# 信号系统

进程在用户态下运行时，若发生软、硬件故障或执行到某些陷入指令则进入核心态。但是在核心态下运行的陷入处理程序对它们并不直接进行处理。信号机构指的是系统中围绕信号的产生、传送和处理而构成的一套机构。除了故障和特殊陷入指令处理以外，信号机构还用于：用户态进程之间以传送信号方式进行简单通讯。

进程运行时，经常检查自己是否接到信号。如果已接到信号则转而执行的程序（见图10.1）。

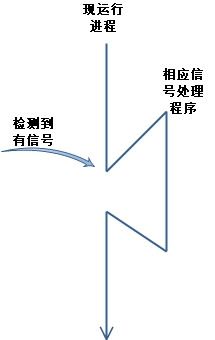


图10.1

这种处理方式与硬件中断非常类似。但是，信号的设置、检查都有是由软件实施的，所以信号有时也称为软中断，相应机构称为软中断机构。

信号机构包括三个部分：

1. 信号的分类、产生和传送。
2. 信号处理方式。
3. 信号检测时机与处理函数。

进程之间通过信号机构可以实施预定的相互控制，但是控制操作限于：控制进程向被控进程发送信号；被控制进程接到信号后执行预定的处理程序。控制进程不能对被控进程的图象进行检查和修改。而在某些场合，例如高级语言程序的调试软件中这种功能却是非常需要的。为此UNIX在信号机构基础上加以扩充，提供了父进程对子进程进行跟踪处理的机构。

## 信号的分类、产生和传送

### 信号类型

UNIX能处理25种不同的信号（NSIG = 25），信号类型编号为0-24。类型0被解释为没有接到信号；类型1-20的含义是系统规定的，而且有标准名称；类型21-24则由用户自己定义。20种信号列表如下：

|  |  |  |
| --- | --- | --- |
| 信号类型号 | 符号常数 | 注释 |
| 0 | （无） | 没有信号 |
| 1 | SIGDIV | 除法错 |
| 2 | SIGDEX | 调试异常 |
| 3 | SIGBKP | 断点 |
| 4 | SIGOVF | 溢出 |
| 5 | SIGBOC | 边界检查 |
| 6 | SIGIVP | 非法指令码 |
| 7 | SIGCNA | 协处理器不存在 |
| 8 | SIGDFT | 双重错误 |
| 9 | SIGCSO | 协处理器段超限 |
| 10 | SIGINT | 无效TSS段 |
| 11 | SIGSNP | 段不存在 |
| 12 | SIGSTE | 栈异常 |
| 13 | SIGGPE | 全局保护错误 |
| 14 | SIGPPV | 页面异常 |
| 15 | SIGNEM | 物理内存不足 |
| 16 | SIGAUM | 访问空页 |
| 17 | SIGCER | 协处理器错误 |
| 18 | SIGKIL | 结束本进程 |
| 19 | SIGBRK | Ctrl+Break中断 |
| 20 | SIGPIPE | 管道结束 |

