

gdb的基本工作原理

刘东

雨丝风片@chinaunix

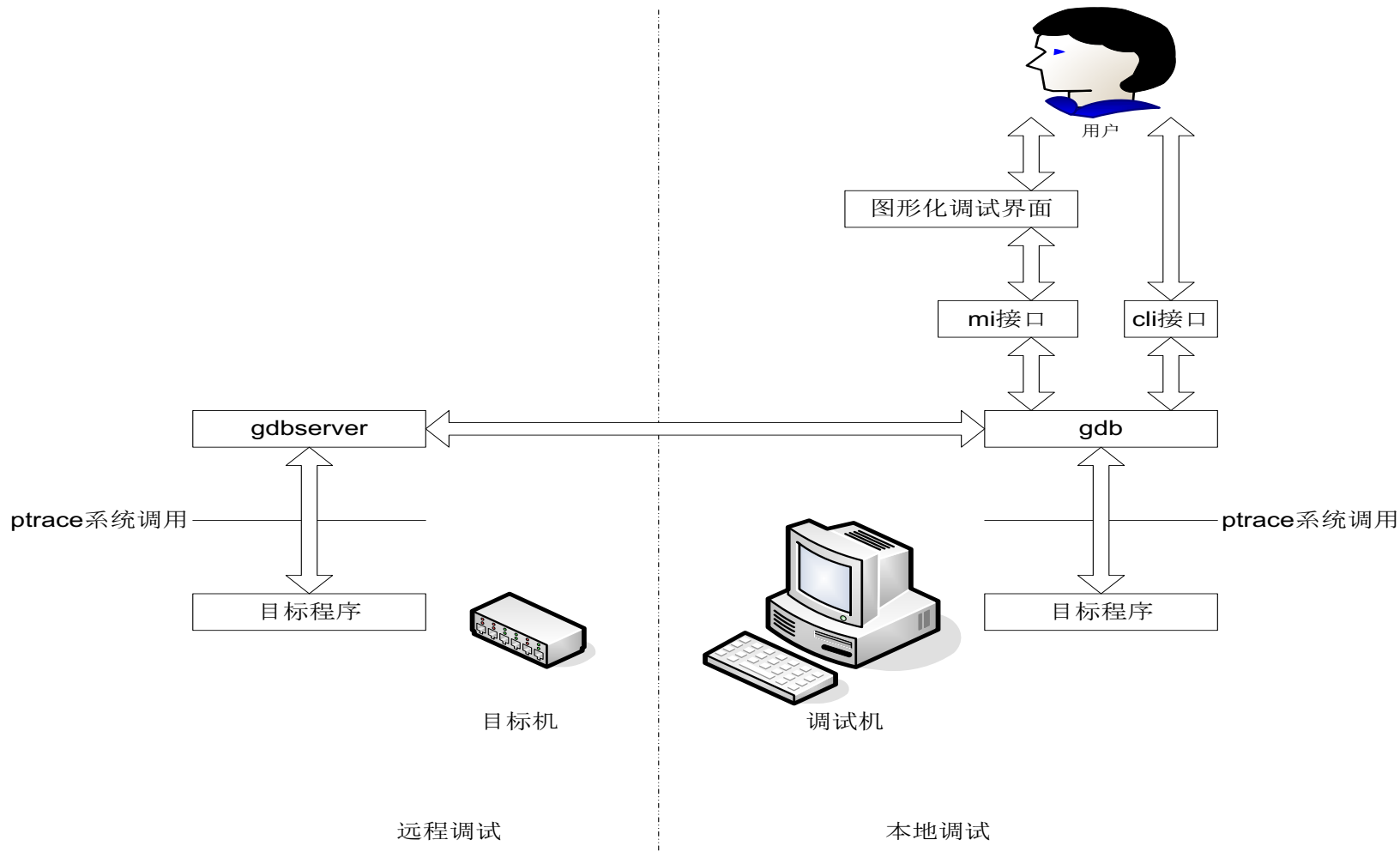
msn: yanbohuachuan@hotmail.com

2007.12.15

gdb简介

- gdb - GNU debugger。
 - gdb的主要功能 – 救死扶伤。
 - gdb的主要用途 – 修复bug；分析程序结构。
 - gdb官方网址 - <http://www.gnu.org/software/gdb/gdb.html>
 - gdb下载地址 - <http://ftp.gnu.org/gnu/gdb/>
-

gdb调试的组成架构



gdb调试的工具 – ptrace系统调用 (1)

