调试多进程和多线程命令

1. 默认设置下，在调试多进程程序时 GDB 只会调试主进程。但是 GDB（>V7.0）支持多进程的**分别以及同时**调试，换句话说，GDB 可以同时调试多个程序。只需要设置 follow-fork-mode(默认值：parent)和 detach-on-fork（默认值：on）即可。

follow-fork-mode	detach-on-fork	说明
parent	on	只调试主进程（GDB 默认）
child	on	只调试子进程
parent	off	同时调试两个进程，gdb 跟主进程，子进程 block 在 fork 位置
child	off	同时调试两个进程，gdb 跟子进程，主进程 block 在 fork 位置

设置方法：set follow-fork-mode [parent|child]　set detach-on-fork [on|off]

查询正在调试的进程：info inferiors
切换调试的进程：　inferior <infer number>
添加新的调试进程：add-inferior [-copies n] [-exec executable] ,可以用 file executable 来分配给 inferior 可执行文件。
其他：remove-inferiors infno，　detach inferior

2. GDB 默认支持调试多线程，跟主线程，子线程 block 在 create thread。
查询线程：info threads
切换调试线程：thread <thread number>

例程：

```
#include <stdio.h>
#include <pthread.h>


void processA();
void processB();
void * processAworker(void *arg);


int main(int argc, const char *argv[])
 {
 int pid;


 pid = fork();
```

```
  if(pid != 0)
    processA();
  else
    processB();

  return 0;
  }

void processA()
  {
  pid_t pid = getpid();
  char prefix[] = "ProcessA: ";
  char tprefix[] = "thread ";
  int tstatus;
  pthread_t pt;

  printf("%s%lu %s\n", prefix, pid, "step1");

  tstatus = pthread_create(&pt, NULL, processAworker, NULL);
  if( tstatus != 0 )
    {
    printf("ProcessA: Can not create new thread.");
    }

  processAworker(NULL);
  sleep(1);
  }

void * processAworker(void *arg)
  {
  pid_t pid = getpid();
  pthread_t tid = pthread_self();
  char prefix[] = "ProcessA: ";
  char tprefix[] = "thread ";
```

```
  printf("%s%lu %s%lu %s\n", prefix, pid, tprefix, tid, "step2");
  printf("%s%lu %s%lu %s\n", prefix, pid, tprefix, tid, "step3");

  return NULL;
  }

 void processB()
  {
  pid_t pid = getpid();
  char prefix[] = "ProcessB: ";
  printf("%s%lu %s\n", prefix, pid, "step1");
  printf("%s%lu %s\n", prefix, pid, "step2");
  printf("%s%lu %s\n", prefix, pid, "step3");

  }
```

输出：

```
[cnwuwil@centos c-lab]$ ./test
ProcessA: 802 step1
ProcessB: 803 step1
ProcessB: 803 step2
ProcessB: 803 step3
ProcessA: 802 thread 3077555904 step2
ProcessA: 802 thread 3077555904 step3
ProcessA: 802 thread 3077553008 step2
ProcessA: 802 thread 3077553008 step3
```

调试：

1. 调试主进程，block 子进程。

```
(gdb) set detach-on-fork off
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is off.
(gdb) catch fork
Catchpoint 1 (fork)
(gdb) r
[Thread debugging using libthread_db enabled]
```

```
Catchpoint 1 (forked process 3475), 0x00110424 in __kernel_vsyscall ()
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.47.el6.i686
(gdb) break test.c:14
Breakpoint 2 at 0x8048546: file test.c, line 14.
(gdb) cont
[New process 3475]
[Thread debugging using libthread_db enabled]

Breakpoint 2, main (argc=1, argv=0xbffff364) at test.c:14
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.47.el6.i686
(gdb) info inferiors
  Num  Description       Executable
  2    process 3475      /home/cnwuwil/labs/c-lab/test
* 1    process 3472      /home/cnwuwil/labs/c-lab/test
```

2. 切换到子进程：

```
(gdb) inferior 2
[Switching to inferior 2 [process 3475] (/home/cnwuwil/labs/c-lab/test)]
[Switching to thread 2 (Thread 0xb7fe86c0 (LWP 3475))]
#0  0x00110424 in ?? ()
(gdb) info inferiors
  Num  Description       Executable
* 2    process 3475      /home/cnwuwil/labs/c-lab/test
  1    process 3472      /home/cnwuwil/labs/c-lab/test
(gdb) inferior 1
[Switching to inferior 1 [process 3472] (/home/cnwuwil/labs/c-lab/test)]
[Switching to thread 1 (Thread 0xb7fe86c0 (LWP 3472))]
#0  main (argc=1, argv=0xbffff364) at test.c:14
(gdb) info inferiors
  Num  Description       Executable
  2    process 3475      /home/cnwuwil/labs/c-lab/test
* 1    process 3472      /home/cnwuwil/labs/c-lab/test
```

