一　函数说明

1. 函数使用说明

名字

ptrace – 进程跟踪

形式

#include <sys/ptrace.h>

int ptrace(int request, int pid, int addr, int data);

描述

Ptrace 提供了一种父进程可以控制子进程运行，并可以检查和改变它的核心 image。它主要用于实现断点调试。一个被跟踪的进程运行中，直到发生一个信号。则进程被中止，并且通知其父进程。在进程中止的状态下，进程的内存空间可以被读写。父进程还可以使子进程继续执行，并选择是否是否忽略引起中止的信号。

Request 参数决定了系统调用的功能：

PTRACE_TRACEME

本进程被其父进程所跟踪。其父进程应该希望跟踪子进程。

PTRACE_PEEKTEXT, PTRACE_PEEKDATA

从内存地址中读取一个字节，内存地址由 addr 给出。

PTRACE_PEEKUSR

从 USER 区域中读取一个字节，偏移量为 addr。

PTRACE_POKETEXT, PTRACE_POKEDATA

往内存地址中写入一个字节。内存地址由 addr 给出。

PTRACE_POKEUSR

往 USER 区域中写入一个字节。偏移量为 addr。

PTRACE_SYSCALL, PTRACE_CONT

重新运行。

PTRACE_KILL

杀掉子进程，使它退出。

PTRACE_SINGLESTEP

设置单步执行标志

PTRACE_ATTACH

跟踪指定 pid 进程。

PTRACE_DETACH

结束跟踪

Intel386 特有：

PTRACE_GETREGS

读取寄存器

PTRACE_SETREGS

设置寄存器
PTRACE_GETFPREGS
读取浮点寄存器
PTRACE_SETFPREGS
设置浮点寄存器
init 进程不可以使用此函数

返回值
成功返回 0。错误返回-1。errno 被设置。

错误
EPERM
特殊进程不可以被跟踪或进程已经被跟踪。
ESRCH
指定的进程不存在
EIO
请求非法

2. 功能详细描述

1) PTRACE_TRACEME

形式：ptrace(PTRACE_TRACEME,0 ,0 ,0)
描述：本进程被其父进程所跟踪。其父进程应该希望跟踪子进程。

2) PTRACE_PEEKTEXT, PTRACE_PEEKDATA

形式：ptrace(PTRACE_PEEKTEXT, pid, addr, data)
ptrace(PTRACE_PEEKDATA, pid, addr, data)
描述：从内存地址中读取一个字节，pid 表示被跟踪的子进程，内存地址由 addr 给出，data 为用户变量地址用于返回读到的数据。在 Linux（i386）中用户代码段与用户数据段重合所以读取代码段和数据段数据处理是一样的。

3) PTRACE_POKETEXT, PTRACE_POKEDATA

形式：ptrace(PTRACE_POKETEXT, pid, addr, data)
ptrace(PTRACE_POKEDATA, pid, addr, data)
描述：往内存地址中写入一个字节。pid 表示被跟踪的子进程，内存地址由 addr 给出，data

为所要写入的数据。

4) PTRACE_PEEKUSR

形式：ptrace(PTRACE_PEEKUSR, pid, addr, data)
描述：从 USER 区域中读取一个字节，pid 表示被跟踪的子进程，USER 区域地址由 addr 给出，data 为用户变量地址用于返回读到的数据。USER 结构为 core 文件的前面一部分，它描述了进程中止时的一些状态，如：寄存器值，代码、数据段大小，代码、数据段开始地址等。在 Linux（i386）中通过 PTRACE_PEEKUSER 和 PTRACE_POKEUSR 可以访问 USER 结构的数据有寄存器和调试寄存器。

5) PTRACE_POKEUSR

形式：ptrace(PTRACE_POKEUSR, pid, addr, data)
描述：往 USER 区域中写入一个字节，pid 表示被跟踪的子进程，USER 区域地址由 addr 给出，data 为需写入的数据。

6) PTRACE_CONT

形式：ptrace(PTRACE_CONT, pid, 0, signal)
描述：继续执行。pid 表示被跟踪的子进程，signal 为 0 则忽略引起调试进程中止的信号，若不为 0 则继续处理信号 signal。

7) PTRACE_SYSCALL

形式：ptrace(PTRACE_SYS, pid, 0, signal)
描述：继续执行。pid 表示被跟踪的子进程，signal 为 0 则忽略引起调试进程中止的信号，若不为 0 则继续处理信号 signal。与 PTRACE_CONT 不同的是进行系统调用跟踪。在被跟踪进程继续运行直到调用系统调用开始或结束时，被跟踪进程被中止，并通知父进程。

8) PTRACE_KILL

形式：ptrace(PTRACE_KILL,pid)
描述：杀掉子进程，使它退出。pid 表示被跟踪的子进程。

9) PTRACE_SINGLESTEP

形式：ptrace(PTRACE_KILL, pid, 0, signle)
描述：设置单步执行标志，单步执行一条指令。pid 表示被跟踪的子进程。signal 为 0 则忽略引起调试进程中止的信号，若不为 0 则继续处理信号 signal。当被跟踪进程单步执行完一个指令后，被跟踪进程被中止，并通知父进程。

10) PTRACE_ATTACH

形式：ptrace(PTRACE_ATTACH,pid)
描述：跟踪指定 pid 进程。pid 表示被跟踪进程。被跟踪进程将成为当前进程的子进程，并进入中止状态。

11) PTRACE_DETACH

形式：ptrace(PTRACE_DETACH,pid)
描述：结束跟踪。 pid 表示被跟踪的子进程。结束跟踪后被跟踪进程将继续执行。

12) PTRACE_GETREGS

形式：ptrace(PTRACE_GETREGS, pid, 0, data)
描述：读取寄存器值，pid 表示被跟踪的子进程，data 为用户变量地址用于返回读到的数据。此功能将读取所有 17 个基本寄存器的值。

13) PTRACE_SETREGS

形式：ptrace(PTRACE_SETREGS, pid, 0, data)
描述：设置寄存器值，pid 表示被跟踪的子进程，data 为用户数据地址。此功能将设置所有 17 个基本寄存器的值。

14) PTRACE_GETFPREGS

形式：ptrace(PTRACE_GETFPREGS, pid, 0, data)
描述：读取浮点寄存器值，pid 表示被跟踪的子进程，data 为用户变量地址用于返回读到的数据。此功能将读取所有浮点协处理器 387 的所有寄存器的值。

15) PTRACE_SETFPREGS

形式：ptrace(PTRACE_SETREGS, pid, 0, data)
描述：设置浮点寄存器值，pid 表示被跟踪的子进程，data 为用户数据地址。此功能将设置所有浮点协处理器 387 的所有寄存器的值。

二 80386 的调试设施

80386 提供的调试设施包括：

- 一字节的陷阱指令
- 单步指令
- 断点检测
- 任务切换时的自陷

1. 调试断点

断点设施是 80386 为调试程序提供的最重要的功能。

一个断点，允许编程人员对特定的线性地址设置特定的条件；当程序访问到该线性地址并满足特定的条件时，即跳转到异常处理程序。80386 可支持同时设置四个断点条件，编程人员可在程序中的四个位置设置条件，使其转向异常处理程序。这四个断点的每一个断点，都可以是如下三种不同类型的任何一种：

只在指令地址与断点地址一致时，断点有效。

数据写入地址与断点地址一致时，断点有效。

数据读出地址或数据写入地址与断点地址一致时，断点有效。

1) 调试寄存器

为支持提供四个调试断点，在 80386 中增加了八个寄存器，编号为 DR0 至 DR7。这八个寄存器中由四个用于断点，两个用于控制，另两个保留未用。对这八个寄存器的访问，只能在 0 级特权级进行。在其它任何特权级对这八个寄存器中的任意一个寄存器进行读或写访问，都将产生无效操作码异常。此外，这八个寄存器还可用 DR6 及 DR7 中的 BD 位和 GD 位进行进一步的保护，使其即使是在 0 级也不能进行读出或写入。

对这些寄存器的访问使用通常的 MOV 指令：

MOV reg Dri

该指令将调试寄存器 i 中的内容读至通用寄存器 reg 中；

MOV Dri reg

该指令将通用寄存器 reg 中的内容写至调试寄存器 i 中。此处 i 的取值可以为 0 至 7 中的任意值。

调试寄存器

图表示了这八个调试寄存器。这些寄存器的功能如下：

DR0—DR3 寄存器 DR0—DR3 包含有与四个断点条件的每一个相联系的线性地址（断点条件则在 DR7 中）。因为这里使用的是线性地址，所以，断点设施的操作，无论分页机制是否启用，都是相同的。

DR4—DR5 保留。

DR6 DR6 是调试状态寄存器。当一个调试异常产生时，处理器设置 DR6 的相应位，用于指示调试异常发生的原因，帮助调试异常处理程序分析、判断，以及作出相应处理。

DR7 DR7 是调试控制寄存器。分别对应四个断点寄存器的控制位，对断点的启用及断点类型的选择进行控制。所有断点寄存器的保护也在此寄存器中规定。

DR6 各位的功能

B0—B3 当断点线性地址寄存器规定的条件被检测到时,将对应的 B0—B3 位置 1。置位 B0—B3 与断点条件是否被启用无关。即 B0—B3 的某位被置 1，并不表示要进行对应的断点异常处理。

BD 如下一条指令要对八个调试寄存器之一进行读或写时，则在指令的边界 BD 位置 1。在一条指令内，每当即将读写调试寄存器时，也 BD 位置 1。BD 位置 1 与 DR7 中 GD 位启用与否无关。

BS 如果单步异常发生时，BS 位被置 1。单步条件由 EFLAGS 寄存器中的 TF 位启用。如果程序由于单步条件进入调试处理程序，则 BS 位被置 1。与 DR6 中的其它位不同的是，BS 位只在单步陷阱实际发生时才置位，而不是检测到单步条件就置位。

BT BT 位对任务切换导致 TSS 中的调试陷阱位被启用而造成的调试异常，指示其原因。对这一条件，在 DR7 中没有启用位。

DR6 中的各个标志位，在处理机的各种清除操作中不受影响，因此，调试异常处理程序在运行以前，应清除 DR6，以避免下一次检测到异常条件时，受到原来的 DR6 中状态位的影响。

DR7 各位的功能

LEN LEN 为一个两位的字段，用以指示断点的长度。每一断点寄存器对应一个这样的字段，所以共有四个这样的字段分别对应四个断点寄存器。LEN 的四种译码状态对应的断点长度如下

LEN	说明
0 0	断点为一字节
0 1	断点为两字节
1 0	保留
1 1	断点为四字节

这里，如果断点是多字节长度，则必须按对应多字节边界进行对齐。如果对应断点是一个指令地址，则 LEN 必须为 00

RWE RWE 也是两位的字段，用以指示引起断点异常的访问类型。共有四个 RWE 字段分别对应四个断点寄存器，RWE 的四种译码状态对应的访问类型如下

RWE	说明
0 0	指令
0 1	数据写
1 0	保留
1 1	数据读和写

GE/LE GE/LE 为分别指示准确的全局/局部数据断点。如果 GE 或 LE 被置位，则处理器将放慢执行速度，使得数据断点准确地把产生断点的指令报告出来。如果这些位没有置位，则处理器在执行数据写的指令接近执行结束稍前一点报告断点条件。建议读者每当启用数据断点时，启用 LE 或 GE。降低处理机执行速度除稍微降低一点性能以外，不会引起别的问题。但是，对速度要求严格的代码区域除外。这时，必须禁用 GE 及 LE，并且必须容许某些不太精确的调试异常报告。

L0—L3/G0—G3 L0—L3 及 G0—G3 位分别为四个断点寄存器的局部及全局启用信号。如果有任一个局部或全局启用位被置位，则由对应断点寄存器 DRi 规定的断点被启用。

GD GD 位启用调试寄存器保护条件。注意，处理程序在每次转入调试异常处理程序入口处清除 GD 位，从而使处理程序可以不受限制地访问调试寄存器。

前述的各个 L 位（即 LE，L0—L3）是有关任务的局部位，使调试条件只在特定的任务启用。而各个 G 位（即 GD，G0—G3）是全局的，调试条件对系统中的所有任务皆有效。在每次任务切换时，处理器都要清除 L 位。

2) 断点地址识别

LEN 字段及断点线性地址的组合,规定调试异常检查的四个线性地址的范围。上面已经提到，断点线性地址必须对齐于 LEN 规定的多字节长度的相应长度边界。事实上，处理器在检查断点时，根据 LEN 规定的长度，忽略线性地址的相应低位。例如，当 LEN=11 时，线性地址的最低两位被忽略，即把线性地址最低两位视为 00，因而按四字节边界对齐。而当 LEN=01 时，线性地址的最低位被忽略，即把线性地址的最低位视为 0，因而按两字节边界对齐。