- ptrace系统调用的原型

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

- ptrace系统调用的简要说明

ptrace系统调用提供了一种方法来让父进程可以观察和控制其它进程的执行，检查和改变其核心映像以及寄存器。

gdb调试的工具 – ptrace系统调用（2）

■ ptrace系统调用的主要选项

PTRACE_TRACEME

表示本进程将被其父进程跟踪，交付给这个进程的所有信号（除SIGKILL之外），都将使其停止，父进程将通过wait()获知这一情况。

PTRACE_ATTACH

attach到一个指定的进程，使其成为当前进程跟踪的子进程，子进程的行为等同于它进行了一次PTRACE_TRACEME操作。

PTRACE_CONT

继续运行之前停止的子进程。可同时向子进程交付指定的信号。

gdb的三种调试方式（1）

- **attach**并调试一个已经运行的进程

调试关系的建立过程：

- 用户确定需要进行调试的进程id;
- 运行gdb，输入**attach pid**，gdb对指定进程执行下述操作：
`ptrace(PTRACE_ATTACH, pid, 0, 0);`

--- gdb的三种调试方式（2）

■ attach并调试一个已经运行的进程

```
[root@localhost ~]# gdb
GNU gdb Red Hat Linux (6.5-16.el5rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb) attach 4253
Attaching to process 4253
Reading symbols from /root/test/a3...done.
Using host libthread_db library "/lib/libthread_db.so.1".
Reading symbols from /lib/libpthread.so.0...done.
[Thread debugging using libthread_db enabled]
[New Thread -1208772928 (LWP 4253)]
[New Thread -1219265648 (LWP 4255)]
[New Thread -1208775792 (LWP 4254)]
Loaded symbols for /lib/libpthread.so.0
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x00634402 in __kernel_vsyscall ()
(gdb)
```

gdb的三种调试方式（3）

- 运行并调试一个新的进程
- 调试关系的建立过程：
 - 运行gdb，通过命令行参数或file命令指定目标程序。
 - 输入run命令，gdb执行下述操作：
 - 通过fork()系统调用创建一个新进程；
 - 在新创建的子进程中执行下述操作：
`ptrace(PTRACE_TRACEME, 0, 0, 0);`
 - 在子进程中通过execv()系统调用加载用户指定的可执行文件。

--- gdb的三种调试方式（4）

■ 运行并调试一个新的进程

```
[root@localhost test]# gdb a3
GNU gdb Red Hat Linux (6.5-16.el5rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) run
Starting program: /root/test/a3
[Thread debugging using libthread_db enabled]
[New Thread -1208756544 (LWP 2551)]
[New Thread -1208759408 (LWP 2554)]
[New Thread -1219249264 (LWP 2555)]
```

gdb的三种调试方式（5）

- 远程调试目标机上新创建的进程
 - gdb运行在调试机上，gdbserver运行在目标机上，两者之间的通信数据格式由gdb远程串行协议（Remote Serial Protocol)定义。
 - RSP协议数据的基本格式为：“\$.....#xx”。
 - gdbserver的启动方式相当于运行并调试一个新创建的进程。

```
[root@localhost test]# gdbserver host:1100 a3
Process a3 created; pid = 2660
Listening on port 1100
```

--- gdb的三种调试方式（6）

■ 远程调试目标机上新创建的进程

```
[/cygdrive/d/test]$ gdb
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=i686-linux".
(gdb) file a3
Reading symbols from /cygdrive/d/test/a3...done.
(gdb) target remote 192.168.150.5:1100
Remote debugging using 192.168.150.5:1100
0x42e25810 in _start () from /cygdrive/d/test/lib/ld-linux.so.2
(gdb)
```

gdb调试的基础 – 信号（1）

- 在使用参数为PTRACE_TRACEME或PTRACE_ATTACH的ptrace系统调用建立调试关系之后，交付给目标程序的任何信号（除SIGKILL之外）都将被gdb先行截获，或在远程调试中被gdbserver截获并通知gdb。
- gdb因此有机会对信号进行相应处理，并根据信号的属性决定在继续目标程序运行时是否将之前截获的信号实际交付给目标程序。

gdb调试的基础 – 信号（2）

- 信号是实现断点功能的基础。以x86为例，向某个地址打入断点，实际上就是往该地址写入断点指令**INT 3**，即**0xCC**。目标程序运行到这条指令之后就会触发**SIGTRAP**信号，gdb捕获到这个信号，根据目标程序当前停止位置查询gdb维护的断点链表，若发现在该地址确实存在断点，则可判定为断点命中。
- gdb暂停目标程序运行的方法是向其发送**SIGSTOP**信号。
`kill_lwp(process->head.id, SIGSTOP);`

