


【趣话计算机底层技术】调试器是个大骗子！

我叫GDB，是一个调试器，程序员通过我可以调试他们编写的软件，分析其中的bug。

作为一个调试器，调试分析是我的看家本领，像是给目标进程设置断点，或者让它单步执行，又或是查看进程中的变量、内存数据、CPU的寄存等等操作，我都手到擒来。

你只要输入对应的命令，我就能帮助你调试你的程序。

我之所以有这些本事，都得归功于一个强大的系统函数，它的名字叫ptrace。



```
long ptrace(  
    enum __ptrace_request request,  
    pid_t pid,  
    void *addr,  
    void *data  
);
```

不管是开始调试进程，还是下断点、读写进程数据、读写寄存器，我都是通过这个函数来进行，要是没了它，我可就废了。

它的第一个参数是一个枚举型的变量，表示要执行的操作，我支持的调试命令很多都是靠它来实现的：

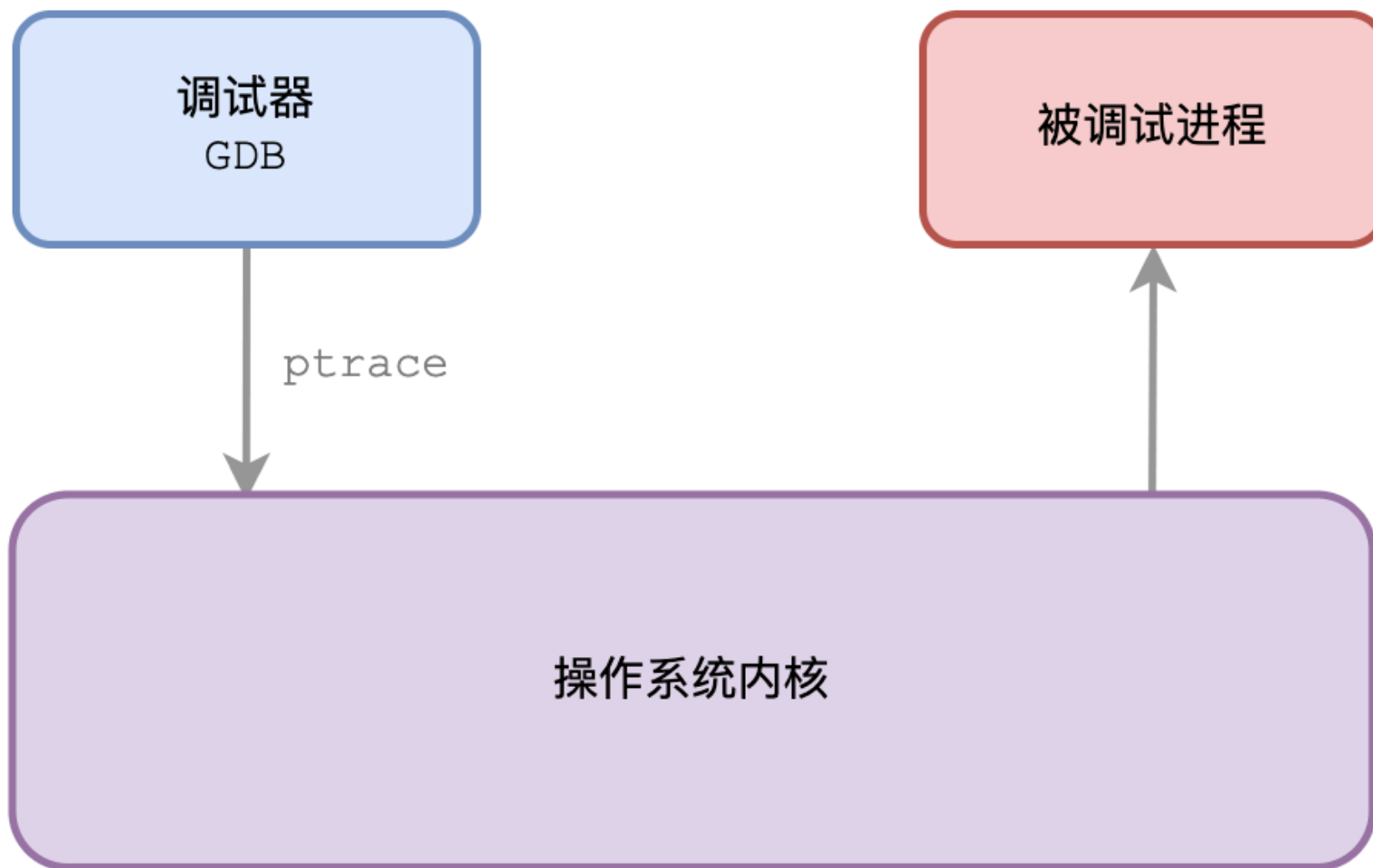
名称	值	说明
PTRACE_TRACEME	0	PTRACE_TRACEME用在目标进程中。表示子进程容许父进程跟踪自己。
PTRACE_PEEKTEXT	1	从目标进程的代码段中读取一个长整型，内存地址由参数addr指定。
PTRACE_PEEKDATA	2	从目标进程的数据段中读取一个长整型，内存地址由参数addr指定。
PTRACE_PEEKUSR	3	在调试coredump文件时，从USER区域中读取一个长整型。USER结构为coredump文件的前面一部分，它描述了进程中止时的一些状态，如：寄存器值，代码、数据段大小，代码、数据段开始地址等。USER区域地址由参数addr参数。
PTRACE_POKETEXT	4	向目标进程的代码段中写入一个长整型，内存地址由参数addr指定。
PTRACE_POKEDATA	5	向目标进程的数据段中写入一个长整型，内存地址由参数addr指定。
PTRACE_POKEUSR	6	往USER区域中写入一个长整型。
PTRACE_CONT	7	继续执行目标进程。参数pid表示被跟踪的目标进程。
PTRACE_KILL	8	中止目标进程。参数pid表示被跟踪的目标进程。
PTRACE_SINGLESTEP	9	设置单步执行标志，单步执行一条指令。被跟踪进程单步执行完一条指令后，被跟踪进程将被中止，并通知调试进程。
PTRACE_GETREGS	12	读取寄存器值，pid表示被跟踪的目标进程，返回的寄存器值保存在data指定的地址中，arm和x86因为寄存器不相同，所以这里data指向的数据结构也不一样
PTRACE_SETREGS	13	设置寄存器值，pid表示被跟踪的目标进程，data为寄存器数据地址。
PTRACE_GETFPREGS	14	读取浮点寄存器值，pid表示被跟踪的目标进程，返回的浮点寄存器值保存在data指定的地址中
PTRACE_SETFPREGS	15	设置浮点寄存器值，pid表示被跟踪的目标进程，data为浮点寄存器数据地址。
PTRACE_ATTACH	16	要求跟踪某个进程。参数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。
PTRACE_DETACH	17	结束跟踪某个进程。参数pid表示被跟踪的子进程。结束跟踪