对于由断点线性地址及 LEN 规定的地址范围内类型正确的任何字节的访问都产生异常，

数据的访问及指令的取出，都要按所有四个断点地址范围进行检查。如果断点地址范围的任何字节匹配，访问的类型也匹配，则断点异常被报告。

下表给出了识别数据断点的几个离子，这里假设所有断点被启用，而且设置了正确的访问类型。

		地址	长度	长度	断点
寄存器内容	DR0 DR1 DR2 DR3	00FF02 00CC32 0D0004 01FF00	1 2 4 4	Len=00 Len=01 Len=11 Len=11	
引起异常的访问		00FF02 00CC33 0D0007 00FEFF 01FF00 01FF03	1 1 2 4 4 4		B0=1 B1=1 B2=1 B0=1 B3=1 B3=1
不引起异常的访问		00FF01 00FF00 00CC34 01FEFF 0D0000	1 2 1 1 4		

3) 代码断点与数据断点的比较

指令访问断点与数据访问断点之间有如下几点区别：

1．在 RWE 字段的设置不同。指令断点，RWE=0；数据断点，RWE≠0。

2．LEN 的设置不同。指令断点的长度只能是 00 即一字节;数据断点的长度可以是 1、2、4 字节。由于很多指令的长度超过一字节（事实上，指令长度为 1—15 字节），所以指令断点必须设置在指令的第一个字节。

3．指令断点属故障类型，数据断点属陷阱类型。指令断点故障在指令执行前被检查及报告。数据断点陷阱则在对断点位置进行读和写之后才被报告。可以看出，数据断点陷阱对断点数据未进行写保护。要保护断点的数据不会被写入操作破坏，必须先在调试处理程序中保存断点原来数据的拷贝。

由于指令断点在指令执行之前被报告，因此，很明显，对该指令不能简单的重新执行。因为每一次新的执行都简单地重复产生故障，所以，如果调试处理程序不禁用断点，则这种故障就会形成无限地循环。为解决这一问题，就需用到 80386 中 EFLAGS 的 RF 位。当 RF 位置位时，任何指令断点都被忽略，因此，在 RF 位保持为置位状态时，指令断点将不再起作用。但 RF 位的置位状态不会长久保持。事实上，处理器的内部逻辑保证，在任何一条指令成功完成后，都将 RF 位清零，因此，RF 位的置位状态最多只保持一条指令的时间。也就是说，在 RF 位置位后的下一条指令，指令断点不起作用，这样只要在重新执行指令之前，将 RF 置 1，即可保证该指令断点不会形成无限循环，而且，也不影响紧接的下一条指令也设置指令断点。

RF 位的置位，不是用某一个操作直接将 EFLAGS 的 RF 位置 1 来完成。每当进入一个故障处理程序，处理器保存中断现场时，需把断点等信息压栈。当把 EFLAGS 寄存器压栈时，推入栈中的EFLAGS的RF位是1,因此用IRET指令推出故障处理程序时,从栈中弹出的EFLAGS寄存器标志位中的 RF 为 1，从而将 RF 位置位。

2. TSS 中的调度陷阱

每当通过 TSS 发生任务切换时，TSS 中的 T 位使调试处理程序被调用，这就为调试程序管理某些任务的活动提供了一种方便的方法。DR6 中的 BT 位指示对该位的检测，DR7 中对该位没有特别的启用位。

如果调试处理程序是通过任务门使用的，则不能设置对应 TSS 的调试陷阱位。否则，将发生调试处理程序的无限循环。

3. INT3

一个断点指令提供调试程序的另一种方法。按这种方法，要求作为断点指令的第一个字节用 INT3 指令替代。因此，程序执行到预先需要的断点处遇到断点指令，并进入 INT3 处理程序。在一些使用 INT3 显然不足的地方还需使用断点寄存器。这样的情况有：

1．由 ROM 提供的代码中，不可能插入 INT3 指令。
2．由于使用了 INT3，原来的程序代码被修改，使执行此代码的其它任务也被中断。
3．INT3 不能执行数据断点。

在另外一些情况下，使用 INT3 则很有用：

1．单步及断点设施仅仅进入调试程序，而对调试处理程序的调试，INT3 则是唯一方便的方法。
2．代码中可以插入任意数量的 INT3 指令，而断点设施只能提供最多四个断点。
3．早期 86 系列的各种型号处理器，没有 80386 提供的断点设施，INT3 指令在这些处理器中是执行任何断点的唯一方法。

概括地说，除了某些特别情况之外，建议使用 INT3 指令在代码中执行断点，保留断点寄存器用于数据断点。

4. 程序的步进执行

单步功能对程序调试者来说，是一个方便的调试手段。通过一条一条地执行指令，对操作数据、操作指令及操作结果地观察和分析，可以帮助调试人员判断出执行某一指令时，是否发生了硬件错误，或是否软件逻辑错误。80386 的单步功能通过陷阱来实现。单步陷阱在 EFLAGS 寄存器中的 TF 位置位时启用。在一条指令开始执行时，如果有 TF=1，则在指令执行的末尾产生调试异常，并进入调试处理程序。在这里，“指令开始执行时，TF=1”这一条件是重要的。有此条件的限制，使 TF 位置位 1 的指令不会产生单步陷阱。每次产生单步陷阱之后，在进入调试处理程序之前要将 TF 位清除。此外，在处理中断或异常时，也清除 TF

位。

如果外部中断与单步中断同时发生，则单步中断被优先处理，并清除 TF。在调试处理程序第一条指令执行之前，如仍有悬挂的中断请求，则响应并处理中断。因此，中断处理是在没有单步启用的情况下完成的。如果希望在中断处理程序中使用单步功能。则需先把中断处理程序的第一条指令设置为断点，当程序运行到断点处停下来之后，再启用单步功能。

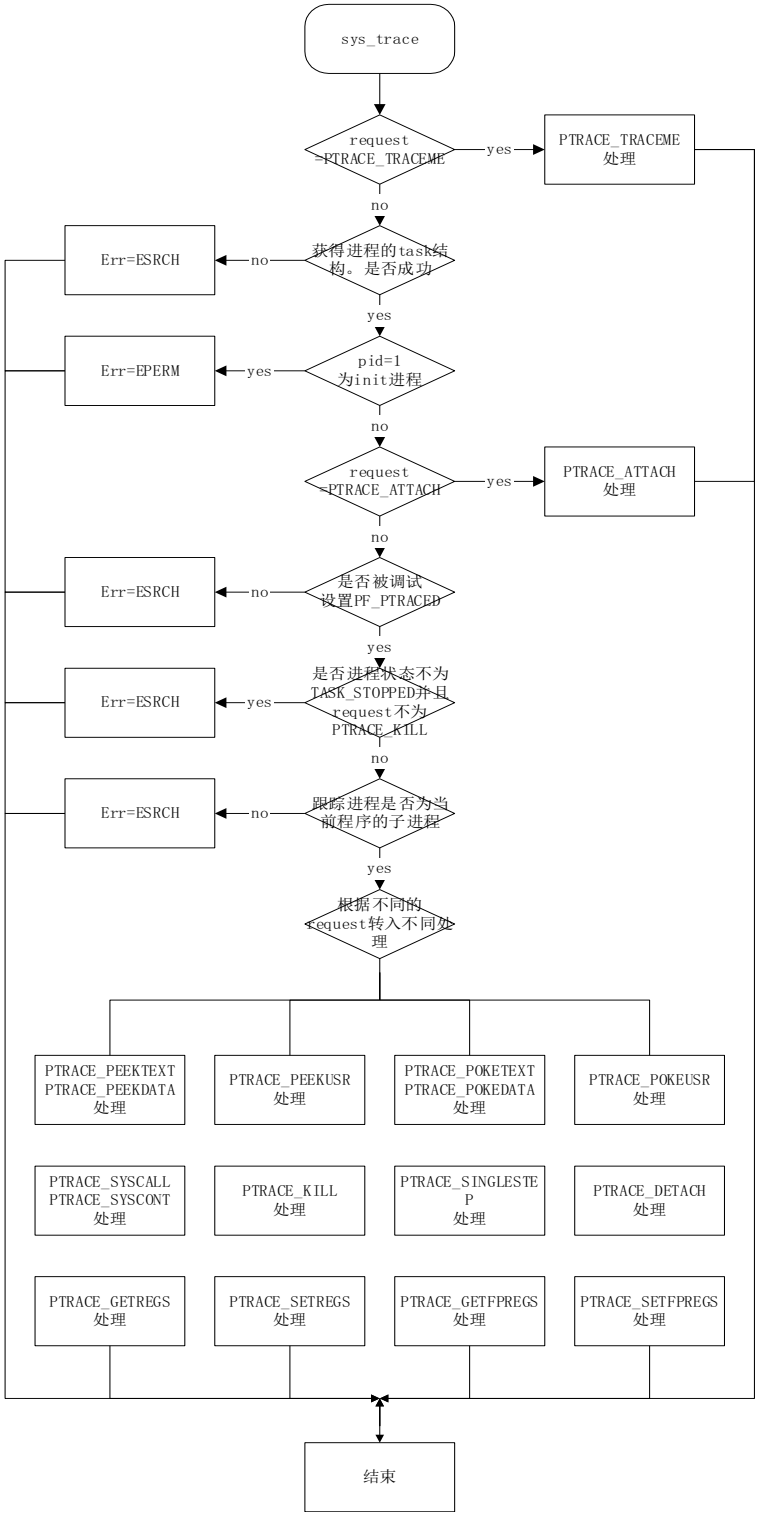
三　代码分析

在 Linux 核心源代码中，与完成 ptrace 功能相关的代码有：

- sys_ptrace 函数，完成 ptrace 系统调用的代码。
- 为完成 sys_ptrace 功能所需调用的一些辅助函数，寄存器读写函数和内存读写函数。
- 信号处理函数中，对被调试进程的处理（中止其运行、继续运行）。
- syscall_trace 函数，完成了系统调用调试下的处理。
- 调试陷阱处理（异常 1 处理），完成单步执行和断点中断处理。
- execve 系统调用中对被调试进程装入后中止的实现。

1. sys_ptrace 函数

ptrace 系统调用在核心对应的处理函数为 sys_ptrace()(/linux/arch/i386/kernel/ptrace.c)。sys_ptrace 函数完成了 ptrace 系统调用功能。ptrace 函数的总体流程如下：



程序和说明注释如下：

```
asmlinkage int sys_ptrace(long request, long pid, long addr, long data)
{
        struct task_struct *child;
        struct user * dummy = NULL;
        unsigned long flags;
        int i, ret;
```

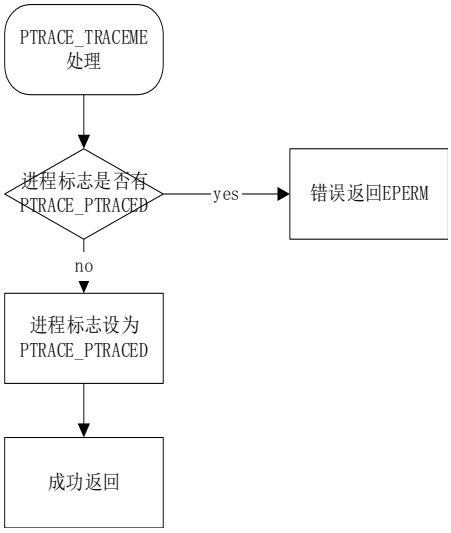
```
lock_kernel();
ret = -EPERM;
if (request == PTRACE_TRACEME) {
        。。。PTRACE_TRACEME 处理
}
ret = -ESRCH;
read_lock(&tasklist_lock);
child = find_task_by_pid(pid);          /*  查找 task 结构   */
read_unlock(&tasklist_lock);
if (!child)                             /*  没有找到 task 结构，所名给定 pid 错误   */
    goto out;
ret = -EPERM;
if (pid == 1)                           /*  init 进程不能调试   */
    goto out;
if (request == PTRACE_ATTACH) {
        。。。PTRACE_ATTACH 处理
}
ret = -ESRCH;
if (!(child->flags & PF_PTRACED))   /*  进程没有被跟踪，不能执行其它功能   */
    goto out;
if (child->state != TASK_STOPPED) {
    if (request != PTRACE_KILL)     /*  除 PTRACE_KILL 外的其它功能要求   */
        goto out;                   /*  要求进程状态为 TASK_STOPPED       */
}
if (child->p_pptr != current)           /*  被跟踪进程要求为当前进程的子进程   */
    goto out;

switch (request) {
    case PTRACE_PEEKTEXT:
    case PTRACE_PEEKDATA: {
        。。。PTRACE_PEEKTEXT，PTRACE_PEEKDATA 处理
    }
    case PTRACE_PEEKUSR: {
        。。。PTRACE_PEEKUSR 处理
    }
    case PTRACE_POKETEXT:
    case PTRACE_POKEDATA:{
        。。。PTRACE_POKETEXT，PTRACE_POKEDATA 处理
    }
    case PTRACE_POKEUSR: {
        。。。PTRACE_POKEUSR 处理
    }
    case PTRACE_SYSCALL:
```

```
            case PTRACE_CONT:
                。。。PTRACE_SYSCALL，PTRACE_CONT 处理
            }
            case PTRACE_KILL: {
                。。。PTRACE_KILL 处理
            }
            case PTRACE_SINGLESTEP: {
                。。。PTRACE_SINGLESTEP 处理
            }
            case PTRACE_DETACH:
                。。。PTRACE_DETACH 处理
            }
            case PTRACE_GETREGS:
                。。。PTRACE_GETREGS 处理
              };
            case PTRACE_SETREGS:
                。。。PTRACE_SETREGS 处理
              };
            case PTRACE_GETFPREGS:
                。。。PTRACE_GETFPREGS 处理
              };
            case PTRACE_SETFPREGS:
                。。。PTRACE_SETFPREGS 处理
              };
            default:
                ret = -EIO;
                goto out;
        }
out:
        unlock_kernel();
        return ret;
}
```

