不是内核的专家，只是想通过了解内核来更好的理解linux app&driver编程.

**Sigaction in Linux**

简介：

信号供进程之间异步通讯所用。作为进程的一个属性，类似于进程的耳朵。故信号自然存储于task\_struct 中。

task\_struct{

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

long signal;//32 bit, 每一位代表一种信号，类似于中断请求寄存器

struct sigaction sigaction[32];//信号向量表，执行数组。类似于中断向量表

long blocked;//信号屏蔽位图, 类似于中断屏蔽寄存器。信号函数运行时需阻塞的信号。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

}

从上可知信号类似于软中断，有类似的中断号、中断向量表、中断处理函数、中断屏蔽位。

1. 主要两个方面， 定义信号处理函数以及 信号处理。

2.定义信号处理函数 有两个函数：

Signal(sihnal, handle)

Sigaction(sianal, handle, 0)

用户态这俩函数被调用时，会调用对应的包裹函数，最终进入内核态并调用 sys\_signal（，，sa\_restorer）sys\_sigaction（，，）。其中，sa\_restorer() 不要考虑，有人会帮你搞定。



**signal() sigaction() 的目的就是更新当前进程中对应信号向量表。--current->sigaction[signum-1]**

主要函数定义：

struct sigaction {

void (\*sa\_handler)(int); //信号处理函数指针，类似于中断处理函数

sigset\_t sa\_mask;//需要屏蔽的信号。sa\_flags |SA\_NOMASK 时 忽略屏蔽

int sa\_flags;//一些标志SA\_ONESHOT SA\_NOMASK

void (\*sa\_restorer)(void); //还原函数，可以忽略，libc负责这个参数的插入。

};

int sys\_signal(int signum, long handler, long restorer)

{

struct sigaction tmp;

if (signum<1 || signum>32 || signum==SIGKILL)

return -1;

tmp.sa\_handler = (void (\*)(int)) handler;

tmp.sa\_mask = 0;

tmp.sa\_flags = SA\_ONESHOT | SA\_NOMASK;

tmp.sa\_restorer = (void (\*)(void)) restorer;

handler = (long) current->sigaction[signum-1].sa\_handler;

current->sigaction[signum-1] = tmp;

return handler;

}

int sys\_sigaction(int signum, const struct sigaction \* action,

struct sigaction \* oldaction)

{

struct sigaction tmp;

if (signum<1 || signum>32 || signum==SIGKILL)

return -1;

tmp = current->sigaction[signum-1];

get\_new((char \*) action,

(char \*) (signum-1+current->sigaction));

if (oldaction)

save\_old((char \*) &tmp,(char \*) oldaction);

if (current->sigaction[signum-1].sa\_flags & SA\_NOMASK)

current->sigaction[signum-1].sa\_mask = 0;

else

current->sigaction[signum-1].sa\_mask |= (1<<(signum-1));

return 0;

}

1. 发现信号

内核态遍历当前进程的task\_struct 中 signal 不为0， current->signal .

开始着手信号的处理。

1. 信号如何处理

void do\_signal(long signr,long eax, long ebx, long ecx, long edx,

long fs, long es, long ds,

long eip, long cs, long eflags,

unsigned long \* esp, long ss)

{

unsigned long sa\_handler;

long old\_eip=eip;

struct sigaction \* sa = current->sigaction + signr - 1;

int longs;

unsigned long \* tmp\_esp;

sa\_handler = (unsigned long) sa->sa\_handler;

if (sa\_handler==1)

return;

if (!sa\_handler) {

if (signr==SIGCHLD)

return;

else

do\_exit(1<<(signr-1));

}

if (sa->sa\_flags & SA\_ONESHOT)

sa->sa\_handler = NULL;

\*(&eip) = sa\_handler;

longs = (sa->sa\_flags & SA\_NOMASK)?7:8;

\*(&esp) -= longs;

verify\_area(esp,longs\*4);

tmp\_esp=esp;

put\_fs\_long((long) sa->sa\_restorer,tmp\_esp++);

put\_fs\_long(signr,tmp\_esp++);

if (!(sa->sa\_flags & SA\_NOMASK))

put\_fs\_long(current->blocked,tmp\_esp++);

put\_fs\_long(eax,tmp\_esp++);

put\_fs\_long(ecx,tmp\_esp++);

put\_fs\_long(edx,tmp\_esp++);

put\_fs\_long(eflags,tmp\_esp++);

put\_fs\_long(old\_eip,tmp\_esp++);

current->blocked |= sa->sa\_mask;