表10.1 信号表

具体源码中的定义如下

static const int NSIG = 25; /\* 信号个数\*/

/\* p\_sig中接受到的信号定义\*/

static const int SIGDIV = 1; /\* Divide Error \*/

static const int SIGDEX = 2; /\* Debug Exceptions \*/

static const int SIGBKP = 3; /\* Breakpoint \*/

static const int SIGOVF = 4; /\* Overflow \*/

static const int SIGBOC = 5; /\* Bounds Check \*/

static const int SIGIVP = 6; /\* Invalid Opcode \*/

static const int SIGCNA = 7; /\* Coprocessor Not Available \*/

static const int SIGDFT = 8; /\* Double Fault \*/

static const int SIGCSO = 9; /\* Coprocessor Segment Overrun \*/

static const int SIGINT = 10; /\* Invalid TSS \*/

static const int SIGSNP = 11; /\* Segment Not Present \*/

static const int SIGSTE = 12; /\* Stack Exception \*/

static const int SIGGPE = 13; /\* General Protection Exception \*/

static const int SIGPPV = 14; /\* Page protection violation \*/

static const int SIGNEM = 15; /\* Not enough physical memory \*/

static const int SIGAUM = 16; /\* Access unallocated memory \*/

static const int SIGCER = 17; /\* Coprocessor Error \*/

static const int SIGKIL = 18; /\* kill \*/

static const int SIGBRK = 19; /\* Ctrl+Break \*/

static const int SIGPIPE = 20; /\* end of pipe \*/

代码10.1 信号类型定义

### 信号的产生与传送

在下列三种情况中产生信号并向有关进程传送。

（1）在陷入处理子程序trap中，针对用户态下产生的各种故障及使用的各种陷入指令产生不同类型的信号。然后交此信号送入该进程proc中的p-sig项。

（2）进程之间传送信号。进程之间通过系统调用kill进行信号传送。系统调用kill的格式是kill(pid,sig)。其功能是将信号sig送入一个或几个进程。当pid非0时，它就是接收信号进程的标识数。当pid为0时，则将信号sig送至与发送进程受同一终端控制的所有进程（除终端控制1#进程）。

Kill处理程序的基本工作就是按照前面叙述的条件在proc表中逐个检查，若发现该进程满足条件，则调用psignal将sig送入其p-sig项；如果在整个proc表中找不到满足上述条件的进程，则设置出错信号。Kill函数流程见图10.2，具体代码实现见代码10.2。Psignal函数流程见图10.3，具体代码见代码10.3。

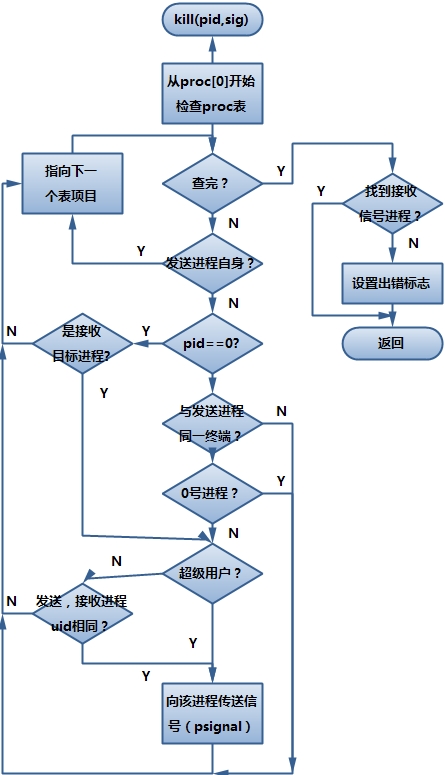


图10.2

int SystemCall::Kill()

{

User& u = Kernel::Instance().GetUser();

ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

int pid = u.u\_arg[0];

int signal = u.u\_arg[1];

bool flag = false;

for ( int i = 0; i < ProcessManager::NPROC; i++ )

{

/\* 不允许发送信号给进程自身\*/

if ( u.u\_procp == &procMgr.process[i] )

{

continue;

}

/\* 不是信号的接收方目标进程，继续搜寻\*/

if ( pid != 0 && procMgr.process[i].p\_pid != pid)

{

if ( pid == 0 && (procMgr.process[i].p\_ttyp != u.u\_procp->p\_ttyp || i == 0 ) )

{

continue;

}

/\* pid为，则将信号发送至与发送进程同一终端的所有进程，#进程不包括在内\*/

if ( pid == 0 && (procMgr.process[i].p\_ttyp != u.u\_procp->p\_ttyp || i == 0 ) )

{

continue;

}

/\* 除非是超级用户，否则要求发送、接收进程u.uid相同，即不可给其它用户进程发送信号\*/

if ( u.u\_uid != 0 && u.u\_uid != procMgr.process[i].p\_uid )

{

continue;

}

flag = true;

/\* 信号发送给满足条件的目标进程\*/

procMgr.process[i].PSignal(signal);