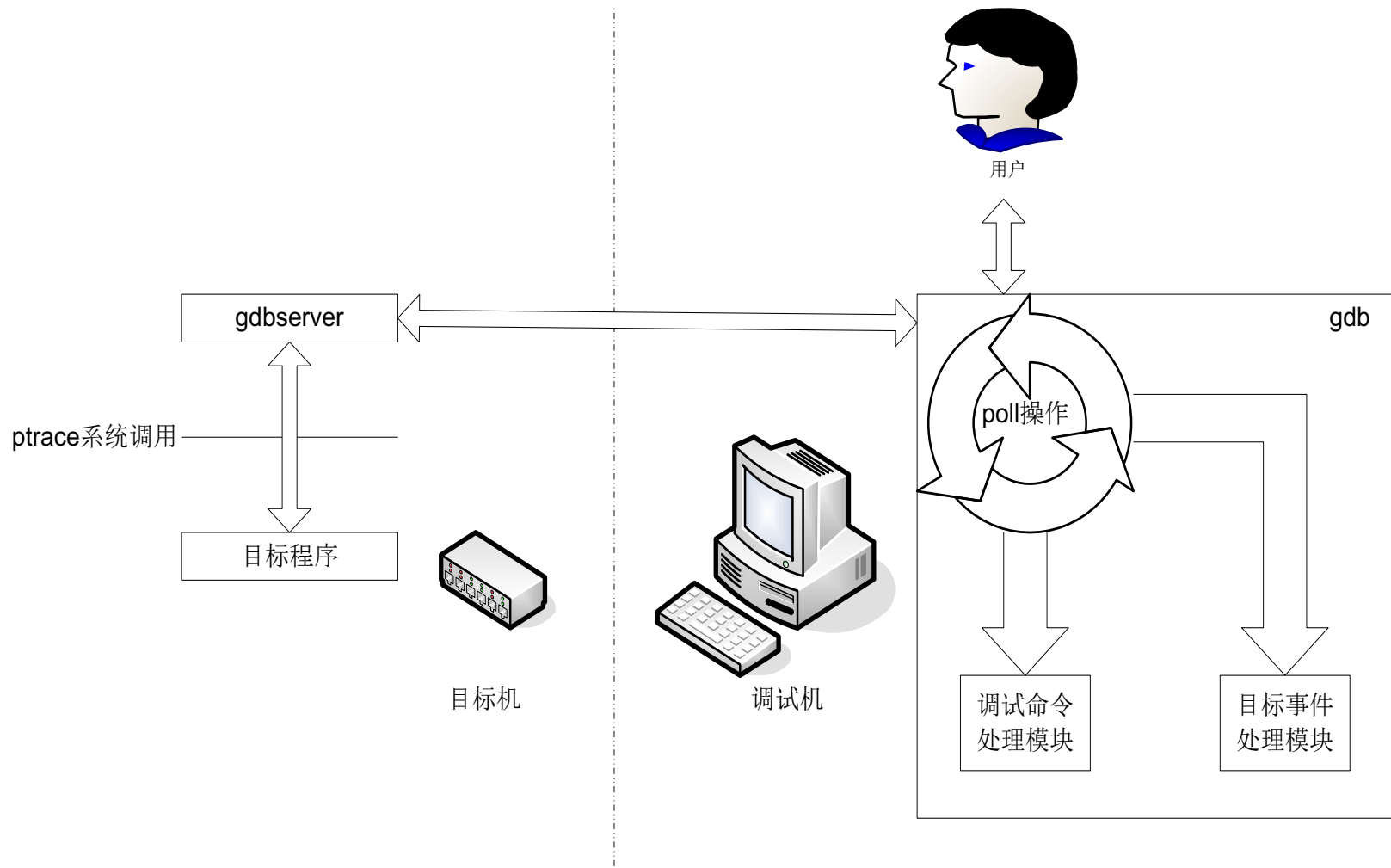
gdb的同步模式和异步模式

- 同步模式 – **gdb**将以同步方式等待目标程序发生停止事件，可称之为“死等”。因此，在目标程序运行期间，**gdb**不再扫描标准输入，用户也无法输入任何调试命令，要么等待目标程序发生调试事件而停止，要么通过“**Ctrl c**”来暂停目标程序的运行。
- 异步模式 – **gdb**不会同步等待目标程序发生停止事件，此类事件将通过异步上报的方式告知**gdb**。在目标程序运行期间，**gdb**仍将扫描标准输入，用户可以输入调试命令。
- 使用同步模式还是异步模式由调试目标决定，在启动**gdb**之后通过**target**命令的参数给出。比如远程同步目标为**target remote ...**，而远程异步目标为**target async ...**。

gdb远程异步模式中的事件循环（1）

- gdb远程异步模式的运转完全是由外部事件来激励的。共有两个外部事件源，一个是标准输入（用户输入的调试命令），一个是远程连接（gdbserver上报的异步事件）。
- 负责对外部事件源进行检测和对事件进行分发处理的功能模块就是事件循环。建立调试连接之后，gdb就会不断地对上述两个文件描述符进行poll操作。一旦发现某个文件描述符上有输入事件，就将该事件分发给相应的功能模块进行处理，该事件处理完毕之后将回到事件循环继续进行poll操作。