1) PTRACE_TRACEME 处理

说明：此处理使当前进程进入调试状态。进程是否为调试状态由进程的标志 PF_PTRACED 表示。

流程：



程序：

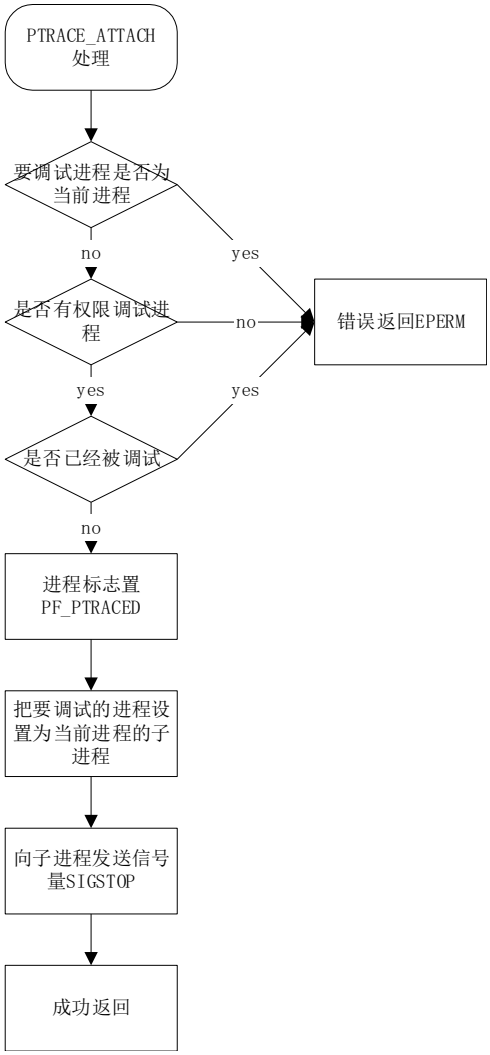
```
if (request == PTRACE_TRACEME) {
    if (current->flags & PF_PTRACED)        /*   是否已经被跟踪   */
        goto out;
    current->flags |= PF_PTRACED;           /*   设置跟踪标志   */
    ret = 0;
    goto out;
}
```

2) PTRACE_ATTACH 处理

说明：此处理设置开始调试某一进程，此进程可以是任何进程（init 进程除外）。对某一进程的调试需有对这一进程操作的权限。不能调试自身进程。一个进程不能 ATTACH 多次。

为完成对一个进程的调试设置，首先设置进程标志置 PF_PTRACED。再将需调试的进程设置为当前进程的子进程。最后向它发信号 SIGSTOP 中止它的运行，使它进入调试状态。

流程：



程序：

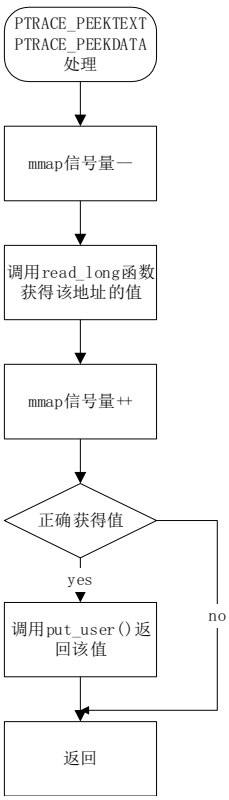
```
if (request == PTRACE_ATTACH) {
    if (child == current)                   /*   不能调试自身进程   */
        goto out;
    if ((!child->dumpable ||
        (current->uid != child->euid) ||
        (current->uid != child->suid) ||
        (current->uid != child->uid) ||
        (current->gid != child->egid) ||
        (current->gid != child->sgid) ||
        (!cap_issubset(child->cap_permitted, current->cap_permitted)) ||
        (current->gid != child->gid)) && !capable(CAP_SYS_PTRACE))
        goto out;                           /*   检验用户权限   */
    if (child->flags & PF_PTRACED) /*   一个进程不能被 attach 多次   */
        goto out;
    child->flags |= PF_PTRACED;     /*   设置进程标志位 PF_PTRACED   */

    write_lock_irqsave(&tasklist_lock, flags);
```

```
        if (child->p_pptr != current) {       /*   设置进程为当前进程的子进程   */
            REMOVE_LINKS(child);
            child->p_pptr = current;
            SET_LINKS(child);
        }
        write_unlock_irqrestore(&tasklist_lock, flags);
        send_sig(SIGSTOP, child, 1);       /*   发送 SIGSTOP 信号，中止它运行   */
        ret = 0;
        goto out;
    }
```

3) PTRACE_PEEKTEXT，PTRACE_PEEKDATA 处理

说明：在 Linux（i386）中，用户代码段和用户数据段是重合的所以 PTRACE_PEEKTEXT，PTRACE_PEEKDATA 的处理是相同的。在其它 CPU 或操作系统上有可能是分开的，那要分开处理。读写用户段数据通过 read_long（）和 write_long（）两个辅助函数完成，具体函数过程参见两函数分析。

流程：



程序：

```
        case PTRACE_PEEKTEXT:
        case PTRACE_PEEKDATA: {
            unsigned long tmp;
            down(&child->mm->mmap_sem);
            ret = read_long(child, addr, &tmp);            /*   读取数据   */
            up(&child->mm->mmap_sem);
```

```
        if (ret >= 0)
            ret = put_user(tmp,(unsigned long *) data);   /*   返回结果   */
        goto out;
    }
```

4) PTRACE_POKETEXT，PTRACE_POKEDATA 处理

说明：与 PTRACE_PEEKTEXT，PTRACE_PEEKDATA 处理相反，此处理为写进程内存（详见上）

流程：

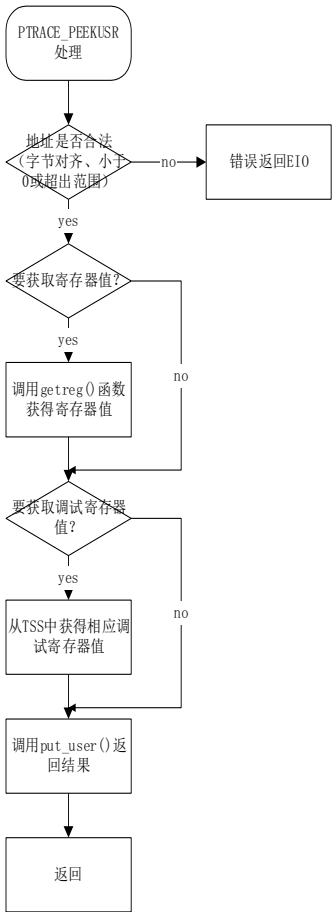
程序：

```
    case PTRACE_POKETEXT:
    case PTRACE_POKEDATA:
        down(&child->mm->mmap_sem);
        ret = write_long(child,addr,data);        /*   修改数据   */
        up(&child->mm->mmap_sem);
        goto out;
```

5) PTRACE_PEEKUSR 处理

说明：在 Linux（i386）中，读写 USER 区域的数据值有用户寄存器和调试寄存器的值。用户寄存器包括 17 个寄存器，它们分别是 EBX、ECX、EDX、ESI、EDI、EBP、EAX、DS、ES、FS、GS、ORIG_EAX、EIP、CS、EFLAGS、ESP、SS。这些寄存器的读写由辅助函数 putreg（）和 getreg（）函数完成，具体实现参见两函数分析。调试寄存器为 DR0—DR7。其中 DR4 和 DR5 为系统保留的寄存器，不可以写。DR0—DR3 中的断点地址必须在用户的 3G 空间内，在核心内存设置断点非法。DR7 中的 RWE 与 LEN 数据位必须合法（LEN≠10 保留、RWE≠10 保留、RWE=00 时 LEN=00 指令断点为一字节）。

流程：



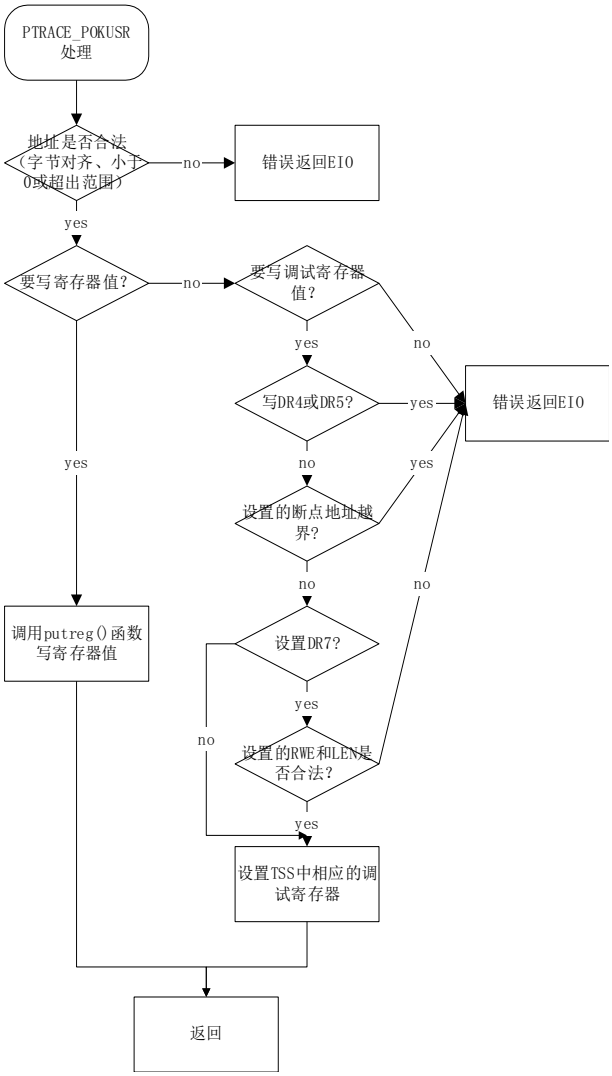
程序：

```
case PTRACE_PEEKUSR: {
        unsigned long tmp;
        ret = -EIO;
        if ((addr & 3) || addr < 0 ||                   /*   越界或字节未对齐出错   */
            addr > sizeof(struct user) - 3)
            goto out;
        tmp = 0;    /* Default return condition */
        if(addr < 17*sizeof(long))                      /*   读取基本寄存器值   */
            tmp = getreg(child, addr);
        if(addr >= (long) &dummy->u_debugreg[0] &&
           addr <= (long) &dummy->u_debugreg[7]){
            addr -= (long) &dummy->u_debugreg[0];
            addr = addr >> 2;
            tmp = child->tss.debugreg[addr];            /*   读取调试寄存器值   */
        };
        ret = put_user(tmp,(unsigned long *) data);  /*   返回结果   */
        goto out;
    }
```