3. 设断点继续调试主进程，主进程产生两个子线程：

```
(gdb) break test.c:50
Breakpoint 3 at 0x804867d: file test.c, line 50. (2 locations)
(gdb) cont
```

ProcessA: 3472 step1

[New Thread 0xb7fe7b70 (LWP 3562)]

ProcessA: 3472 thread 3086911168 step2

Breakpoint 3, processAworker (arg=0x0) at test.c:50

(gdb) info inferiors

  Num Description   Executable

  2   process 3475   /home/cnwuwil/labs/c-lab/test

* 1   process 3472   /home/cnwuwil/labs/c-lab/test

(gdb) info threads

  3 Thread 0xb7fe7b70 (LWP 3562) 0x00110424 in __kernel_vsyscall ()

  2 Thread 0xb7fe86c0 (LWP 3475) 0x00110424 in ?? ()

* 1 Thread 0xb7fe86c0 (LWP 3472) processAworker (arg=0x0) at test.c:50

4. 切换到主进程中的子线程，注意：线程 2 为前面产生的子进程

(gdb) thread 3

[Switching to thread 3 (Thread 0xb7fe7b70 (LWP 3562))]#0 0x00110424 in __kernel_vsyscall ()

(gdb) cont

ProcessA: 3472 thread 3086911168 step3

ProcessA: 3472 thread 3086908272 step2

[Switching to Thread 0xb7fe7b70 (LWP 3562)]

Breakpoint 3, processAworker (arg=0x0) at test.c:50

(gdb) info threads

* 3 Thread 0xb7fe7b70 (LWP 3562) processAworker (arg=0x0) at test.c:50

  2 Thread 0xb7fe86c0 (LWP 3475) 0x00110424 in ?? ()

  1 Thread 0xb7fe86c0 (LWP 3472) 0x00110424 in __kernel_vsyscall ()

(gdb) thread 1

Linux 下多线程查看工具(pstree、ps、pstack)

http://www.cnblogs.com/aixingfou/archive/2011/07/28/2119875.html

http://blog.csdn.net/nancygreen/article/details/14226925

先介绍一下 GDB 多线程调试的基本命令。 **info threads** 显示当前可调试的所有线程，每个线程会有一个 GDB 为其分配的 ID，后面操作线程的时候会用到这个 ID。 前面有*的是当前调试的线程。 **thread ID** 切换当前调试的线程为指定 ID 的线程。 **break thread_test.c:123 thread all** 在所有线程中相应的行上设置断点 **thread apply ID1 ID2 command** 让一个或者多个线程执行 GDB 命令 command。 **thread apply all command** 让所有被调试线程执行 GDB 命令 command。 **set scheduler-locking off|on|step** 估计是实际使用过多线程调试的人都可以发现，在使用 step 或者 continue 命令调试当前被调试线程的时候，其他线程也是同时执行的，怎么只让被调试 程序执行呢？通过这个命令就可以实现这个需求。off 不锁定任何线程，也就是所有线程都执行，这是默认值。 on 只有当前被调试程序会执行。 step 在单步的时候，除了 next 过一个函数的情况(熟悉情况的人可能知道，这其实是一个设置断点然后 continue 的行为)以外，只有当前线程会执行。

gdb 对于多线程程序的调试有如下的支持：

- 线程产生通知：在产生新的线程时, gdb 会给出提示信息

```
(gdb) r
Starting program: /root/thread
[New Thread 1073951360 (LWP 12900)]
[New Thread 1082342592 (LWP 12907)]---以下三个为新产生的线程
[New Thread 1090731072 (LWP 12908)]
[New Thread 1099119552 (LWP 12909)]
```

- 查看线程：使用可以查看运行的线程。info threads

```
(gdb) info threads
  4 Thread 1099119552 (LWP 12940)   0xffffe002 in ?? ()
  3 Thread 1090731072 (LWP 12939)   0xffffe002 in ?? ()
  2 Thread 1082342592 (LWP 12938)   0xffffe002 in ?? ()
* 1 Thread 1073951360 (LWP 12931)   main (argc=1, argv=0xbfffda04) at thread.c:21
(gdb)
```