gdb远程异步模式中的事件循环（2）



gdb远程异步模式中的事件循环（3）

- 标准输入上会出现哪些事件？

用户输入的任何调试命令，比如cli的continue、next、step、breakpoint或者是mi的exec-continue、exec-next、exec-step、break-insert等等。

- 远程连接上会出现哪些事件？

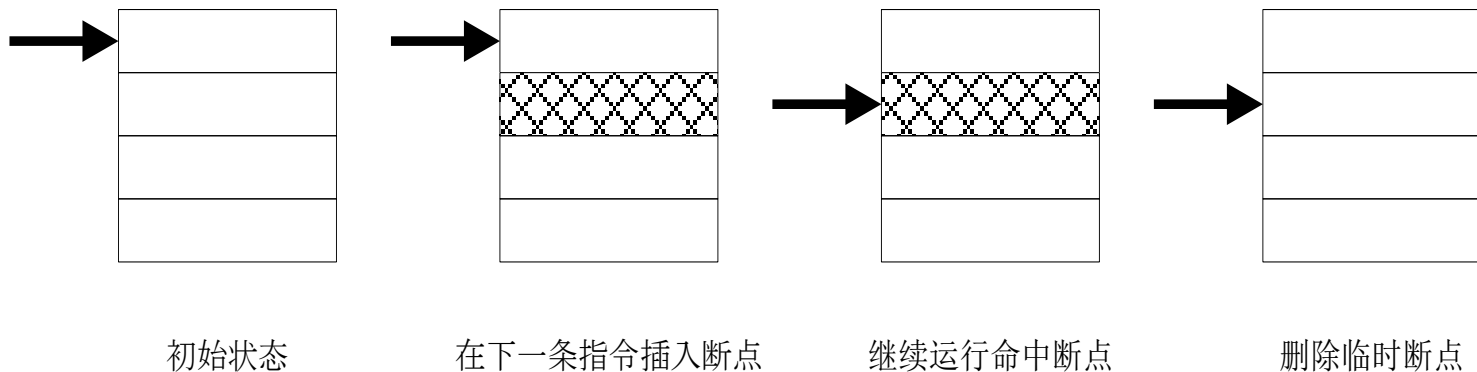
目标程序可能遇到的任何调试事件，比如遇到断点、收到随机信号、单步结束、线程创建、线程退出、进程退出等等。

gdb指令级单步的实现（1）

- 所谓指令级单步就是指gdb控制目标程序只运行一条指令之后即停止。指令级单步是next、step、nexti、stepi等运行类调试命令的基础。
- 指令级单步有硬件单步和软件单步之分。所谓硬件单步是指cpu架构本身就支持指令级单步，目标程序可以在运行一条指令之后自动停止。所谓软件单步是指cpu架构不支持指令级单步，需要gdb用软件方法来实现指令级单步。
- 支持硬件单步的架构如x86和ppc。对于x86，可通过设置EFLAGS寄存器中的TF标志来将cpu置于单步模式。对于ppc，则可通过设置MSR寄存器中的SE标志来将cpu置于单步模式。在单步模式中，cpu每执行一条指令，就会产生一个单步异常，通知gdb进行处理。