6) PTRACE_POKEUSR 处理

说明：与 PTRACE_PEEKUSR 处理相反，此处理为写 USER 区域（详见上）。

流程：



程序：

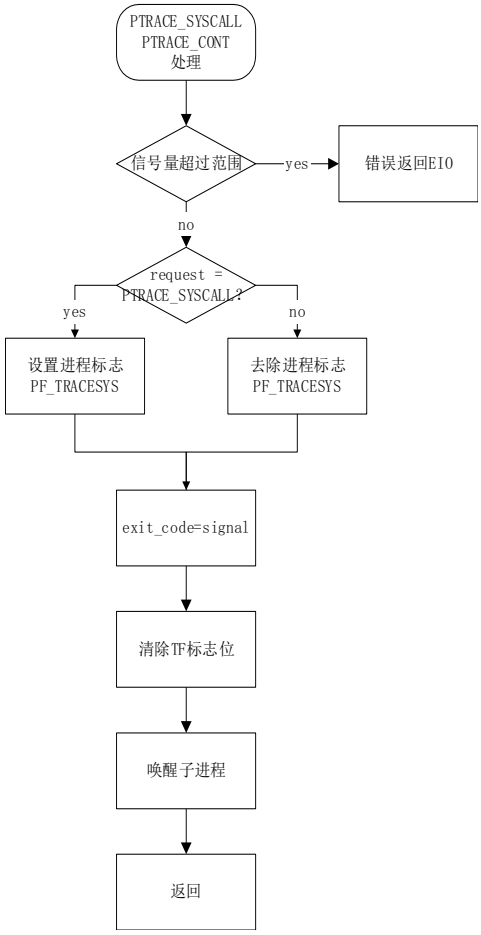
```
        case PTRACE_POKEUSR:
            ret = -EIO;
            if ((addr & 3) || addr < 0 ||                   /*   越界或字节未对齐出错   */
                addr > sizeof(struct user) - 3)
                goto out;
            if (addr < 17*sizeof(long)) {
                ret = putreg(child, addr, data);            /*   写基本寄存器值   */
                goto out;
            }
          if(addr >= (long) &dummy->u_debugreg[0] &&
             addr <= (long) &dummy->u_debugreg[7]){
               if(addr == (long) &dummy->u_debugreg[4]) return -EIO;        /*   写 DR4 出错
*/
```

```
        if(addr == (long) &dummy->u_debugreg[5]) return -EIO; /*   写 DR5 出错   */
        if(addr < (long) &dummy->u_debugreg[4] &&
            ((unsigned long) data) >= TASK_SIZE-3) return -EIO;
                                        /*   断点地址越界出错   */
        ret = -EIO;
        if(addr == (long) &dummy->u_debugreg[7]) { /*   写 DR7   */
              data &= ~DR_CONTROL_RESERVED;
              for(i=0; i<4; i++)
                    if ((0x5f54 >> ((data >> (16 + 4*i)) & 0xf)) & 1)
                          goto out;     /*   LEN RWE 非法出错   */
        };
        addr -= (long) &dummy->u_debugreg;
        addr = addr >> 2;
        child->tss.debugreg[addr] = data;      /*   写调试寄存器值   */
        ret = 0;
        goto out;
  };
  ret = -EIO;
  goto out;
```

7) PTRACE_SYSCALL，PTRACE_CONT 处理

说明：PTRACE_SYSCALL 和 PTRACE_CONT 有着相同的处理，都是让子进程继续运行，其区别 PTRACE_SYSCALL 设置了进程标志 PF_TRACESYS。这样可以使进程在下一次系统调用开始或结束时中止运行。继续执行要保证清除单步执行标志。用户参数 data 为用户提供的信号，希望子进程继续处理此信号。如果为 0 则不处理，如果不为 0 则在唤醒子进程后向子进程发送此信号（在 do_signal()和 syscall_trace()函数中完成）。

流程：



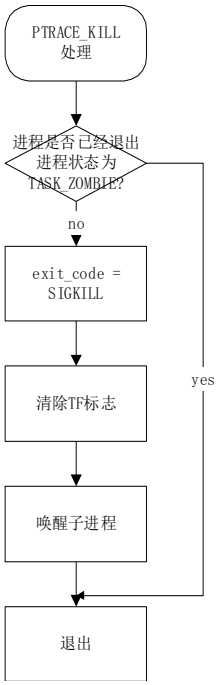
程序：

```
case PTRACE_SYSCALL:
case PTRACE_CONT:
    long tmp;
    ret = -EIO;
    if ((unsigned long) data > _NSIG)        /*   信号超过范围   */
        goto out;
    if (request == PTRACE_SYSCALL)
        child->flags |= PF_TRACESYS;   /*   设置 PF_TRACESYS 标志   */
    else
        child->flags &= ~PF_TRACESYS; /*   去除 PF_TRACESYS 标志   */
    child->exit_code = data;                    /*   设置继续处理的信号   */
    tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
    put_stack_long(child, EFL_OFFSET,tmp);   /*   清除 TF 标志   */
    wake_up_process(child);                          /*   唤醒子进程   */
    ret = 0;
    goto out;
}
```

8) PTRACE_KILL 处理

说明：此功能完成杀死子进程的功能。以往杀死进程只要往此进程发送 SIGKILL 信号。在此处理类似于 PTRACE_CONT 处理，只是把子进程继续的信号设置为 SIGKILL，则唤醒子进程后，子进程会受到 SIGKILL 信号。

流程：



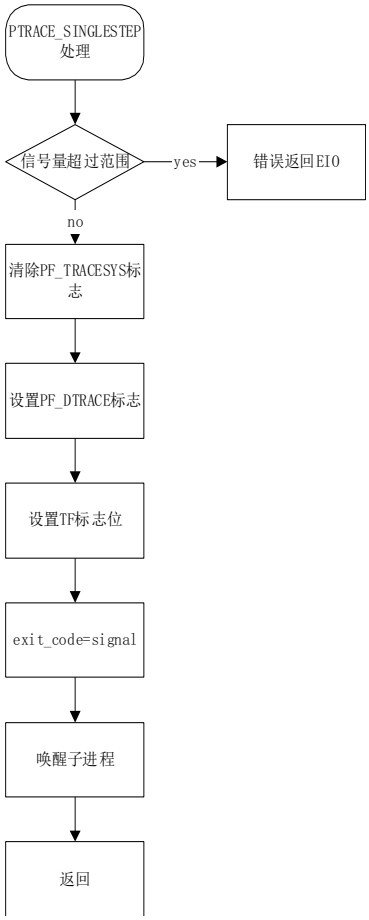
程序：

```
case PTRACE_KILL: {
    long tmp;
    ret = 0;
    if (child->state == TASK_ZOMBIE)    /*    进程已经退出    */
        goto out;
    child->exit_code = SIGKILL;             /*    设置继续处理的信号 SIGKILL    */
    tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
    put_stack_long(child, EFL_OFFSET, tmp);  /*    清除 TF 标志    */
    wake_up_process(child);                                 /*    唤醒子进程    */
    goto out;
}
```

9) PTRACE_SINGLESTEP 处理

说明：单步调试，子进程运行一条指令。此处理类似于 PTRACE_CONT 处理。不同的只是设置类单步调试标志 TF。

流程：



程序：

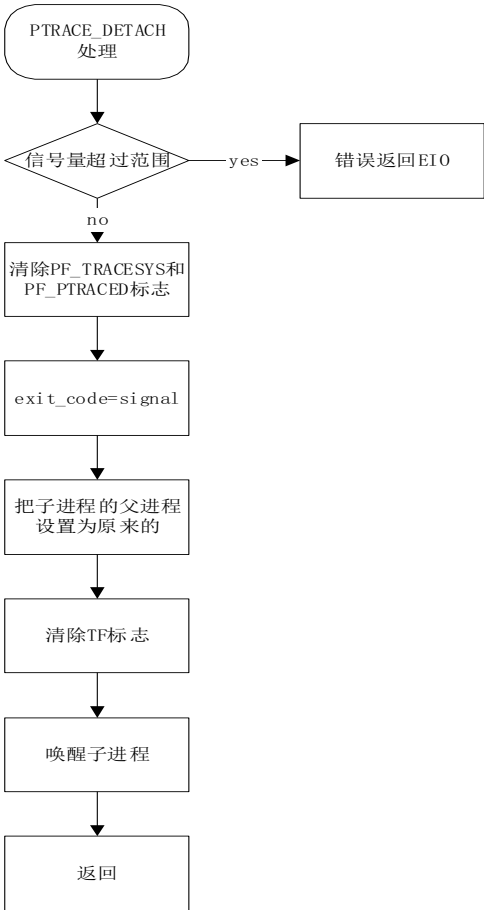
```
case PTRACE_SINGLESTEP: {
    long tmp;
    ret = -EIO;
    if ((unsigned long) data > _NSIG)        /*   信号超过范围   */
        goto out;
    child->flags &= ~PF_TRACESYS;      /*   清除 PF_TRACESYS 标志   */
    if ((child->flags & PF_DTRACE) == 0) {
        child->flags |= PF_DTRACE;      /*    设置 PF_DTRACE 标志   */
    }
    tmp = get_stack_long(child, EFL_OFFSET) | TRAP_FLAG;
    put_stack_long(child, EFL_OFFSET, tmp);  /*   设置 TF 标志   */
    child->exit_code = data;                 /*   设置继续处理的信号   */
    wake_up_process(child);                      /*   唤醒子进程   */
    ret = 0;
    goto out;
}
```

10)　PTRACE_DETACH 处理

说明：终止调试一个子进程。此处理与 PTRACE_ATTACH 处理相反。在此做了一些清理操作：清除 PF_TRACESYS 和 PF_PTRACED 进程标志，清除 TF 标志，父进程指针还原。最

后唤醒此进程，让其继续执行。

流程：



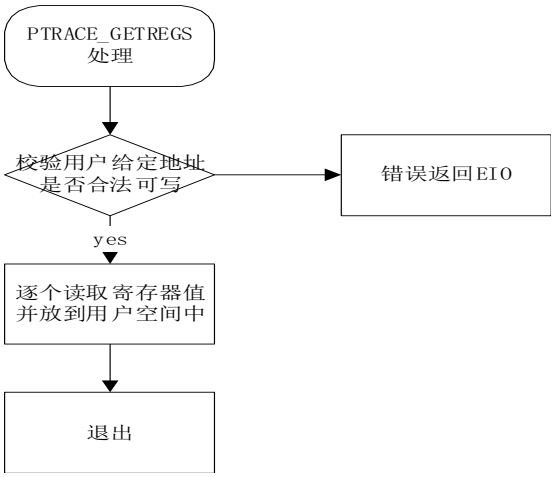
程序：

```
case PTRACE_DETACH: {
    long tmp;
    ret = -EIO;
    if ((unsigned long) data > _NSIG)        /*  信号超过范围  */
        goto out;
    child->flags &= ~(PF_PTRACED|PF_TRACESYS);
                        /*  清除 PF_TRACESYS 和 PF_PTRACED 标志  */
    child->exit_code = data;                 /*  设置继续处理的信号  */
    write_lock_irqsave(&tasklist_lock, flags);
    REMOVE_LINKS(child);
    child->p_pptr = child->p_opptr;   /*  把子进程的父进程设置为原来的  */
    SET_LINKS(child);
    write_unlock_irqrestore(&tasklist_lock, flags);
    tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
    put_stack_long(child, EFL_OFFSET, tmp);  /*  清除 TF 标志  */
    wake_up_process(child);                  /*  唤醒子进程  */
    ret = 0;
    goto out;
}
```

11) PTRACE_GETREGS 处理

说明：此功能完成读取所有的 17 个用户寄存器。读寄存器值使用函数 getreg（），详见 getreg（）分析。此功能为 i386 特有。

流程：



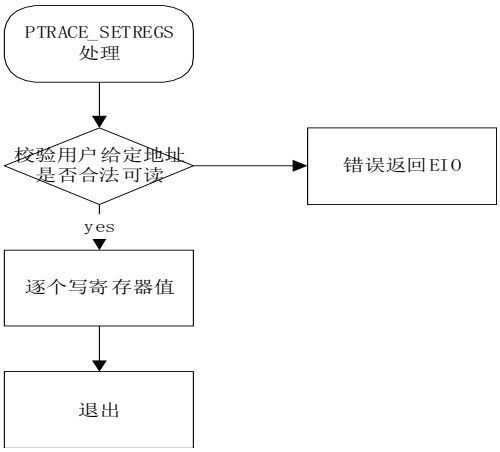
程序：

```
case PTRACE_GETREGS: {
      if (!access_ok(VERIFY_WRITE, (unsigned *)data,
                     17*sizeof(long)))     /*   校验用户给定地址是否合法可写   */
        {
          ret = -EIO;
          goto out;
        }
      for ( i = 0; i < 17*sizeof(long); i += sizeof(long) )
        {                                  /*   逐个读取寄存器值并放到用户空间中   */
          __put_user(getreg(child, i),(unsigned long *) data);
          data += sizeof(long);
        }
      ret = 0;
      goto out;
    };
```