}

简短点来讲就提供 把一些内核寄存器压入用户进程地址空间。从而改变进程的运行顺序。就像进程突然暂停下来去运行信号处理函数，运行后，然后做一些后续工作包括运行sa\_restorer()、把入栈的current->blocked 还原到当前的current->blocked，最后再继续执行之前暂停的工作。

1. 注意事项

1）.signal（） 这，对于sa\_mask 和 sa\_flags 是hard code .

　sigaction（）是可制定sa\_mask sa\_flags的。且有专门的C函数来更新sa\_mask sa\_flags

1. signal() 的 sa\_flags 含有SA\_ONESHOT：意味着信号处理函数只生效一次， 即在do\_signal() 中sa->sa\_handler被置空（为啥不直接置为SIG\_DFL， 虽然！NULL ==!SIG\_DFL），所以在恢复到用户态准备执行信号处理函数时，该信号的信号处理函数是 变为默认处理函数SIG\_DFL（除了SIGCHLD外，直接do\_exti()）。 所以有时候会在信号处理函数中再次调用signal() 来再次设置该信号的信号处理函数。
2. 从2）中可知Signal()的弱点在于：当下信号已经发生而进入信号处理函数中，在重新再次设置自己的处理函数之前（此时信号处理函数为SIG\_DFL 或者NULL），这个极限的时间段内，再次收到该信号时会发生什么情况。

首先，signal() 中该信号sigaction的sa\_flags 含有 SA\_NOMASK且sa\_mask = 0 .所以在do\_signal()中不会更新current->blocked且不会把current->blocked压栈。

一般情况下，调用signal()的结果是不会阻塞当前信号。当然也有例外。**这取决于current->blocked 中是否包含新信号了**。

不阻塞该信号（），而是会嵌套执行之前的流程。而此时该信号的处理函数已被设置为是默认值SIG\_DFL，而不是2）中预期的情况。

如果是收到的其他信号，且不阻塞，那就要看的不同信号处理函数之间的嵌套后的结果了。

1. sigaction 中优点在于可以自定义sa\_flags 和sa\_mask。具体工作内容：

1.把用户态的sigaction 中的sa\_flags 和sa\_mask 更新到 内核态中对应的sigaction.

2.action :更新内核信号sigaction 为action .

3.Oldaction 一般置空，啥也不干。不为空时，就是把 “更新内核信号sigaction 为action” 之前内核信号sigaction 更新到Oldaction .

4. sa\_flags 若包含SA\_NOMASK， 则直接置空sa\_mask。不会空则把当前信号更新到sa\_mask。 若不包含SA\_NOMASK，则在sa\_mask |= (1<<(signum-1))，即把当前信号也假如sa\_mask.

1. 就目前的代码来看，不管是signal() 或sigaction()，当sa\_flags 含有 SA\_NOMASK 时，sa\_mask最终会是0. 这种情况下，不压栈 current->blocked，也不会更新 current->blocked |= sa->sa\_mask。
2. 关于阻塞，就是等当前信号处理过程完全结束后，再去处理目前所捕获的信号，而不是嵌套处理。

在do\_signal() 中如果需要阻塞时，先压栈 current->blocked再更新 current->blocked |= sa->sa\_mask。注意两点，一是 ，此时sa->sa\_mask一般为为sigaction() 指定sa\_mask| 当前信号，和signal()关系不大；二是，此时入栈的current->blocked 和最新的current->blocked 可能已经不同了。我猜测的原因是 当开始执行在用户地址空间执行信号处理函数时收到新信号，根据 current->blocked判断新信号是否阻塞，收到非阻塞信号时 就嵌套之前收到信号后的流程。收到阻塞信号时，从而无法嵌套执行新信号的处理函数，等当之前信号处理函数结束后，恢复current->blocked 为之前入栈时的值，此时不在阻塞，并开始处理收到的新信号（也意味着，同一信号多次阻塞，和一次阻塞是一样的）。

在信号函数处理完成之后，因为伴随着信号处理函数的结束，阻塞值会还原到信号发生前的状态。 因为本来的目的就是希望运行信号处理函数期间阻塞某些信号，现在处理函数都运行完了，自然没有必要在阻塞了。

1. signal() sigaction() 中最后一个参数sa\_restore是一个函数指针，它在编译链接程序时由Libc 函数库提供。被用于信号处理函数结束后清理用户态堆栈，并恢复系统调用存放在eax中的返回值（我理解的是恢复之前来信号前的现场）。

备注：

以上主要是结合这三个内核函数（linux-0.11）所得到的一些体会。有些是自己的理解，可能和实际有出入。

但是目的是为了不用去死记signal() sigaction() 的用法。而是跳出来从内核方向去分析，从而去理解signal 机制。 肯定不够深刻。不够底层，但是完全可以通透地在apps 编程时 使用 signal .