}

if ( false == flag )

{

u.u\_error = User::ESRCH;

}

return 0; /\* GCC likes it ! \*/

}

代码10.2 SystemCall::kill()

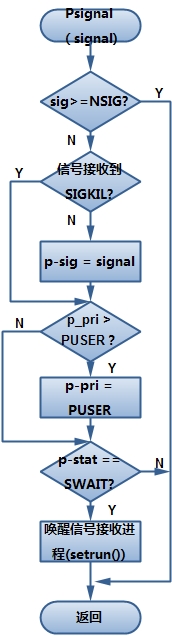


图10.3

void Process::PSignal( int signal )

{

if ( signal >= User::NSIG )

{

return;

}

/\* 如果已经接收到SIGKIL信号，则忽略后续信号\*/

if ( this->p\_sig != User::SIGKIL )

{

this->p\_sig = signal;

}

/\* 若进程的优先数大于PUSER(100)，则将其设置为PUSER \*/

if ( this->p\_pri > ProcessManager::PUSER )

{

this->p\_pri = ProcessManager::PUSER;

}

/\* 若进程的处于低优先权睡眠，则将其唤醒\*/

if ( this->p\_stat == Process::SWAIT )

{

this->SetRun();

}

}

代码10.3 Process::PSignal()

（3）进程将信息写入通讯文件pipe时，发现该文件读通道已经关闭，也就是没有一个进程将从中读取信息，则向本进程发送SIGPIPE信号，见代码10.4。

/\* 如果管道的读者、写者中已经有一方关闭，则返回\*/

if ( pInode->i\_count < 2 )

{

pInode->Prele();

u.u\_error = User::EPIPE;

u.u\_procp->PSignal(User::SIGPIPE);

return;

}

代码10.4 witrep中产生信号SIGPIPE

## 信号处理方式

### 三种处理方式

进程接到信号后，在一定时机对其进行处理。处理方式有三种：一种终止本进程，这是针对用户态下故障信号最经常采用的一种方式。第二种是忽略该信号，也就是不作任何处理。最后一种是执行一段预先编好的位于虚地址窨中的信号处理程序。一个进程对各种信号的处理方式都有记录在信号处理表u-signal[NSIG]中。如果相应元素值是0，说明对该信号采用的处理方式是终止本进程。如果相应元素值为奇数，说明对该信号不作任何处理。如为非0偶数，则它是信号处理程序入口地址。

### 信号处理方式的预置

u-signal[NSIG]表中各元素的值可用系统调用Ssig(sig,func)加以改变。参数sig表示信号类型号；func表示对该信号的处理方式。但是对于信号SIGKIL，其处理方式固定为终止接收到该信号的进程，不能用系统调用Ssig设置。

在Ssig中，先检查信号类型数是否合法。如果它不在1~24范围内或它是SIGKIL，则为非法信号设置，设置出错标志然后返回。如果信号类型合法，则将该信号原先处理方式送入栈中存放r0的单元，以便系统调用处理结束后回送。接着将信号处理方式func（在u-arg[1]中）送入信号表中相应项。如果该进程在此以前已接到这种信号，因为它的处理方式发生了变化，所以将其清除。具体实现详见代码10.5。

int SystemCall::Ssig()

{

User& u = Kernel::Instance().GetUser();

int signal = u.u\_arg[0];

unsigned long func = u.u\_arg[1];

/\* 这几个信号不许设置\*/

if ( signal <= 0 || signal >= User::NSIG || signal == User::SIGKIL )

{

u.u\_error = User::EINVAL;

return 0;

}

/\* 设置函数地址到信号处理函数数组\*/

u.u\_ar0[User::EAX] = u.u\_signal[signal];

u.u\_signal[signal] = func;

/\* 清当前信号\*/

if ( u.u\_procp->p\_sig == signal )

{

u.u\_procp->p\_sig = 0;

}

return 0; /\* GCC likes it ! \*/

}

代码10.5 SystemCall::Ssig()