12) PTRACE_SETREGS 处理

说明：此功能完成设置所有的 17 个用户寄存器。写寄存器值使用函数 putreg（），详见 putreg（）分析。此功能为 i386 特有。

流程：



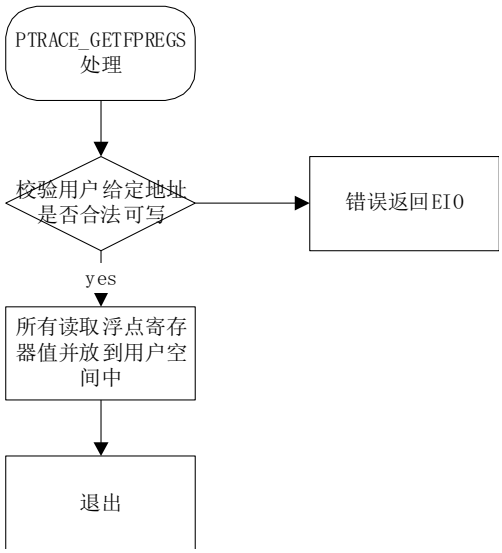
程序：

```
case PTRACE_SETREGS: {
    unsigned long tmp;
    if (!access_ok(VERIFY_READ, (unsigned *)data,
                    17*sizeof(long)))     /*   校验用户给定地址是否合法可读   */
      {
        ret = -EIO;
        goto out;
      }
    for ( i = 0; i < 17*sizeof(long); i += sizeof(long) )
      {                    /*   逐个写寄存器值   */
        __get_user(tmp, (unsigned long *) data);
        putreg(child, i, tmp);
        data += sizeof(long);
      }
    ret = 0;
    goto out;
  };
```

13)　PTRACE_GETFPREGS 处理

说明：此功能完成读取所有浮点寄存器。所有浮点寄存器存放于 TSS 中，由 TSS 中的 i386 联合表示浮点寄存器。如果有浮点处理器，则存放硬件的寄存器内容。否则，则存放软件模拟的寄存器内容。此功能为 i386 特有。

流程：



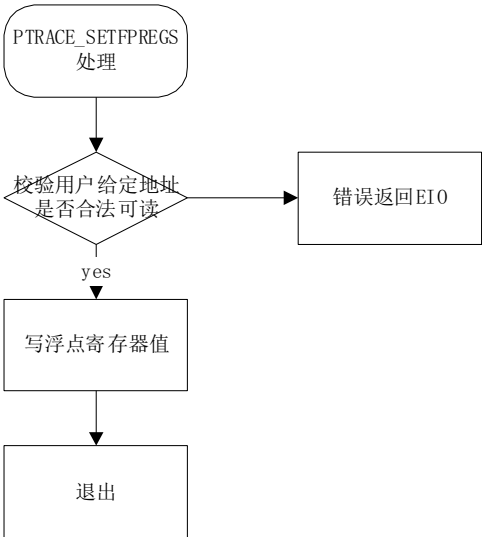
程序：

```
        case PTRACE_GETFPREGS: {
            if (!access_ok(VERIFY_WRITE, (unsigned *)data,
                        sizeof(struct user_i387_struct)))
              {                         /*   校验用户给定地址是否合法可写   */
                ret = -EIO;
                goto out;
              }
            ret = 0;
            if ( !child->used_math ) { /*   模拟一个空的浮点处理器   */
              /* Simulate an empty FPU. */
              child->tss.i387.hard.cwd = 0xffff037f;
              child->tss.i387.hard.swd = 0xffff0000;
              child->tss.i387.hard.twd = 0xffffffff;
            }
#ifdef CONFIG_MATH_EMULATION
            if ( boot_cpu_data.hard_math ) {
#endif
                __copy_to_user((void *)data, &child->tss.i387.hard,
                            sizeof(struct user_i387_struct));
                                    /*   复制浮点寄存器值（硬件）   */
#ifdef CONFIG_MATH_EMULATION
            } else {
              save_i387_soft(&child->tss.i387.soft,
                        (struct _fpstate *)data);
                                    /*   复制浮点寄存器值（软件模拟）   */
            }
#endif
            goto out;
          };
```

14) PTRACE_SETFPREGS 处理

说明：此功能完成设置所有浮点寄存器。此功能为 i386 特有。

流程：



程序：

```
        case PTRACE_SETFPREGS:
            if (!access_ok(VERIFY_READ, (unsigned *)data,
                        sizeof(struct user_i387_struct)))
              {                         /*  校验用户给定地址是否合法可读  */
                ret = -EIO;
                goto out;
              }
            child->used_math = 1;   /*  设置标志使用浮点处理器  */
#ifdef CONFIG_MATH_EMULATION
            if ( boot_cpu_data.hard_math ) {
#endif
              __copy_from_user(&child->tss.i387.hard, (void *)data,
                        sizeof(struct user_i387_struct));
                                    /*  设置浮点寄存器值（硬件）  */
#ifdef CONFIG_MATH_EMULATION
            } else {
              restore_i387_soft(&child->tss.i387.soft,
                        (struct _fpstate *)data);
                                    /*  设置浮点寄存器值（软件模拟）  */
            }
#endif
            ret = 0;
            goto out;
          };
```

2. 寄存器读写辅助函数

getreg() putreg()是在 ptrace.c 中定义的两个辅助函数，它们完成了对被调试子进程的寄存器读写功能。函数中参数 regno，表示寄存器的序号。定义如下：

Regno/4	寄存器
0	EBX
1	ECX
2	EDX
3	ESI
4	EDI
5	EBP
6	EAX
7	DS
8	ES
9	FS
10	GS
11	ORIG_EAX
12	EIP
13	CS
14	EFL
15	USEP
16	SS

进程结构中 TSS 存有所有的进程寄存器值，但不能使用这些寄存器的值。因为在调试器调用 ptrace()读写寄存器时，被调试进程必须在中止状态，引起被调试进程中止有两种可能：1.接受到信号，do_signal 处理中。2.系统调用调试中断，syscall_trace 处理中。在这时被调试进程在核心态运行，那 TSS 中的寄存器为核心态运行时的寄存器状态，而通过 ptrace()读写的寄存器为用户态的寄存器状态。所以，getreg()和 putreg()不从 TSS 结构中读写寄存器值，而要通过操作核心态的堆栈（堆栈中保存有用户态的寄存器值）来读写寄存器值。

当进程系统调用或时钟中断处理时，系统会把所有的用户态的寄存器压入堆栈保存，而处理完毕之后恢复寄存器的值，这些寄存器值在堆栈中的顺序如下（与 regno 比较）：

堆栈		regno	
偏移(ESP+n)	寄存器	regno	寄存器
0	EBX	0	EBX
4	ECX	4	ECX
8	EDX	8	EDX
C	ESI	C	ESI
10	EDI	10	EDI
14	EBP	14	EBP
18	EAX	18	EAX
1C	DS	1C	DS
20	ES	20	ES

24	ORIG_EAX	24	FS
28	EIP	28	GS
2C	CS	2C	ORIG_EAX
30	EFL	30	EIP
34	USEP	34	CS
38	SS	38	EFL
		3C	USEP
		40	SS

其中不包含寄存器 fs 和 gs。对这两个寄存器的操作通过访问 TSS 结构实现。

对写寄存器，有以下的限制：

1．对 ORIG_EAX 寄存器不能写。

2．对段寄存器（CS、DS、ES、FS、GS、SS）的修改，其中 RPL 必须为 11（优先极为 3）。

3．对标志寄存器 EFLAG 中标志 IF、RF、VM、IOPL 不能修改。

函数 get_stack_long()和 put_stack_long()为对子进程核心堆栈的操作。

源程序与注释如下：

```
static inline int get_stack_long(struct task_struct *task, int offset)
{
    unsigned char *stack;
    stack = (unsigned char *)task->tss.esp0;          /*  获得 ESP0 寄存器值  */
    stack += offset;                                  /*  加偏移量  */
    return (*((int *)stack));
}

static inline int put_stack_long(struct task_struct *task, int offset,
    unsigned long data)
{
    unsigned char * stack;
    stack = (unsigned char *) task->tss.esp0;     /*  获得 ESP0 寄存器值  */
    stack += offset;                              /*  加偏移量  */
    *(unsigned long *) stack = data;
    return 0;
}

static int putreg(struct task_struct *child,
    unsigned long regno, unsigned long value)
{
    switch (regno >> 2) {
        case ORIG_EAX:              /*  不能读写 EAX  */
            return -EIO;
        case FS:
            if (value && (value & 3) != 3)
```

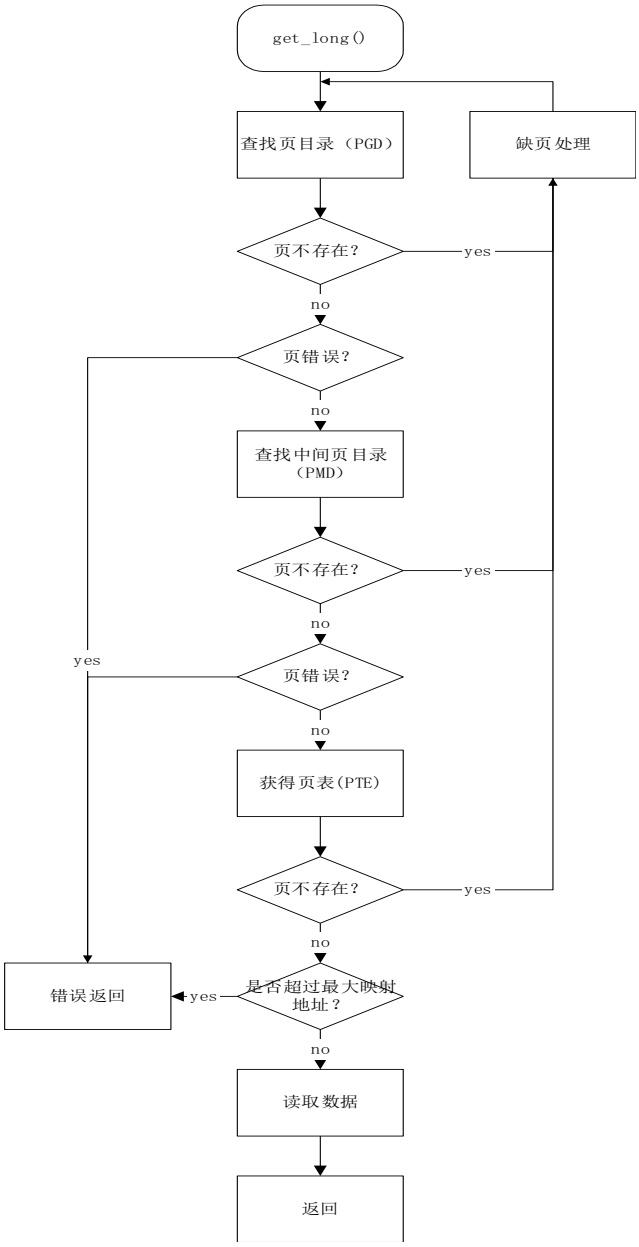
```
                return -EIO;          /*  优先级不为 3，出错  */
            child->tss.fs = value;    /*  通过 TSS 写寄存器 fs  */
            return 0;
        case GS:
            if (value && (value & 3) != 3)
                return -EIO;          /*  优先级不为 3，出错  */
            child->tss.gs = value;    /*  通过 TSS 写寄存器 gs  */
            return 0;
        case DS:
        case ES:
            if (value && (value & 3) != 3)
                return -EIO;          /*  优先级不为 3，出错  */
            value &= 0xffff;          /*  ds es 为 16 位 */
            break;
        case SS:
        case CS:
            if ((value & 3) != 3)
                return -EIO;          /*  优先级不为 3，出错  */
            value &= 0xffff;          /*  ss cs 为 16 位 */
            break;
        case EFL:
            value &= FLAG_MASK;       /*  EFLAG 访问权限设定  */
            value |= get_stack_long(child, EFL_OFFSET) & ~FLAG_MASK;
    }
    if (regno > GS*4)
        regno -= 2*4;                 /*  修正偏移量  */
    put_stack_long(child, regno - sizeof(struct pt_regs), value);
    return 0;
}

static unsigned long getreg(struct task_struct *child,
    unsigned long regno)
{
    unsigned long retval = ~0UL;
    switch (regno >> 2) {
        case FS:
            retval = child->tss.fs;           /*  通过 TSS 读寄存器 gs  */
            break;
        case GS:
            retval = child->tss.gs;           /*  通过 TSS 读寄存器 fs  */
            break;
        case DS:
        case ES:
        case SS:
```

```
            case CS:
                retval = 0xffff;                /*    ds es ss cs 为 16 位  */
            default:
                if (regno > GS*4)               /*    修正偏移量    */
                    regno -= 2*4;
                regno = regno - sizeof(struct pt_regs);
                retval &= get_stack_long(child, regno);
        }
        return retval;
}
```