PTRACE_DETACH	17	结束跟踪并「脱挂」。参数pid表示被跟踪的「脱挂」进程。结束跟踪后被跟踪进程将继续执行。
PTRACE_SYSCALL	24	继续执行被中止的进程。参数pid表示被跟踪的子进程。与PTRACE_CONT不同的是当被跟踪进程进行系统调用或者从系统调用中返回时，被跟踪进程将被中止，并通知调试进程。

你可以通过我来启动一个新的进程调试，我会使用fork创建出一个新的子进程，然后在子进程中通过execv来执行你指定的程序。

不过在执行你的程序之前，我会在子进程中调用ptrace函数，然后指定第一个参数为PTRACE_TRACEME，这样一来，我就能监控子进程中发生的事情了，也才能对你指定的程序进行调试。

你也可以让我attach到一个已经运行的进程分析，这样的话，我直接调用ptrace函数，并且指定第一个参数为PTRACE_ATTACH就可以了，然后我就会变成那个进程的父进程。



具体要选择哪种方式来调试，这就看你的需要了。不过不管哪种方式，最终我都会“接管”被调试的进程，它里面发生的各种信号事件我都能得到通知，方便我对它进行调试操作。

软件断点

作为一个调试器，最常用的功能就是给程序下断点了。

你可以通过`break`命令告诉我，你要在程序的哪个位置添加断点。

当我收到你的命令之后，我会偷偷把被调试进程中那个位置的指令修改为一个0xCC，这是一条特殊指令的CPU机器码——int 3，是x86架构CPU专门用来支持调试的指令。

我的这个修改是偷偷进行的，你如果通过我来查看被调试进程的内存数据，或者在反汇编窗口查看那里的指令，会发现跟之前一样，这其实是我使的障眼法，让你看起来还是原来的数据，实际上已经被我修改过了，你要是不信，你可以另外写个程序来查看那里的数据内容，看看我说的是不是真的。