## 信号检测时机与处理函数

### 信号检测的时机

针对不同的信号源，UNIX定点定时检查并处理信号。

1. 在陷入处理子程序trap末尾，检测该进程是否已接到信号，如果接到信号则调用psig程序。选择这一时机进行检测主要是针对各种故障陷入。一旦发生故障则进入陷入机构，在陷入处理子程序中将在用有态下产生的各种故障转换成信号，随之检测并对其进行处理。这种故障处理的速度是非常快的，于是就能避免故障造成的损害扩展开来，及时保持了故障产生时的现场。为各种特殊陷入指令形成的信号，以及在写pipe时产生的信号SIGPIPE也在此时进行检测。我们可以在trap末段看到这段处理：

/\* 判断有无接收到信号，如接收到信号则进行响应\*/

if ( u.u\_procp->IsSig() )

{

u.u\_procp->PSig(context);

}

代码10.6 trap末段对于信号的检测与处理

1. 在执行系统调用处理程序过程中，可能因各种原因多重嵌套调用到sleep程序。在进程进入低优先权睡眠状态前和被唤醒后，检查是否已接到不可忽略信号（参见代码10.7，代码10.8）。若已接到，立即停止执行sleep程序，直接跳转回trap程序，在其末尾同样作第一条中所述处理。选择这种时机的主要目的是加快低优先权睡眠进程在睡眠期中接到的信号的处理过程，同时也便于实施发送、接收信号进程之间的同步。

/\*

\* 进程在进入低优先权睡眠之前，以及被唤醒之后，如果接收到不可忽略

\* 的信号，则停止执行Sleep()，通过aRetU()直接跳转回Trap1()函数

\*/

if ( this->IsSig() )

{

aRetU(u.u\_qsav);

return;

}

代码10.7 sleep中 睡眠前对于信号的检测与处理

/\* 被唤醒之后再次检查信号\*/

if ( this->IsSig() )

{

aRetU(u.u\_qsav);

return;

}

代码10.8 sleep中 唤醒后再次检查信号

（3）若时钟中断前为用户态，则每隔1秒，时钟中断处理程序检测现运行进程是否已接到信号，若已接到信号，调用psig进行信号处理。见代码10.9：

/\* 如果中断前为用户态，则考虑进行信号处理\*/

if ( mode != 0 )

{

//u.u\_ar0 = &regs->eax;

if ( current->IsSig() )

{

current->PSig(context);

}

/\* 计算当前进程优先数\*/

current->SetPri();

}

代码10.9 时钟中断处理程序中检查信号

在上列三点中，第一、二点为定点检测，第三点则为定时检测。

### 信号处理函数

一个进程在适当时机要检测它是否已接到了信号，如果接到了信号，则按预定方式进行信号处理。检测有无接到信号的程序是issig，进行信号处理的程序是psig，两者的使用序列是：

if ( u.u\_procp->IsSig() )

{

u.u\_procp->PSig(context);

}

Issig()函数用来检测进程是否接收到信号。其具体实现见代码10.10：

int Process::IsSig()

{

User& u = Kernel::Instance().GetUser();

/\* 未接受到信号\*/

if ( this->p\_sig == 0 )

{

return 0;

}

/\* u.u\_signal[n]为偶数才表示对信号进程处理\*/

else if ( (u.u\_signal[this->p\_sig] & 1) == 0 )

{

return this->p\_sig;

}

return 0;

}

代码 10.10 Process::IsSig()

若未接受到信号，或u\_signal[p\_sig]的值为奇数，则返回0；若为偶数，返回p\_sig（非零）。

Psig()函数用来对信号进行处理。其流程图见图10.4，具体实现见代码10.11

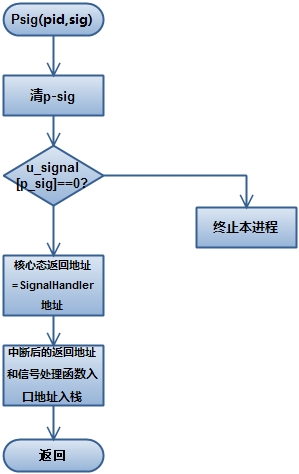


图10.4

void Process::PSig(struct pt\_context\* pContext)