3. 内存读写辅助函数

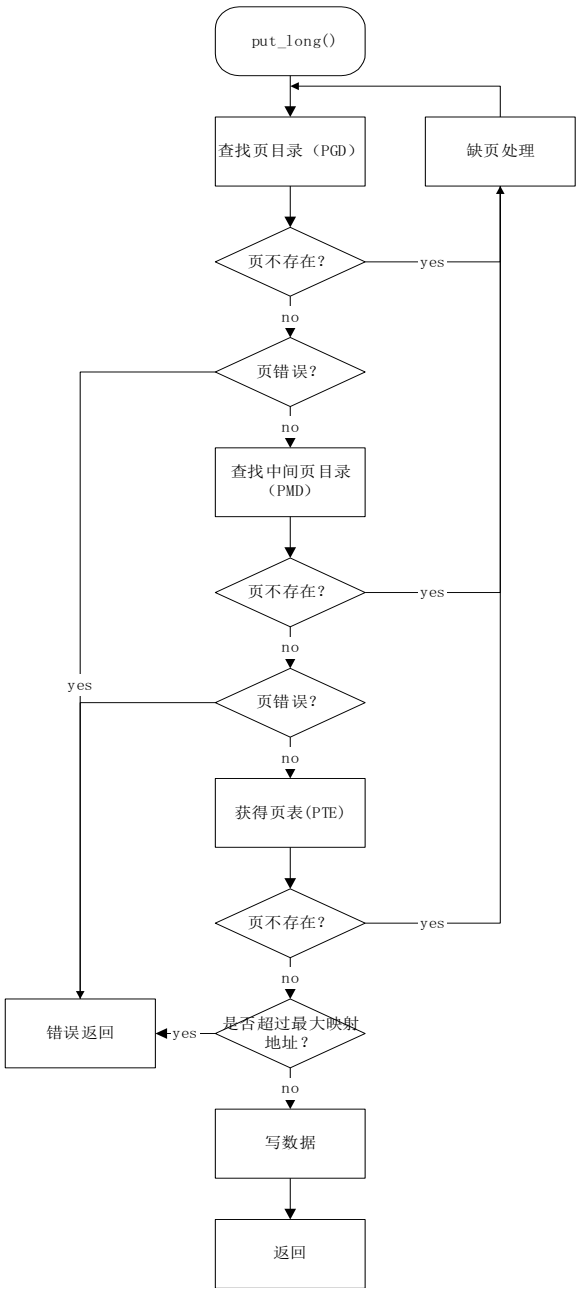
在 sys_ptrace 函数中对用户空间的访问通过辅助函数 write_long()和 read_long()函数完成的。访问进程空间的内存是通过调用 Linux 的分页管理机制完成的。从要访问进程的 task 结构中读出对进程内存的描述 mm 结构，并依次按页目录、中间页目录、页表的顺序查找到物理页，并进行读写操作。函数 put_long()和 get_long()完成的是对一个页内数据的读写操作。而 write_long()和 read_long()函数考虑了，所要访问的数据在两页之间，这时则需对两页分别调用 put_long()和 get_long()函数完成其功能。

其中相关函数和流程如下：



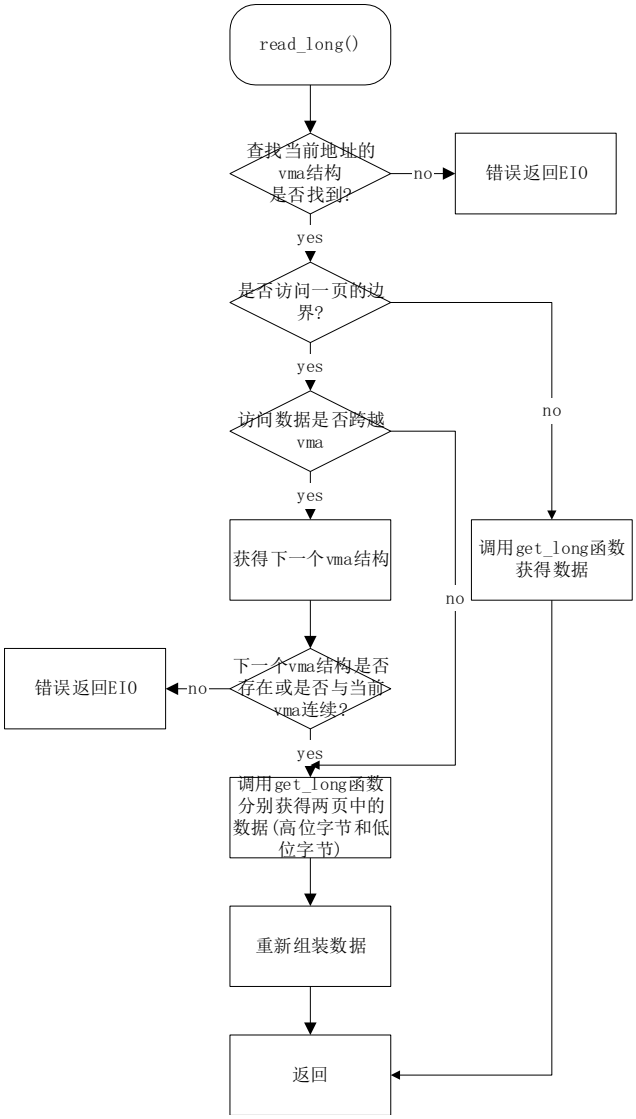

```
static unsigned long get_long(struct task_struct * tsk,
        struct vm_area_struct * vma, unsigned long addr)
{
        pgd_t * pgdir;
        pmd_t * pgmiddle;
        pte_t * pgtable;
        unsigned long page;

repeat:
        pgdir = pgd_offset(vma->vm_mm, addr);           /*   查找页目录   */
        if (pgd_none(*pgdir)) {
                handle_mm_fault(tsk, vma, addr, 0);     /*   缺页处理   */
                goto repeat;
```

```
        }
        if (pgd_bad(*pgdir)) {
                printk("ptrace: bad page directory %08lx\n", pgd_val(*pgdir));
                pgd_clear(pgdir);                       /*  页出错  */
                return 0;
        }
        pgmiddle = pmd_offset(pgdir, addr);             /*  查找中间页目录  */
        if (pmd_none(*pgmiddle)) {
                handle_mm_fault(tsk, vma, addr, 0);     /*  缺页处理  */
                goto repeat;
        }
        if (pmd_bad(*pgmiddle)) {
                printk("ptrace: bad page middle %08lx\n", pmd_val(*pgmiddle));
                pmd_clear(pgmiddle);                    /*  页出错  */
                return 0;
        }
        pgtable = pte_offset(pgmiddle, addr);           /*  查找页表  */
        if (!pte_present(*pgtable)) {
                handle_mm_fault(tsk, vma, addr, 0);     /*  缺页处理  */
                goto repeat;
        }
        page = pte_page(*pgtable);
        if (MAP_NR(page) >= max_mapnr)
                return 0;                               /*  越界出错  */
        page += addr & ~PAGE_MASK;
        return *(unsigned long *) page;
}
```



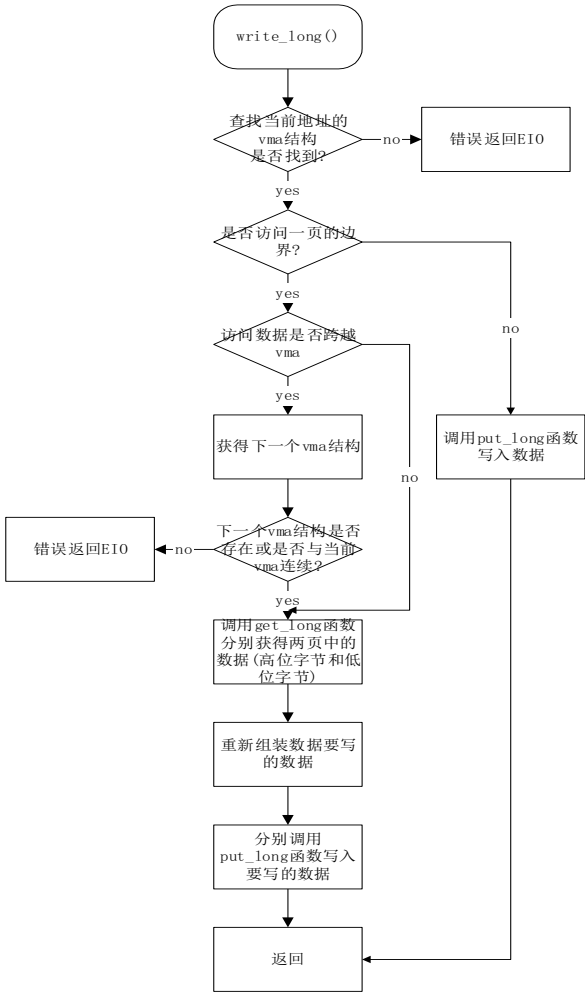

```
static void put_long(struct task_struct * tsk, struct vm_area_struct * vma, unsigned long addr,
        unsigned long data)
{
        pgd_t *pgdir;
        pmd_t *pgmiddle;
        pte_t *pgtable;
        unsigned long page;

repeat:
        pgdir = pgd_offset(vma->vm_mm, addr);           /*   查找页目录   */
        if (!pgd_present(*pgdir)) {
                handle_mm_fault(tsk, vma, addr, 1);     /*   缺页处理   */
```

```
        goto repeat;
    }
    if (pgd_bad(*pgdir)) {
        printk("ptrace: bad page directory %08lx\n", pgd_val(*pgdir));
        pgd_clear(pgdir);                       /*  页出错  */
        return;
    }
    pgmiddle = pmd_offset(pgdir, addr);         /*  查找中间页目录  */
    if (pmd_none(*pgmiddle)) {
        handle_mm_fault(tsk, vma, addr, 1);     /*  缺页处理  */
        goto repeat;
    }
    if (pmd_bad(*pgmiddle)) {
        printk("ptrace: bad page middle %08lx\n", pmd_val(*pgmiddle));
        pmd_clear(pgmiddle);                    /*  页出错  */
        return;
    }
    pgtable = pte_offset(pgmiddle, addr);       /*  查找页表  */
    if (!pte_present(*pgtable)) {
        handle_mm_fault(tsk, vma, addr, 1);
        goto repeat;
    }
    page = pte_page(*pgtable);                  /*  读页  */
    if (!pte_write(*pgtable)) {                 /*  是否可写  */
        handle_mm_fault(tsk, vma, addr, 1);     /*  页出错  */
        goto repeat;
    }
    if (MAP_NR(page) < max_mapnr)
        (unsigned long ) (page + (addr & ~PAGE_MASK)) = data; /*  写数据  */
    set_pte(pgtable, pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
    flush_tlb();
}
```




```
static int read_long(struct task_struct * tsk, unsigned long addr,
    unsigned long * result)
{
    struct vm_area_struct * vma = find_extend_vma(tsk, addr);   /*   查找对应的 VMA   */
    if (!vma)
        return -EIO;                                            /*   出错   */
    if ((addr & ~PAGE_MASK) > PAGE_SIZE-sizeof(long)) {   /*   是否跨页访问   */
        unsigned long low,high;
        struct vm_area_struct * vma_high = vma;
        if (addr + sizeof(long) >= vma->vm_end) {           /*   是否跨 VMA 访问   */
            vma_high = vma->vm_next;                        /*   获得下一个 VMA   */
            if (!vma_high || vma_high->vm_start != vma->vm_end)
                return -EIO;                                /*   出错   */
        }
        low = get_long(tsk, vma, addr & ~(sizeof(long)-1));    /*   低字节   */
        high = get_long(tsk, vma_high, (addr+sizeof(long)) & ~(sizeof(long)-1));
                                                            /*   高字节   */
```

```
        switch (addr & (sizeof(long)-1)) {        /*    重新组装数据    */
            case 1:
                low >>= 8;
                low |= high << 24;
                break;
            case 2:
                low >>= 16;
                low |= high << 16;
                break;
            case 3:
                low >>= 24;
                low |= high << 8;
                break;
        }
        *result = low;
    } else
        *result = get_long(tsk, vma, addr);              /*    非跨页访问    */
    return 0;
}
```