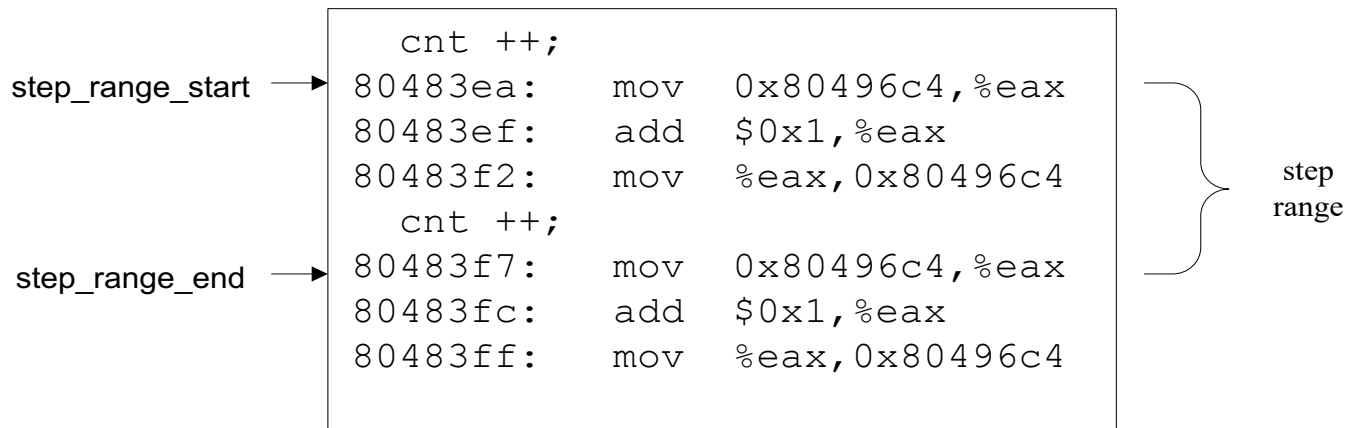
gdb指令级单步的实现（2）

- 不支持硬件单步的架构如arm和mips。对于此类架构，gdb采用的是用临时的软件断点来模拟单步的方法。即在需执行指令的下一条指令处临时插入一个断点，然后让目标程序继续运行，它会在执行完当前指令之后遇到下一条指令处的临时断点，于是目标程序停止，通知gdb命中断点，gdb再将此断点删除，由此来完成指令级单步的过程。（插入临时断点需要gdb实现代码分支预测的功能）



gdb next命令的实现（1）

- next命令实现c代码级的单步。分析其实现机制首先需要理解step range以及step_range_start和step_range_end的概念。
- 执行next命令时，gdb会计算出当前停止位置的c语句的第一条指令的地址作为step_range_start，然后计算出当前停止位置下一行的c语句的第一条指令的地址作为step_range_end，随后控制目标程序从当前停止位置开始走指令级单步，直至pc超出step range为止。