注意，行首的蓝色文字为 gdb 分配的线程号，对线程进行切换时，使用该该号码，而不是上文标出的绿色数字。

另外，行首的红色星号标识了当前活动的线程

- 切换线程：使用 thread THREADNUMBER 进行切换，THREADNUMBER 为上文提到的线程号。下例显示将活动线程从 1 切换至 4。

```
(gdb) info threads
   4 Thread 1099119552 (LWP 12940)   0xffffe002 in ?? ()
   3 Thread 1090731072 (LWP 12939)   0xffffe002 in ?? ()
   2 Thread 1082342592 (LWP 12938)   0xffffe002 in ?? ()
* 1 Thread 1073951360 (LWP 12931)   main (argc=1, argv=0xbfffda04) at thread.c:21
(gdb) thread 4
[Switching to thread 4 (Thread 1099119552 (LWP 12940))]#0   0xffffe002 in ?? ()
(gdb) info threads
* 4 Thread 1099119552 (LWP 12940)   0xffffe002 in ?? ()
   3 Thread 1090731072 (LWP 12939)   0xffffe002 in ?? ()
   2 Thread 1082342592 (LWP 12938)   0xffffe002 in ?? ()
   1 Thread 1073951360 (LWP 12931)   main (argc=1, argv=0xbfffda04) at
thread.c:21
(gdb)
```

以上即为使用 gdb 提供的对多线程进行调试的一些基本命令。另外，gdb 也提供对线程的断点设置以及对指定或所有线程发布命令的命令。

初次接触 gdb 下多线程的调试，往往会忽视 gdb 中活动线程的概念。一般来讲，在使用 gdb 调试的时候，只有一个线程为活动线程，如果希望得到其他的线程的输出结果，必须使用 thread 命令切换至指定的线程，才能对该线程进行调试或观察输出结果。

最后介绍一下我最近遇见的一个多线程调试和解决。

基本问题是在一个 Linux 环境中，调试多线程程序不正常，info threads 看不到多线程的信息。
我先用命令 maintenance print target-stack 看了一下 target 的装载情况，发现 target"multi-thread"没有被装载，用 GDB 对 GDB 进行调试，发现在 函数 check_for_thread_db 在调用 libthread_db 中的函数 td_ta_new 的时候，返回了 TD_NOLIBTHREAD，所 以没有装载 target"multi-thread"。
在时候我就怀疑是不是 libpthread 有问题，于是检查了一下发现了问题，这个环境中的 libpthread 是被 strip 过的，我想可能 就是以为这个影响了 td_ta_new 对 libpthread 符号信息的获取。当我换了一个没有 strip 过的 libpthread 的时候，问题果然解决 了。
最终我的解决办法是拷贝了一个.debug 版本的 libpthread 到 lib 目录中，问题解决了。
多线程如果 dump，多为段错误，一般都涉及内存非法读写。可以这样处理，使用下面的命令打开系统开关，让其可以在死掉的时候生成 core 文件。
ulimit -c unlimited

这样的话死掉的时候就可以在当前目录看到 core.pid(pid 为进程号)的文件。接着使用 gdb:
gdb ./bin ./core.pid
进去后，使用 bt 查看死掉时栈的情况，在使用 frame 命令。

还有就是里面某个线程停住，也没死，这种情况一般就是死锁或者涉及消息接受的超时问题(听人说的，没有遇到过)。遇到这种情况，可以使用：
gcore pid （调试进程的 pid 号）
手动生成 core 文件，在**使用 pstack(linux 下好像不好使)查看堆栈的情况**。如果都看不出来，就仔细查看代码，看看是不是在 if，return，break，continue 这种语句操作是忘记解锁，还有嵌套锁的问题，都需要分析清楚了。

原文地址
http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=program&Number=692404&page=0&view=collapsed

