暂停 ”。

2) 正等待的父进程发现此“ 暂停 ”的子进程 (3301行)，然后使其复苏。其后……

3) 父进程可以执行“ ptrace ”系统调用，它将对子进程的一条要求消息放入系统定义的“ ipc ”结构中(3939).

4) 然后父进程调用“ sleep ”进入睡眠状态，同时子进程则是苏醒的。

5) 子进程读“ ipc ”中的消息，然后按照该消息中的要求进行动作 (例如，将它自己的一个值复制到ipc.ip_data中)。

6) 然后，子进程调用“ sleep ”，同时父进程则是苏醒的。

7) 父进程查看记录在“ ipc ”中的结果。

8) 可以顺次重复第3至第7步若干次。

最后，父进程可以允许子进程继续正常执行，甚至该子进程可能完全不知道已经发生过软件中断。

关于跟踪设施的讨论请参见U PM的 PTRACE(II)部分。除了在“ 故障 ” (Bugs)段落中所说明的功能限制说明外，在效率方面我们可作如下注释：

- 应当提供每次能从子进程向父进程传输大块信息的机构，例如一次能传输 256个字，而从父进程向子进程每次传输的信息则不必这样多。
- 在父、子进程之间应当有一种合作例程过程 (类似于swtch),以便实施父、子进程之间控制的快速转移。

13.18 stop(4016)

若已设置跟踪标志(STRC(0395))，则“ issig ”调用此过程(3999)。

4022：若该进程的父进程是＃1进程 (亦即执行/etc/init的进程)，则调用“ exit ”(4032行)。

4023：否则在“ proc ”数组中查找父进程…，找到后将其唤醒，然后声明自身“ 暂停 ”，然后……调用“ swtch ”。(注意此处并不调用sleep,为什么?)

4028：如果跟踪标志已复位，或者过程“ procxmt ”的执行结果为真，则返回至“ issig ”。

4029：否则再从头执行一次。

13.19 wait(3270)(继续)

3301：如果该子进程已“ 暂停 ”，并且……

3302：如果该子进程的“ SWTED ”标志没有设置(亦即，父进程最近没有通知过此子进程)……

3303：为了帮助记忆，设置“ SWTED ”标志。设置“ u.u_ar0〔R0〕”和“ u.u_ar0〔R1〕”以便将子进程状态字(子进程标识数和子进程接到的信号)返回给父进程。

3309：“ SWTED ”标志已设置。这意味着父进程连续执行至少 2次“ wait ”，其中没有调用“ ptrace ”,因它对此子进程并无兴趣。复位该子进程的“ STRC ”和“ SWTED ”标志，然后释放该子进程。 (注意这里调用的是setrun而非wakeup，这样就可以和swtch调用(4027)配对)。

13.20 ptrace(4164)

此过程实现＃26系统调用“ ptrace ”。

4168：“ u.u_arg〔2〕”对应于“ C ”程序调用序列中的第1个参数。若其为0，则一个子进程正要求被其父进程跟踪，为此设置“ STRC ”标志，然后立即返回。

注意，这段代码处理在跟踪中要求子进程采取的唯一显式的动作。为什么这种动作应由子进程采取而不是由父进程实施呢?似乎没有什么实际的理由。从安全角度考虑，非常希望只有当子进程给出许可后，它才是可被跟踪的。在另一方面，如果子进程要求父进程对它进行跟踪，但是父进程却忽略了此种要求，那么造成的后果可能是子进程无限期地阻塞。最好的解决方法可能是：只有在父、子进程双方都显式地提出跟踪和被跟踪要求后，才设置“ STRC ”标志。

4172：搜索“ proc ”表，查找满足下列条件的进程：

- 该进程处于“ 暂停 ”状态。
- 与给出的进程标识数相匹配。
- 是当前进程的子进程。

4181：等待“ ipc ”结构变成可用的(如果当前它正被使用)。

4183：将参数复制到“ ipc ”中……

4187：清“ SWTED ”标志，然后……

4188：将该子进程的状态设置为“ 准备运行 ”。

4189：睡眠直至“ ipc.ip_req ”值非正(4212)。

4191：取出应返回给父进程的值，检查是否出错，解锁“ ipc ”，最后，唤醒因“ ipc ”而睡眠的进程。

注意，在4182和4190行都要求进入睡眠状态，但是它们的原因在本质上是有区别的，为了将两者区分开来以获得更好效果，可以将4190和4213行中的“ &ipc ”代换成“ &ipc.ip_req ”。

13.21 procxmt(4204)

依据父进程送入“ ipc ”中的数据，子进程以不同路径执行本过程。

4209：若“ ipc.ip_lock ”对当前进程进行错误地设置，则确实应当忽略“ ipc ”的余下部分，立即返回。

在“ stop ”(4027)调用“ swtch ”后，“ exit ”，“ wait ”和“ ptrace ”可分别调用“ setrun ”重新启动此子进程，而这些“ setrun ”使“ STRC ”和“ SWTED ”标志成为下面指出的状态：

		STRC	SWTED	ipc.ip_lock
exit	(3254)	设置	设置	任意
wait	(3310)	清0	清0	任意
ptrace	(4188)	设置	清0	正常设置

在上列第3种情况中，在重新启动子进程之前，“ ptrace ”总是正确地设置“ ipc.ip_lock ”，所以4209行的测试不会失败。

在第2种情况中，父进程已经忽略了子进程，事实上决不会调用“ procxmt ”。

在4210行执行语句“ return(0) ”，“ procxmt ”强迫“ stop ”回到4020行。在父进程已经终止情况下，在4022行的测试将失败，于是造成调用“ exit ”(4032)。

4211：在清“ ipc.ip_req ”之前，存储其值，“ 唤醒 ”父进程，然后按已经存储的“ ipc.ip_req ”值选择下一个动作。

“ ptrace ”中针对“ ipc.ip_req ”的不同值而采取的各种动作在UPM的PTRACE(II)中都有所说明，但其中对1、2和4、5两种情况的说明中，关于指令空间和数据空间的有关部分有错误，把指令空间和数据空间正好说反了。