{

User& u = Kernel::Instance().GetUser();

int signal = this->p\_sig;

/\* 清除已进入处理流程的信号\*/

this->p\_sig = 0;

if ( u.u\_signal[signal] != 0 )

{

unsigned int old\_eip = pContext->eip;

/\* 核心态返回值为预定的用户函数SignalHandler()的首地址\*/

pContext->eip = 0xC0000000 + ((unsigned long)SignalHandler - (unsigned long)runtime);

pContext->esp -= 8;

int\* pInt = (int \*)pContext->esp;

\*pInt = u.u\_signal[signal];

\*(pInt + 1) = old\_eip;

return;

}

/\* u.u\_signal[n]为，则对信号的处理方式是终止本进程\*/

u.u\_procp->Exit();

}

代码10.11 Process::PSig()

信号检测以及用终止方式对信号进行处理都有是在核心态下实施的。但是如果对接获信号的处理方式是执行一段用户自编程序，那么就必须在用户态下才能进行实际信号处理。比较合适的处理时机是进程由核心态恢复为用户态运行时。一般而言，一个进程在用户态下运行时，如若因中断或陷入转入核心态，则在有关处理结束后，或立即回复到原进程用户态，或经过一段时间间隔被再次切换调度进入处理机后，又恢复为用户态，从断点继续执行被子中断过的用户程序。如果需要插入执行一段信号处理程序，那么执行顺序变成如图10.5所示。

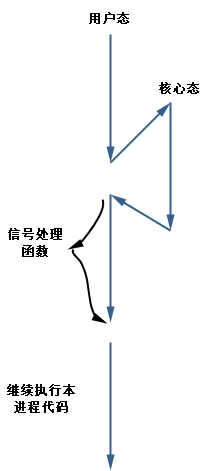


图10.5

问题是如何才能使断点eip变成信号处理程序入口地址u-signal[n]（n是接到的信号类型号）；在执行完信号处理程序后又臬继续从eip执行被中断过的用户程序？

其方法是：

（1）将中断后的返回地址和信号处理函数入口转存入用户栈栈顶，并相应修改存放在核心站中的用户指针（esp = esp - 8）。

（2）将系统固定的信号处理的程序入口地址送入原来核心栈的返回地址。这就保证了进程由核心态返回用户态时，立即执行系统固定的信号处理的程序。

此时的堆栈如图10.6所示：

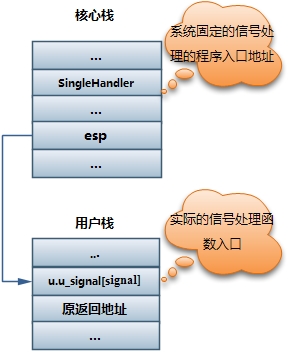


图10.6

系统固定的信号处理的程序的源代码如下：

extern "C" void SignalHandler()

{

\_\_asm\_\_ \_\_volatile\_\_(" leave; \

xchgl %%eax, (%%esp); \

pushl %%ebx; \

pushl %%ecx; \

pushl %%edx; \

pushl %%esi; \

pushl %%edi; \

pushl %%ebp; \

call \*%%eax; \

popl %%ebp; \

popl %%edi; \

popl %%esi; \

popl %%edx; \

popl %%ecx; \

popl %%ebx; \

popl %%eax; \

ret"::);

}

代码10.12 SignalHandler()

开头的 xchgl %%eax, (%%esp) 语句将EAX寄存器和栈顶的内容交换，这样保证了可以保护EAX寄存器的内容，同时也取出了栈顶的信号处理函数入口，供以后调用。

信号处理函数处理完后会恢复现场，返回时恢复了中断时的现场，进程继续从EIP开始执行被中断了的程序。