pstack 在我机子上貌似不好使用

```
yingc@yingc:~/tmp/sisuo$ pstac
k 13011
Could not attach to target
13011: Operation not permitted.
detach: No such process
yingc@yingc:~/tmp/sisuo$ sudo
pstack 13011
[sudo] password for yingc:

13011: ./lock
(No symbols found in )
```

```
(No symbols found in
/lib/i386-linux-gnu/libc.so.6)
(No symbols found in
/lib/ld-linux.so.2)
0xb778bc7c: _fini + 0x2caec
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
0x09279010: _fini + 0x123054c
(b759db40, 1, b778bc6c,
15a6a100, e608f43e, 0) +
ffffffe0
```

```
yingc@yingc:~/tmp/sisuo$ sudo gdb -q ./lock 13011
Reading symbols from ./lock...done.
Attaching to program: /home/yingc/tmp/sisuo/lock, process 13011
Reading symbols from /lib/i386-linux-gnu/libpthread.so.0...Reading
symbols from
/usr/lib/debug//lib/i386-linux-gnu/libpthread-2.19.so...done.
done.
[New LWP 13015]
[New LWP 13014]
[New LWP 13013]
[New LWP 13012]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Loaded symbols for /lib/i386-linux-gnu/libpthread.so.0
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading symbols from
/usr/lib/debug//lib/i386-linux-gnu/libc-2.19.so...done.
done.
Loaded symbols for /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/ld-linux.so.2...Reading symbols from
/usr/lib/debug//lib/i386-linux-gnu/ld-2.19.so...done.
done.
Loaded symbols for /lib/ld-linux.so.2
0xb778bc7c in __kernel_vsyscall ()
(gdb) info threads
  Id   Target Id         Frame
  5    Thread 0xb759db40 (LWP 13012) "lock" 0xb778bc7c in
__kernel_vsyscall ()
  4    Thread 0xb6d9cb40 (LWP 13013) "lock" 0xb778bc7c in
__kernel_vsyscall ()
  3    Thread 0xb659bb40 (LWP 13014) "lock" 0xb778bc7c in
__kernel_vsyscall ()
  2    Thread 0xb5d9ab40 (LWP 13015) "lock" 0xb778bc7c in
__kernel_vsyscall ()
* 1    Thread 0xb759e700 (LWP 13011) "lock" 0xb778bc7c in
```

```
35 __kernel_vsyscall ()
36 (gdb) t 5
37 [Switching to thread 5 (Thread 0xb759db40 (LWP 13012))]
38 #0  0xb778bc7c in __kernel_vsyscall ()
39 (gdb) bt
   #0  0xb778bc7c in __kernel_vsyscall ()
   #1  0xb775b792 in __lll_lock_wait ()
   at ../nptl/sysdeps/unix/sysv/linux/i386/i686/../i486/lowlevellock.S:144
   #2  0xb77571e3 in _L_lock_851 () from /lib/i386-linux-gnu/libpthread.so.0
   #3  0xb7757020 in __GI___pthread_mutex_lock (mutex=0x804a060 <mutex2>)
   at ../nptl/pthread_mutex_lock.c:79
   #4  0x08048738 in func1 () at lock.cpp:18
   #5  0x080487ee in thread1 (arg=0x0) at lock.cpp:43
   #6  0xb7754f16 in start_thread (arg=0xb759db40) at pthread_create.c:309
   #7  0xb768b9fe in clone () at ../sysdeps/unix/sysv/linux/i386/clone.S:129
   (gdb)
```

一、多线程调试

多线程调试重要就是下面几个命令：

info thread 查看当前进程的线程。

thread <ID> 切换调试的线程为指定 ID 的线程。

break file.c:100 thread all 在 file.c 文件第 100 行处为所有经过这里的线程设置断点。

set scheduler-locking off|on|step，这个是问得最多的。

在使用 step 或者 continue 命令调试当前被调试线程的时候，其他线程也是同时执行的，怎么只让被调试程序执行呢？通过这个命令就可以实现这个需求。

off 不锁定任何线程，也就是所有线程都执行，这是默认值。

on 只有当前被调试程序会执行。

step 在单步的时候，除了 next 过一个函数的情况(熟悉情况的人可能知道，这其实是一个设置断点然后 continue 的行为)以外，只有当前线程会执行。

二、调试宏

这个问题超多。在 GDB 下，我们无法 print 宏定义，因为宏是预编译的。但是我们还是有办法来调试宏，这个需要 GCC 的配合。

在 GCC 编译程序的时候，加上**-ggdb3** 参数，这样，你就可以调试宏了。

另外，你可以使用下述的 GDB 的宏调试命令 来查看相关的宏。

info macro – 你可以查看这个宏在哪些文件里被引用了，以及宏定义是什么样的。

macro – 你可以查看宏展开的样子。

三、源文件

这个问题问的也是很多的，太多的朋友都说找不到源文件。在这里我想提醒大家做下面的检查：

编译程序员是否加上了-g 参数以包含 debug 信息。 路径是否设置正确了。使用 GDB 的 directory 命令来设置源文件的目录。

下面给一个调试/bin/ls 的示例（ubuntu 下）

```
$ apt-get sourcecoreutils
$ sudoapt-get installcoreutils-dbgsym
$ gdb /bin/ls
GNU gdb (GDB) 7.1-ubuntu
(gdb) list main
1192       ls.c: No such fileor directory.
inls.c
(gdb) directory ~/src/coreutils-7.4/src/
Source directories searched:
/home/hchen/src/coreutils-7.4:$cdir:$cwd
(gdb) list main
1192               }
1193       }
1194
1195       int
1196       main (int argc, char **argv)
1197       {
1198           int i;
1199           struct pending *thispend;
1200           int n_files;
1201
```