一旦被调试的进程运行到那个位置，CPU执行这条特殊的指令时，会陷入内核态，然后取出中断描述符表IDT中的3号表项中的处理函数来执行。

IDT中的内容，操作系统一启动早就安排好了，所以系统内核会拿到CPU的执行权，随后内核会发送一个SIGTRAP信号给到被调试的进程。

被调试进程

调试器
GDB

```
.....  
00000010  8B FF          mov edi, edi  
00000015  55             push ebp  
0000001A  E8 11 00 00 00 call sub_fun  
.....
```

给地址00000015处下断点

```
.....  
00000010  8B FF          mov edi, edi  
00000015  CC             int 3  
0000001A  E8 11 00 00 00 call sub_fun  
.....
```

触发异常

IDT

发送信号
SIGTRAP

而因为我的存在，这个信号会被我截获，我收到以后会检查一下是不是程序员之前下的断点，如果是的话，就会显示断点触发了，然后等待程序员的下一步指示。

在没有下一步指示之前，被调试的进程都不会进入就绪队列被调度执行。

直到你使用`continue`命令告诉我继续，我再偷偷把替换成`int 3`的指令恢复，然后我再次调用`ptrace`函数告诉操作系统让它继续运行。

这就是我给程序下断点的秘密。

不知道你有没有发现一个问题，当我把替换的指令恢复后让它继续运行，以后就再也不会中断在这里了，可程序员并没有撤销这个断点，而是希望每次执行到这里都能中断，这可怎么办呢？

我有一个非常巧妙的办法，就是让它单步执行，只执行一条指令，然后又会上中断到我这里，但这时候我并不会通知程序员，而仅仅是把刚才恢复的断点又给打上（替换指令），然后就继续运行。这一切都发生的神不知鬼不觉，程序员根本察觉不到。

单步调试

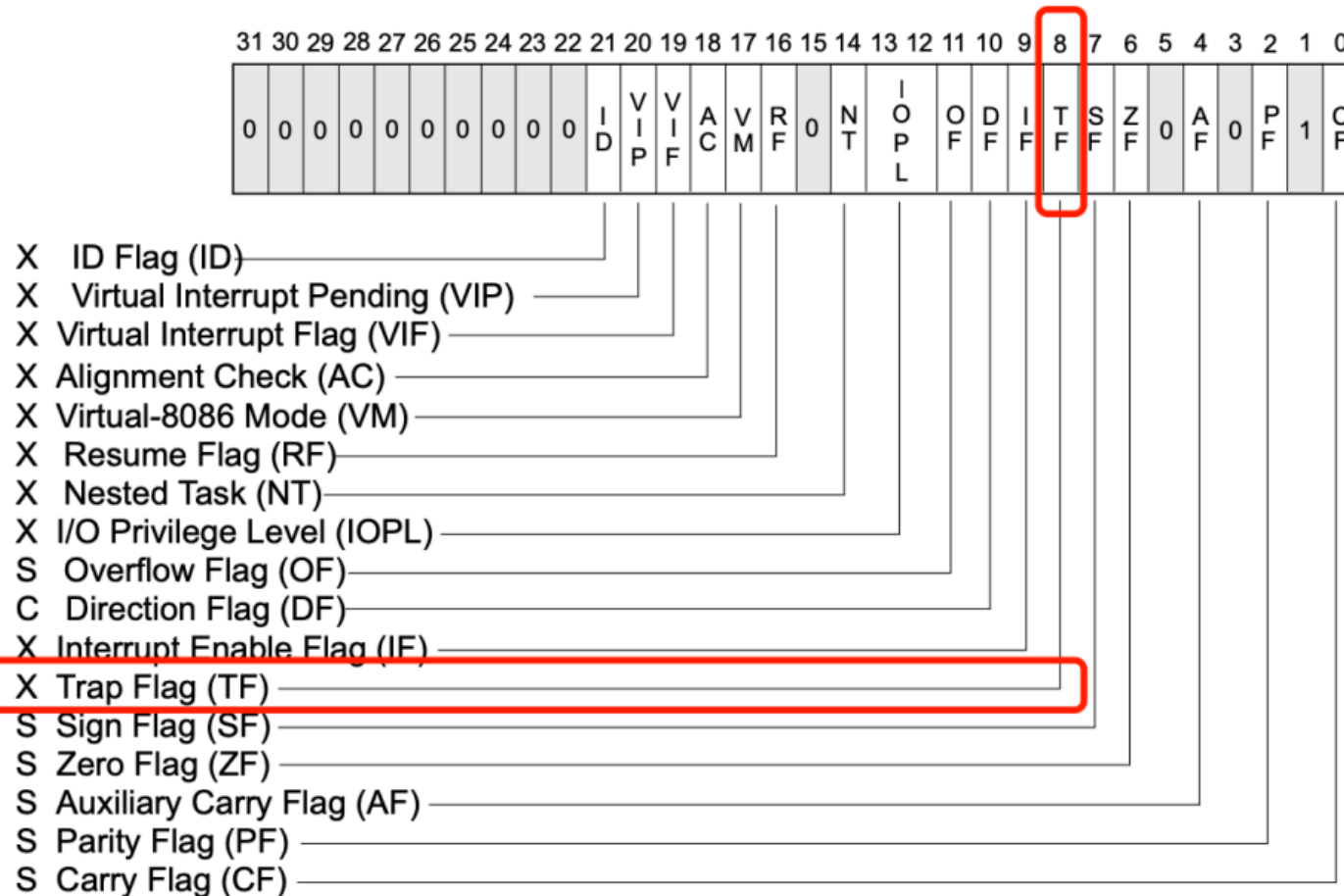
说到单步执行，应该算是程序员调试程序的时候除了下断点之外最常见的操作了，每一次只让被调试的进程运行一条指令，这样方便跟踪排查问题。