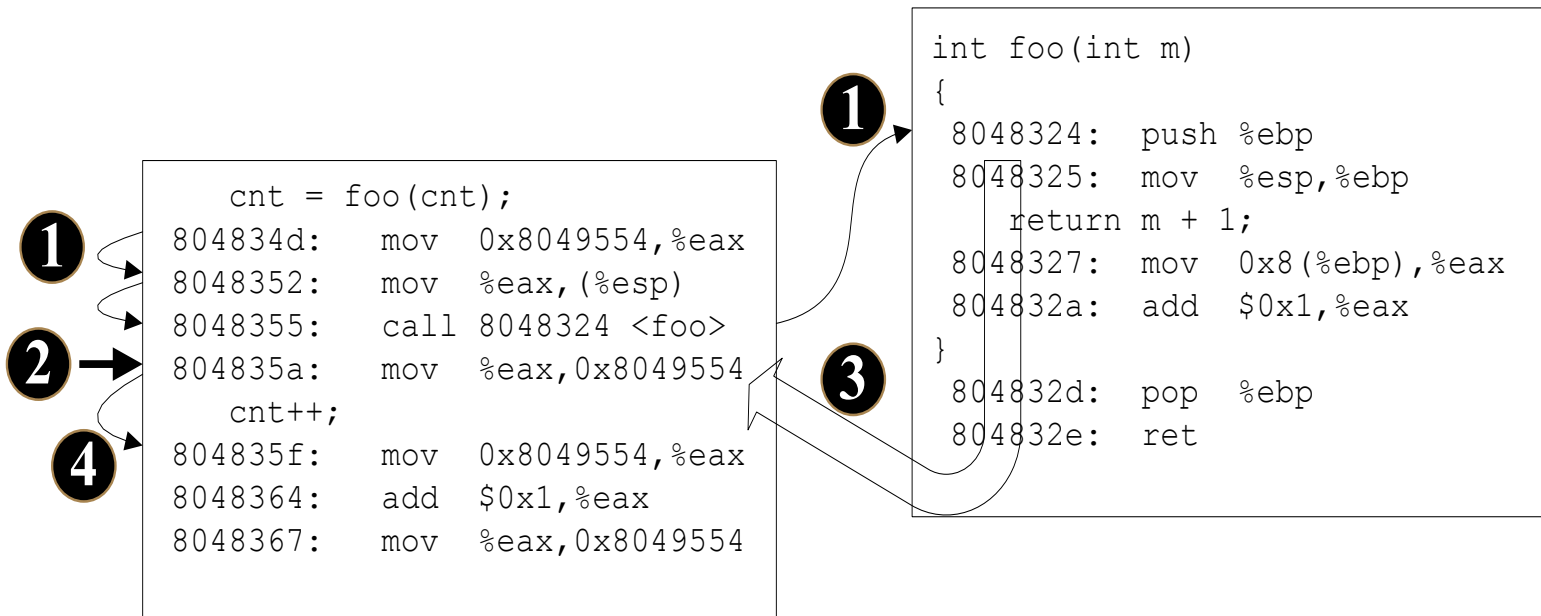


`gdb next`命令的实现（2）

- `next`命令的结束条件：
`pc < step_range_start || pc >= step_range_end`。
- 之所以不能简单地判断`pc`是否到达`step_range_end`，是因为`step_range_end`仅仅是c源代码意义上的下一行的第一条指令的地址，目标程序实际运行时未必就会到达那里。因此，`next`命令的结束条件可以理解为只要`pc`离开当前源代码行即可。
- `next`过程中遇到函数调用怎么办？我们知道，`next`命令是会跨过函数调用的，这个过程是如何实现的呢？

gdb next命令的实现 (3)

- **next**命令跨越函数调用的过程：**1**、从当前停止位置开始走指令级单步；**2**、走到子函数第一条指令时发现是函数调用，就在函数返回地址插入一个临时断点；**3**、让目标程序继续运行，通过子函数体，直至遇到之前插入的临时断点；**4**、继续走指令级单步，直至满足**next**命令的结束条件为止。



gdb step、nexti、stepi命令的实现

- **step**命令和**next**命令一样，也是实现c源代码级的单步，对于简单语句，**step**完全等同于**next**。唯一不同的是，若单步过程中遇到函数调用，**step**命令将停止在子函数的起始处，而不是将其跨越（无调试信息的子函数除外）。
- **nexti**命令实现指令级单步，和**next**命令类似，**nexti**命令单步过程中不会进入子函数调用。
- **stepi**命令实现指令级单步，而且是严格的指令级单步，每次直接走一条指令后即停止，不再区分是否存在函数调用。