四、条件断点

条件断点是语法是：break [where] if [condition]，这种断点真是非常管用。尤其是在一个循环或递归中，或是要监视某个变量。注意，这个设置是在 GDB 中的，只不过每经过那个断点时 GDB 会帮你检查一下条件是否满足。

五、命令行参数

有时候，我们需要调试的程序需要有命令行参数，很多朋友都不知道怎么设置调试的程序的命令行参数。其实，有两种方法：

gdb 命令行的 –args 参数 gdb 环境中 set args 命令。

六、gdb 的变量

有时候，在调试程序时，我们不单单只是查看运行时的变量，我们还可以直接设置程序中的变量，以模拟一些很难在测试中出现的情况，比较一些出错，或是 switch 的分支语句。使用 set 命令可以修改程序中的变量。

另外，你知道 gdb 中也可以有变量吗？就像 shell 一样，gdb 中的变量以$开头，比如你想打印一个数组中的个个元素，你可以这样：

```
1(gdb) set$i = 0
2
3(gdb) p a[$i++]
4
5...    #然后就一路回车下去了
```

当然，这里只是给一个示例，表示程序的变量和 gdb 的变量是可以交互的。

七、x 命令

也许，你很喜欢用 p 命令。所以，当你不知道变量名的时候，你可能会手足无措，因为 p 命令总是需要一个变量名的。x 命令是用来查看内存的，在 gdb 中 “help x” 你可以查看其帮助。

x/x 以十六进制输出 x/d 以十进制输出 x/c 以单字符输出 x/i 反汇编 – 通常，我们会使用 x/10i $ip-20 来查看当前的汇编（$ip 是指令寄存器）x/s 以字符串输出 八、command 命令

有一些朋友问我如何自动化调试。这里向大家介绍 command 命令，简单的理解一下，其就是把一组 gdb 的命令打包，有点像字处理软件的“宏”。下面是一个示例：

```
1 (gdb) breakfunc
2 Breakpoint 1 at 0x3475678: filetest.c, line 12.
3 (gdb) command1
4 Type commands forwhen breakpoint 1 is hit, one per line.
5 End with a line saying just "end".
6 >print arg1
7 >print arg2
8 >print arg3
9 >end
10(gdb)
```

当我们的断点到达时，自动执行 command 中的三个命令，把 func 的三个参数值打出来。

设置 core 环境

uname -a 查看机器参数

ulimit -a 查看默认参数

ulimit -c 1024 设置 core 文件大小为 1024

ulimit -c unlimit 设置 core 文件大小为无限

多线程如果 dump，多为段错误，一般都涉及内存非法读写。可以这样处理，使用下面的命令打开系统开关，让其可以在死掉的时候生成

core 文件。

ulimit -c unlimited

线程调试命令

(gdb)info threads

显示当前可调试的所有线程，每个线程会有一个 GDB 为其分配的 ID，后面操作线程的时候会用到这个 ID。

前面有*的是当前调试的线程。

(gdb)thread ID

切换当前调试的线程为指定 ID 的线程。

(gdb)thread apply ID1 ID2 command

让一个或者多个线程执行 GDB 命令 command。

(gdb)thread apply all command

让所有被调试线程执行 GDB 命令 command。

(gdb)set scheduler-locking off|on|step

估计是实际使用过多线程调试的人都可以发现，在使用 step 或者 continue 命令调试当前被调试线程的时候，其他线程也是同时执行的，怎么只让被调试程序执行呢？通过这个命令就可以实现这个需求。

off 不锁定任何线程，也就是所有线程都执行，这是默认值。

on 只有当前被调试程序会执行。

step 在单步的时候，除了 next 过一个函数的情况(熟悉情况的人可能知道，这其实是一个设置断点然后 continue 的行为)以外，只有当前线程会执行。

//显示线程堆栈信息

(gdb) bt

察看所有的调用栈

(gdb) f 3

调用框层次

(gdb) i locals

显示所有当前调用栈的所有变量