你可能很好奇我是如何让它单步执行的呢？

单步执行的实现可比下断点简单多了，我不用去修改被调试进程内存中的指令，只需要调用`ptrace`函数，传递一个`PTRACE_SINGLESTEP`参数就行了，操作系统会自动把它设置为单步执行的模式。

我也很好奇操作系统是怎么办到的，就去打听了一下。

原来x86架构CPU有一个标志寄存器，名叫`eflags`，它里面不止包含了程序运行的一些状态，还有一些工作模式的设定。



Reserved bit positions. DO NOT USE.
Always set to values previously read.

其中就有一个TF标记，用来告诉CPU进入单步执行模式，只要把这个标记为设置为1，CPU每执行一条指令，就会触发一次调试异常，调试异常的向量号是1，所以触发的时候，都会取出IDT中的1号表项中的处理函数来

执行。

接下来的事情就跟命中断点差不多了，我会截获到内核发给被调试进程的SIGTRAP信号，然后等待程序员的下一步指令。

如果你继续进行单步调试，那我便继续重复这个过程。

如果你有程序的源代码，你还可以进行源码级别的单步调试，不过这里的单步就指的是源代码中的一行了。

这种情况下要稍微麻烦一点，我还要分析出每一行代码对应的指令有哪些，然后用上面说的单步执行指令的方法，一条条指令快速掠过，直到这一行代码对应的指令都执行完成。

内存断点

有的时候，直接给程序中代码的位置下断点并不能包治百病。比如程序员发现某个内存地址的内容老是莫名其妙被修改，想知道到底是哪个函数干的，这时候连地址都没有，根本没法下断点。