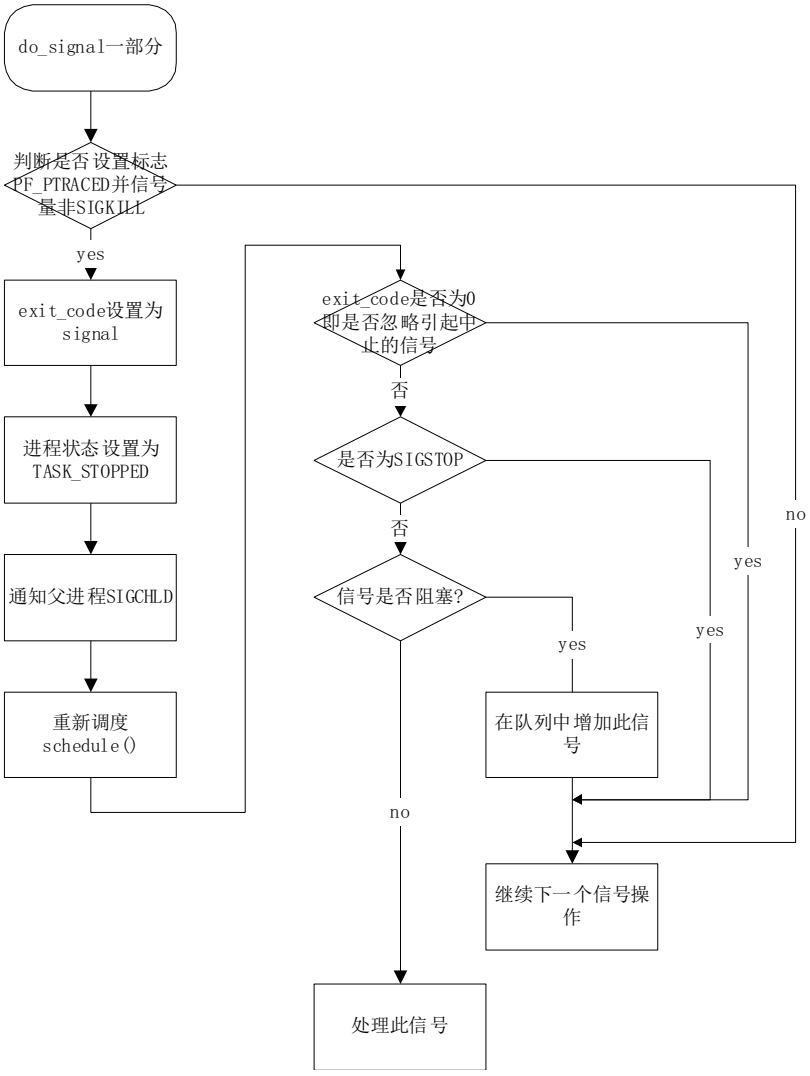


```
static int write_long(struct task_struct * tsk, unsigned long addr,
    unsigned long data)
{
    struct vm_area_struct * vma = find_extend_vma(tsk, addr); /*   查找对应的 VMA   */
    if (!vma)
        return -EIO;                                          /*   出错   */
    if ((addr & ~PAGE_MASK) > PAGE_SIZE-sizeof(long)) {
        unsigned long low,high;
        struct vm_area_struct * vma_high = vma;
        if (addr + sizeof(long) >= vma->vm_end) {             /*   是否跨 VMA 访问   */
            vma_high = vma->vm_next;                          /*   获得下一个 VMA   */
            if (!vma_high || vma_high->vm_start != vma->vm_end)
                return -EIO;                                  /*   出错   */
        }
        low = get_long(tsk, vma, addr & ~(sizeof(long)-1));   /*   获得原来低字节数据   */
        high = get_long(tsk, vma_high, (addr+sizeof(long)) & ~(sizeof(long)-1));
                                                              /*   获得原来高字节数据   */
        switch (addr & (sizeof(long)-1)) {                    /*   重新组装要回写的数据   */
            case 0:
                low = data;
                break;
            case 1:
                low &= 0x000000ff;
                low |= data << 8;
                high &= ~0xff;
                high |= data >> 24;
                break;
            case 2:
                low &= 0x0000ffff;
                low |= data << 16;
                high &= ~0xffff;
                high |= data >> 16;
                break;
            case 3:
                low &= 0x00ffffff;
                low |= data << 24;
                high &= ~0xffffff;
                high |= data >> 8;
                break;
        }
        put_long(tsk, vma, addr & ~(sizeof(long)-1),low);     /*   写低字节数据   */
        put_long(tsk, vma_high, (addr+sizeof(long)) & ~(sizeof(long)-1),high);
                                                              /*   写高字节数据   */
    } else
```

```
        put_long(tsk, vma, addr, data);          /*  非跨页访问  */
    return 0;
}
```

4. 信号处理

在进程接受到信号时，如果判断进程被跟踪，则中止当前进程，并通知其父进程。其操作在函数 do_signal 中处理。其处理流程如下：



其代码和说明注释如下：

```
        if ((current->flags & PF_PTRACED) && signr != SIGKILL) {
                                /*  判断是否被跟踪  */
            current->exit_code = signr;          /*  通知父进程造成中止的信号  */
                                                 /*  父进程 wait 接收  */
            current->state = TASK_STOPPED;/*  进程状态置为 TASK_STOPPED  */
            notify_parent(current, SIGCHLD);/*  通知父进程 SIGCHLD  */
            schedule();                          /*  重新调度，使调试器运行  */

            /*  调试器返回，继续执行  */
```

```
        if (!(signr = current->exit_code))  /*  是否忽略造成中止的信号  */
            continue;
        current->exit_code = 0;
        if (signr == SIGSTOP)               /*  忽略 SIGSTOP 信号
            continue;
        if (signr != info.si_signo) {
            info.si_signo = signr;                  /*  更新 siginfo 结构  */
            info.si_errno = 0;
            info.si_code = SI_USER;
            info.si_pid = current->p_pptr->pid;
            info.si_uid = current->p_pptr->uid;
        }
        if (sigismember(&current->blocked, signr)) { /*  信号是否被阻塞  */
            send_sig_info(signr, &info, current);    /*  把信号加入队列  */
            continue;
        }
    }
```

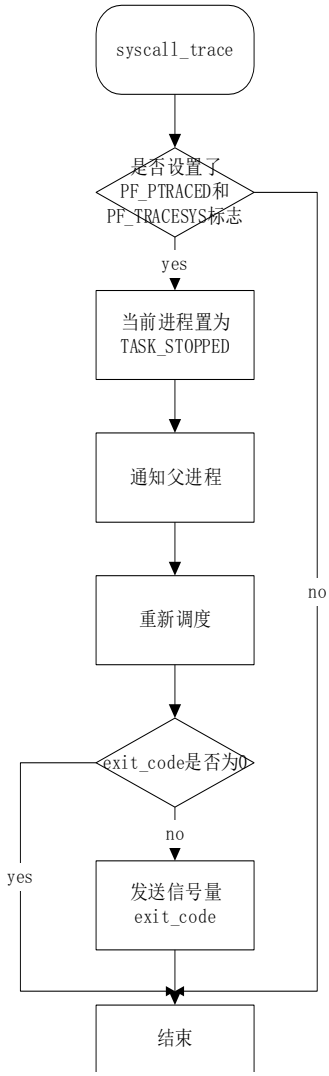
5. 系统调用跟踪

系统调用跟踪是一种使被调试进程在进入系统调用或完成系统调用时，中止进程的调试方法。此功能相当于在进入系统调用或系统调用完处设置断点。调试器通过调用 ptrace(PTRACE_SYSCALL) 使进程继续运行，直到系统调用开始或结束。在 ptrace(PTRACE_SYSCALL)处理中，设置了进程标志 PF_TRACESYS。在系统调用时如果判断进程标志设置了 PF_TRACESYS 则调用函数 syscall_trace。代码如下：

（/linux/arch/i386/kernel/entry.S）

```
    testb $0x20,flags(%ebx)     # PF_TRACESYS /*  判断 PF_TRACESYS 标志  */
    jne tracesys
    …….
tracesys:
    movl $-ENOSYS,EAX(%esp)
    call SYMBOL_NAME(syscall_trace)             /*  调用 syscall_trace  */
    movl ORIG_EAX(%esp),%eax
    call *SYMBOL_NAME(sys_call_table)(,%eax,4)  /*  调用系统调用  */
    movl %eax,EAX(%esp)     # save the return value
    call SYMBOL_NAME(syscall_trace)             /*  调用 syscall_trace  */
    jmp ret_from_sys_call
```

syscall_trace 函数在/linux/arch/i386/ptrace.c 中定义。syscall_trace 函数完成了系统调用中断的功能，其流程如下：



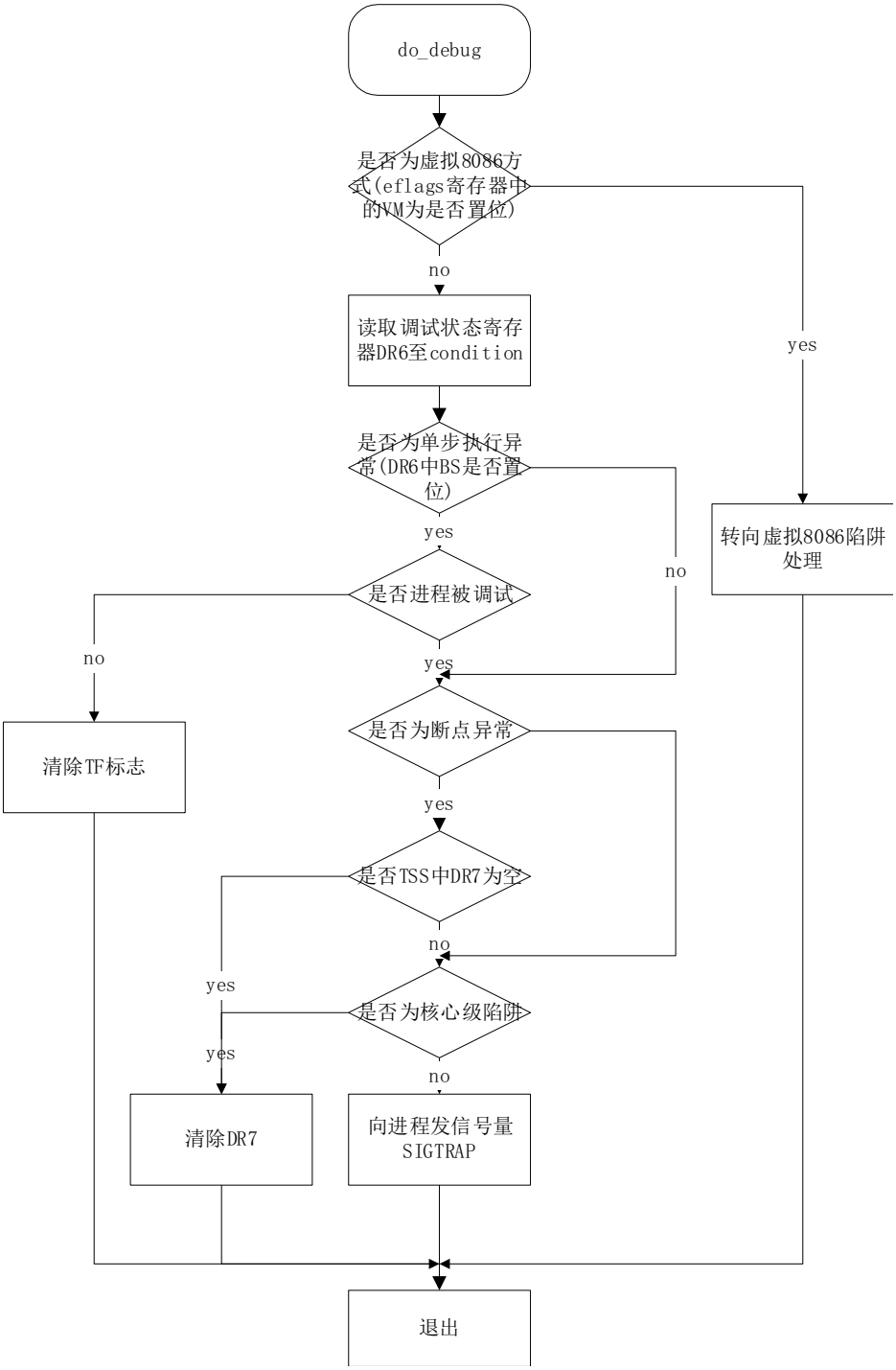
程序及说明注释如下：

```
asmlinkage void syscall_trace(void)
{
    if ((current->flags & (PF_PTRACED|PF_TRACESYS))
            != (PF_PTRACED|PF_TRACESYS))   /*   判断是否系统调用跟踪   */
        return;
    current->exit_code = SIGTRAP;          /*   通知父进程中止原因 SIGTRAP   */
    current->state = TASK_STOPPED;         /*   进程状态设置为 TASK_STOPPED   */
    notify_parent(current, SIGCHLD);       /*   通知父进程 SIGCHLD   */
    schedule();
    /*   重新调度，执行调试器   */
    /*   ……                    */
    /*   调试器命令继续执行      */
    if (current->exit_code) {                   /*   是否忽略信号   */
        send_sig(current->exit_code, current, 1);   /*   继续信号   */
        current->exit_code = 0;
    }
}
```