`gdb finish`命令的实现

- **finish**命令会让目标程序继续执行完当前函数的剩余语句，并停止在返回到上一级函数之后的第一条指令处（也就是调用当前函数时的返回地址）。因此，实现**finish**命令时，只需找到当前函数的返回地址，并在该处插入一个临时断点，然后让目标程序继续运行，直至遇到该断点而停止。

gdb until命令的实现

- 不带参数的**until**命令让目标程序运行至当前函数中当前行后的任意一行。和**next**命令类似，这种**until**命令也是用指令级单步来实现的，但不同的是它的**step_range_start**设定为当前函数的起始位置，也就是说，若指令级单步过程中**pc**向函数前部移动，程序是不会停止的，仅当程序单步至当前行后的某一行时程序才会停止，这就提供了一种跳出循环体的快捷方式。
- 带参数的**until**命令让目标程序继续运行，直至达到指定位置为止。因为只要在当前函数体内，**until**命令的目的地址可以任意指定，因此不能再用指令级单步来实现它，而是采用在指定地址插入临时断点，然后让目标程序继续运行直至遇到断点停止的方法。
- 关于**until**命令需要注意的是，不管带参数还是不带参数，**until**都是针对当前函数内部而言的，也就是说，只要**pc**离开当前函数体程序就会停止。

gdb对断点的处理（1）

- 断点功能的实现就是在指定位置插入断点指令，使目标程序运行至该处时产生**SIGTRAP**信号，该信号被**gdb**捕获，通过断点地址的匹配确定是否命中断点。
- 断点的属性：
 - 是否有条件（由**condition**命令修改）；
 - 是否有忽略次数（由**ignore**命令修改）；
 - 是否只针对某个线程有效（由**break**命令的**thread**参数指定）；
 - 是否是临时断点（由**tbreak**命令插入）。

gdb对断点的处理（2）

- 断点命中的判定：目标程序遇到断点，并不一定就需要停下来，该停就停，不该停的还是要继续跑。只有真正需要停止运行的情况才认为是断点命中。是否命中断点的判定因素主要有以下这些：
 - 导致目标程序本次停止运行的信号是不是**SIGTRAP**；
 - 在**gdb**维护的断点链表中是否存在一个断点的地址与目标程序本次停止位置匹配；
 - 若断点存在条件，此时条件是否满足；
 - 断点的忽略次数此时是否为0；
 - 若断点只针对某个线程有效，那么遇到该断点的线程是否就是断点所设定的线程；
- 若前两个条件之一不满足，则认为目标程序本次是因随机信号而停止。若后三个条件之一不满足，则认为目标程序本次没有命中断点，**gdb**会让其继续运行。

gdb对断点的处理（3）

- 临时断点 – 断点命中之后的处理。当判定为断点命中之后，若该断点为临时断点，**gdb**就会将这个断点删除。也就是说，临时断点只命中一次。可能用到临时断点的场合：
 - 用户通过**tbreak**命令显式插入；
 - **next**、**nexti**、**step**命令需要跨越函数调用的时候，由**gdb**自动在函数返回地址处插入临时断点；
 - **finish**命令需要在当前函数返回地址处插入临时断点；
 - 带参数的**until**命令需要在当前函数返回地址以及参数指定地址插入临时断点；
 - 在不支持硬件单步的架构上，**gdb**需要逐指令插入临时断点来实现软件单步；

gdb对断点的处理（4）

- gdb将断点实际插入目标程序的时机：当用户通过**break**命令设置一个断点时，这个断点并不会立即生效，因为gdb此时只是在内部的断点链表中为这个断点新创建了一个节点而已。gdb会在用户下次发出继续目标程序运行的命令时，将所有断点插入目标程序，新设置的断点到这个时候才会实际存在于目标程序中。与此相呼应，当目标程序停止时，gdb会将所有断点暂时从目标程序中清除。
- 断点命中失败的情况下，跨越断点继续运行的过程：
 - 清除断点
 - 单步到断点的下一条指令
 - 恢复断点
 - 继续目标程序运行

gdb对随机信号的处理

- 对于gdb而言，导致目标程序本次停止的信号有随机和非随机之分。非随机信号是指gdb已经预知其会发生或者本身就是gdb导致的信号，也就是说，这些信号是具有明确的调试含义的，比如遇到断点指令时的SIGTRAP。而随机信号则是gdb没有预知的、不了解其实际含义的信号，比如因程序异常而导致的SIGSEGV，因定时机制而产生的SIGALRM，或者是用户程序自己内部使用的信号。
- 对于随机信号，gdb提供了两个属性来决定对它的处理方式。一个是当此信号发生时是否停止目标程序的运行，一个是在目标程序因此信号而停止之后，用户发出继续目标程序运行的命令时，是否将此信号交付给目标程序。
- 可通过info signals命令查看信号的配置属性，并通过handle signal命令来修改信号的属性。

谢谢!