单步执行也不行，那么多条指令，得执行到猴年马月去才能找到？

不用担心，我可以帮你解决这个烦恼。

你可以通过watch命令告诉我，让我监视被调试进程中某个内存地址的数据变化，一旦发现被修改，我都会把它给停下来报告给你。

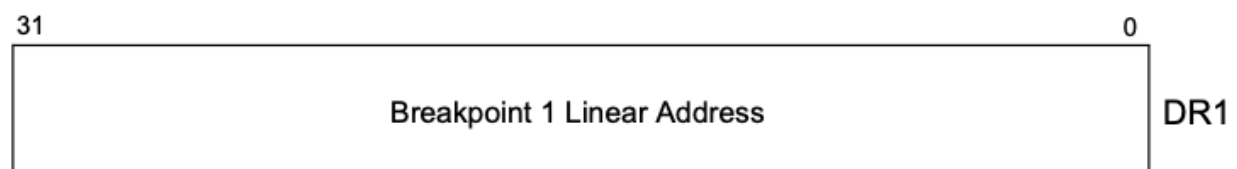
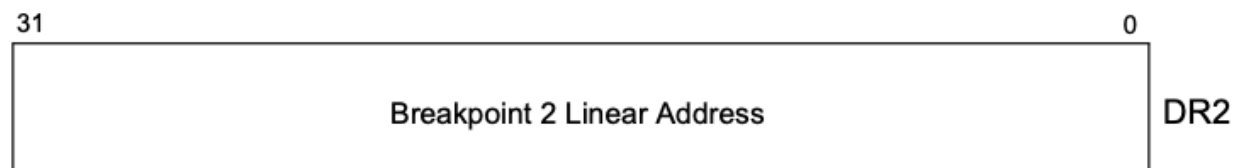
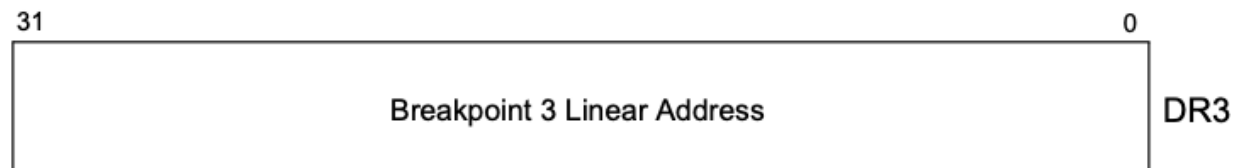
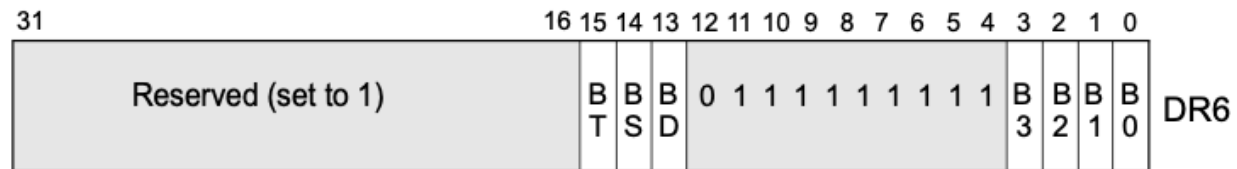
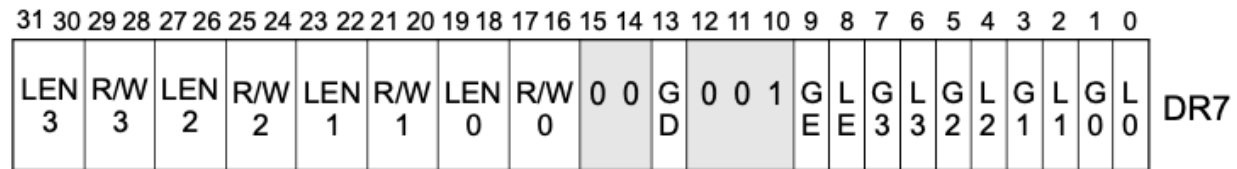
猜猜我是如何做到的呢？

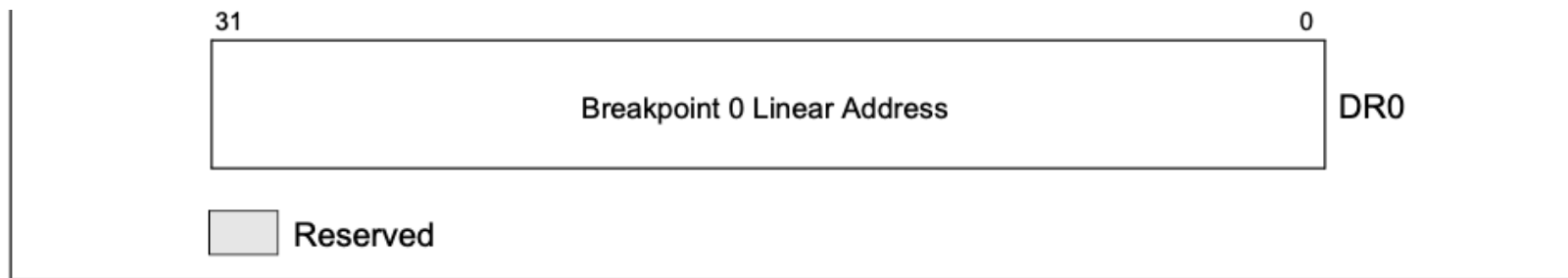
我可以用单步执行的方式，每执行一步，就检查一下内容有没有没修改，一旦发现就停下来通知你们程序员。

不过这种方式实在是太麻烦了，会严重拖垮被调试进程的性能。

好在x86架构的CPU提供了硬件断点的能力，帮我解决了大问题。

在x86架构CPU的内部内置了一组调试寄存器，从DR0到DR7，总共8个。通过在DR0-DR3中设置要监控的内存地址，然后在DR7中设置要监控的模式，是读还是写，剩下的交给CPU就好了。





CPU执行的时候，一旦发现有符合调试寄存器中设置的情况发生时，就会产生调试异常，然后取出IDT中的1号表项中的处理函数来执行，接下来的事情就跟单步调试产生的异常差不多了。

CPU内部依靠硬件电路来完成监控，可比我们软件一条一条的检查快多了！

现在，你不止可以使用watch命令来监控内存被修改，还可以使用rwatch、awatch命令来告诉我去监控内存被读或者被写。

我叫GDB，是你调试程序的好伙伴，现在你该知道我是如何工作的了吧！