6. 调试陷阱处理

调试异常的编号为 1，在 Linux（i386）中由\linux\arch\i386\kernel\traps.c 中的函数 do_debug 完成。在 i386 中引起调试异常的条件有：程序断点（指令和数据断点）、单步执行、TSS 调试陷阱。do_debug 函数处理中对于正常的调试异常产生 SIGTRAP 信号，正常调试异常指通过 ptrace 调试进程，进程在调试状态下。对于一些非正常的调试异常则做一些清理工作，非正常的调试有可能是用户进程故意引起或一些寄存器没有初始化或设置。

do_debug 的处理流程如下：



程序及说明注释如下：

```
asmlinkage void do_debug(struct pt_regs * regs, long error_code)
{
    unsigned int condition;              /*   DR6 调试状态   */
    struct task_struct *tsk = current;

    if (regs->eflags & VM_MASK)          /*   判断是否是虚拟 8086 方式   */
        goto debug_vm86;

    __asm__ __volatile__("movl %%db6,%0" : "=r" (condition));   /*   读取 DR6 值   */

    if (condition & DR_STEP) {           /*   是否为单步异常   */
        if ((tsk->flags & (PF_DTRACE|PF_PTRACED)) == PF_DTRACE)
                                         /*   是否设置 PF_PTRACED   */
            goto clear_TF;               /*   为了防止用户态进程修改 TF 标志   */
                                         /*   以及 TF 标志错误设置造成异常      */
    }

    if (condition & (DR_TRAP0|DR_TRAP1|DR_TRAP2|DR_TRAP3)) {
                                         /*   是否为断点异常   */
        if (!tsk->tss.debugreg[7])       /*   是否 TSS 中 DR7 为 0   */
            goto clear_dr7;              /*   为了防止 DR7 的错误设置导致异常   */
    }

    if ((regs->xcs & 3) == 0)            /*   是否为核心引起异常   */
        goto clear_dr7;

    tsk->tss.trap_no = 1;
    tsk->tss.error_code = error_code;
    force_sig(SIGTRAP, tsk);             /*   产生 SIGTRAP 信号   */
    return;

debug_vm86:                              /*   转向虚拟 8086 陷阱处理   */
    lock_kernel();
    handle_vm86_trap((struct kernel_vm86_regs *) regs, error_code, 1);
    unlock_kernel();
    return;

clear_dr7:                               /*   清除 DR7   */
    __asm__("movl %0,%%db7"
        : /* no output */
        : "r" (0));
    return;
```

```
clear_TF:                                   /*  清除 TF 标志  */
    regs->eflags &= ~TF_MASK;
    return;
}
```

7. execve 系统调用

execve 系统调用完成的功能是，把当前进程代码替换为新的二进制代码，并执行。调用 execve 系统调用的进程如果已经被调试（PF_PTRACED 置位），则把代码转入后则会向自身发送信号 SIGTRAP，使其中止执行。这样可以使调试器装入代码后，进程停止在代码的最开始。

对于不同的二进制文件，execve 调用不同的代码。以下是 aout 和 elf 文件格式的代码转入程序中完成上述功能的代码。

do_load_elf_binary()函数（/linux/fs/binfmt_elf.c）

```
    。。。
    if (current->flags & PF_PTRACED)
        send_sig(SIGTRAP, current, 0);
    。。。
```

do_load_aout_binary()函数（/linux/fs/binfmt_aout.c）

```
    。。。
    if (current->flags & PF_PTRACED)
        send_sig(SIGTRAP, current, 0);
    。。。
```

四　ptrace 的使用

ptrace 提供了一种父进程可以控制子进程运行，并可以检查和改变它的核心的功能，ptrace 大多被调试器所使用。

1. 启动、中止调试程序

1) 启动调试程序

调试器对一个程序进行调试，如果此程序没有在运行，则调试器需要装入其代码，并创建相应的进程，并使其进入调试状态。其使用 ptrace 实现的方法如下：

```
  int pid;
  pid = fork ();
  if (pid < 0)
    perror_with_name ("fork");
```

```
if (pid == 0)
    {
      ptrace (PTRACE_TRACEME, 0, 0, 0);
      execv (program, allargs);       /* char *program;
                                          char **allargs; 指向程序名和参数
      fprintf (stderr, "Cannot exec %s: %s.\n", program,
               errno < sys_nerr ? sys_errlist[errno] : "unknown error");
      fflush (stderr);
      _exit (0177);
    }
    wait(pid);
    ptrace (PTRACE_CONT, pid, 0, 0);
    wait(pid);
```

首先调试器需创建进程、装入代码。调用 fork（），创建子进程。对于子进程首先调用 ptrace (PTRACE_TRACEME, 0, 0, 0)，进程标志 PF_PTRACED 置位，子进程进入调试状态。再执行 execv 系统调用，系统装入可执行文件代码，装入完毕后，则会向自身发送 SIGTRAP 信号**（参见 execve 分析）**。这时，子进程已经进入调试状态（PF_PTRACED 置位），则在处理 SIGTRAP 信号的时候， 中止进程执行，并通知调试器（其父进程）让其运行,父进程则从 wait 调用中返回**（参见 do_signal 分析）**。至此，要调试的程序已经调入、并作为调试器的一个子进程运行，并中止在程序的第一条指令上。

接着为了让子进程运行，父进程调用 ptrace (PTRACE_CONT, pid, 0, 0)，从而使子进程继续执行。再调用 wait 等待子进程中断或退出。

在 GDB 调试器中命令 run 则是通过上述方法完成运行调试运行程序的。

2) 对现有进程进行调试

调试器还能对已经运行的进程进行调试。通过调用 ptrace 的 PTRACE_ATTACH 功能可以实现。具体做法如下：

```
ptrace(PTRACE_ATTACH，pid, 0,0)
wait(pid);
```

首先调用 ptrace(PTRACE_ATTACH，pid, 0,0)。ptrace 为完成对这个进程的调试设置，首先设置进程标志置 PF_PTRACED。再将它设置为调试器的子进程，最后向它发信号 SIGSTOP 中止它的运行，使它进入调试状态。**（具体分析见 ptrace 函数分析）**

调试器在调用 ptrace 后需调用 wait，等待要调试的进程进入 STOP 状态。

在 GDB 调试器中命令 attach 则是通过上述方法调试现有的进程的。

3) 退出对进程调试

对于使用 ptrace 的 PTRACE_ATTACH 功能调试的程序，希望放弃调试，使其继续执行时可以使用 ptrace 的 PTRACE_DETACH 功能。

调用了 ptrace(PTRACE_DETACH，pid, 0,0)，终止调试一个子进程。此处理与 PTRACE_ATTACH 处理相反。在此做了一些清理操作：清除 PF_TRACESYS 和 PF_PTRACED 进程标志，清除 TF 标志，父进程指针还原。最后唤醒此进程，让其继续执

行（**具体分析见 ptrace 函数分析**）。

在 GDB 调试器中命令 detach 则是通过上述方法调试现有的进程的。

4) 终止调试进程运行

对被调试的进程，不想再调试时，可以调用 ptrace 的 PTRACE_KILL 功能杀死被调试的进程。

调用了 ptrace(PTRACE_KILL，pid, 0,0)，把子进程继续的信号设置为 SIGKILL，然后唤醒子进程，由于子进程是在 do_signal 处理中进入 stop 的，所以它将继续处理 SIGKILL 部分的代码，从而使子进程终止。**（具体分析见 ptrace 函数分析）**

在 GDB 调试器中命令 kill 则是通过上述方法调试现有的进程的。

2. 调试进程的控制

1) 中止进程运行

在使用调试器调试程序时，被调试程序被中断的条件有:：

1. 调试器设置的断点（指令断点和数据断点）满足条件。
2. 进程收到一个信号（SIGKILL 除外）。
3. 单步调用完成。
4. 系统调用调试下，进入或离开系统调用。

A. 断点

设置断点是调试器中的一个重要功能。80386 提供了两种方式，INT3 和利用调试寄存器**（详见前面 80386 的调试设施）**。

如果使用 INT3 方式设置断点，则调试器通过 ptrace 的 PTRACE_POKETEXT 功能在断点处插入 INT3 单字节指令。当进程运行到断点时（INT3 处），则系统进入异常 3 的处理。

若使用调试寄存器，则调试器通过调用 ptrace(PTRACE_POKEUSR，pid，0，data)在 DR0-DR3 寄存器设置与四个断点条件的每一个相联系的线性地址在 DR7 中设置断点条件。被跟踪进程运行到断点处时，CPU 产生异常 1，从而转至函数 do_debug 处理。由于子进程在调试状态下属于正常调试异常，所以 do_debug 函数处理中产生 SIGTRAP 信号，为处理这个信号，进入 do_signal，使被调试进程停止，并通知调试器（父进程），此时得到子进程终止原因为 SIGTRAP。

B. 信号

在有些情况之下，要求调试器调试某进程时，当进程收到某一信号的时候中断进程运行。如：被调试进程在某处运算错误，进程会接收到 SIGFPE 信号，在正常运行状况下，会 Coredump，而调试的情况下则希望在产生错误代码处停止运行，可以让用户调试错误原因。

对于已经被调试的进程（PF_PTRACED 标志置位），当受到任何信号（SIGKILL 除外）会中止其运行，并通知调试器（父进程）。**（详见 do_signal 分析）**

C. 单步执行

单步执行也是一种使进程中止的情况。当用户调用 ptrace 的 PTRACE_SINGLESTEP 功能时，ptrace 处理中，将用户态标志寄存器 EFLAG 中 TF 标志为置位，并让进程继续运行**（具体分析见 ptrace 函数分析）**。当进程回到用户态运行了一条指令后，CPU 产生异常 1，

从而转至函数 do_debug 处理。由于子进程在调试状态下属于正常调试异常，所以 do_debug 函数处理中产生 SIGTRAP 信号，为处理这个信号，进入 do_signal，使被调试进程停止，并通知调试器（父进程），此时得到子进程终止原因为 SIGTRAP。

D. 系统调用调试

对程序的调试，有时希望对系统调用进程跟踪。当程序进行系统调用时中断其运行。ptrace 提供 PTRACE_SYSCALL 功能完成此功能。在 ptrace 调用中设置了进程标志 PF_TRACESYS，表示进程对系统调用进行跟踪，并继续执行进程（具体分析见 ptrace 函数分析）。直到进程调用系统调用时，则中止其运行，并通知调试器（父进程）。**（详见 syscall_trace 分析）**

2) 继续进程执行

让中断的进程继续执行，ptrace 提供三种功能

1. 继续执行（PTRACE_CONT）
2. 系统调用调试（PTRACE_SYSCALL）
3. 单步执行（PTRACE_SINGLESTEP）

三种功能的区别在于 PTRACE_CONT 功能让进程继续执行直到下一个断点或收到一个信号会中止进程运行。PTRACE_SYSCALL 功能让进程继续执行增加了一个中止条件，进程调用系统调用。PTRACE_SINGLESTEP 功能让进程继续执行，只执行一个机器指令，则就中止其运行。

当被调试进程因为受到一个信号而中止时，这个信号并没有被处理。如果希望继续运行进程时继续处理这个信号，则在上述三个 ptrace 功能调用时，最后一个参数 data 设置要继续处理的信号。这种情况出现在，如：中止进程运行的信号为用户自定义信号，用户想继续运行进程，而不要忽略用户信号处理。有时，用户希望忽略其信号处理，这时则参数 data 设置为 0，这种情况出现在，如：由于算术错误接收到 SIGFPE 信号使进程中止，而用户发现了错误，重新设置了正确的值，然后希望其继续执行，这时 SIGFPE 信号则需要忽略。

3. 进程数据存取

当被调试进程中止的状态下，调试器一般提供给用户观察、修改程序变量的功能，以及反汇编代码，以及观察、修改内存地址和寄存器的功能。读写代码段数据使用 ptrace 的 PTRACE_PEEKTEXT 和 PTRACE_POKETEXT 功能。读写数据段数据使用 ptrace 的 PTRACE_PEEKDATA 和 PTRACE_POKEDATA 功能。读写寄存器数据使用 ptrace 的 PTRACE_PEEKUSR 和 PTRACE_POKEUSR 功能。对于 Linux（i386）下可以使用 ptrace 的 PTRACE_GETREGS 和 PTRACE_PUTREGS 读写所有 i386 用户寄存器，使用 ptrace 的 PTRACE_GETFPREGS 和 PTRACE_PUTFPREGS 读写所有 i387 浮点寄存器。