Module02-04
Linux 开发环境：使用 gdb

Linux 开发环境 - 使用 gdb

- vim
- 使用 gcc/g++
- make 和 makefile
- ➔ 使用 gdb
- CVS
- Eclipse CDT

使用 gdb

这次课我们将通过以下内容来熟悉 GNU Debugger （ gdb ）这一强大的代码调试工具：

- 一个简短的演示
- 执行 gdb 命令、获取命令帮助
- 随时查看所调试的程序代码
- 控制调试过程（中断或继续程序的运行）
- 查看或监控运行期的数据（如变量、栈信息等）
- 改变程序的执行
- 调试正在运行的程序
- 调试多线程程序

使用 gdb - 一个简短的演示

- 本次演示涉及的代码
  - 头文件

```
// shell_sort.h
#ifndef SHELL_SORT_
#define SHELL_SORT_

#include <vector>
#include <string>

void shell_sort(std::vector<int>& vec);

void print(std::vector<int>& vec, const std::string& hint);

#endif
```

使用 gdb - 一个简短的演示

- 本次演示涉及的代码（续）
  - 实现文件

```
// shell_sort.cpp

#include <iostream>

#include "shell_sort.h"

void shell_sort(std::vector<int>& vec) {
    const size_t n = vec.size();
    for (int gap = n / 2; 0 < gap; gap /= 2)
        for (int i = gap; i < n; ++i)
            for (int j = i - gap; 0 <= j; j -= gap)
                if (vec[j + gap] < vec[j])
                    std::swap(vec[j], vec[j + gap]);
}
```

使用 gdb - 一个简短的演示

- 本次演示涉及的代码（续）
  - 实现文件（续）

```
// shell_sort.cpp

void print(std::vector<int>& vec, const std::string& hint) {
    std::cout << hint;
    for (size_t i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << ' ';
    }
    std::cout << std::endl;
}
```

使用 gdb - 一个简短的演示

- 本次演示涉及的代码（续）
  - 测试驱动

```
// sort_test.cpp

#include "shell_sort.h"

int main() {
    std::vector<int> vec;
    vec.push_back(90);
    vec.push_back(2);
    vec.push_back(5);
    vec.push_back(120);
    vec.push_back(8);

    print(vec, "original: ");
    shell_sort(vec);
    print(vec, "sorted: ");
}
```

使用 gdb - 一个简短的演示

- 编译代码

```
# 特别注意：用 -g 选项告诉编译器在目标代码中添加调试信息
$ g++ -g -Wall -o shell_sort shell_sort.cpp sort_test.cpp
```

- 简单调试

```
$ gdb shell_sort        #0 启动 gdb
GNU gdb (GDB) 7.0-ubuntu
......
(gdb) list 13,16        #1 列出当前文件的 13-16 行，list 可简写为 l
13      print(vec, "original: ");
14      shell_sort(vec);
15      print(vec, "sorted: ");
16  }
```

使用 gdb - 一个简短的演示

- 简单调试（续）

```
(gdb) l shell_sort.cpp:6,13      #2 列出该程序其它源文件的代码
6   void shell_sort(std::vector<int>& vec) {
7       const size_t n = vec.size();
8       for (int gap = n / 2; 0 < gap; gap /= 2)
9           for (int i = gap; i < n; ++i)
10              for (int j = i - gap; 0 <= j; j -= gap)
11                  if (vec[j + gap] < vec[j])
12                      std::swap(vec[j], vec[j + gap]);
13  }
(gdb) break 14                   #3 在当前文件的第 14 行设置断点
Breakpoint 1 at 0x8048c27: file shell_sort.cpp, line 14.
(gdb) break shell_sort.cpp:12    #4 在其它文件指定行设置断点
Breakpoint 2 at 0x8048b99: file shell_sort.cpp, line 12.
(gdb) b print                    #5 在函数 print() 处设置断点
Note: breakpoint 1 also set at pc 0x8048c27.
Breakpoint 3 at 0x8048c27: file shell_sort.cpp, line 16.
(gdb)
```

使用 gdb - 一个简短的演示

- 简单调试（续）

```
(gdb) info break          #6 查看断点信息
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048c27 in
print(std::vector<int, std::allocator<int> >&, std::string)
at shell_sort.cpp:14
    breakpoint already hit 1 time
2       breakpoint     keep y   0x08048b99 in
shell_sort(std::vector<int, std::allocator<int> >&) at
shell_sort.cpp:12
3       breakpoint     keep y   0x08048c27 in
print(std::vector<int, std::allocator<int> >&, std::string)
at shell_sort.cpp:16
    breakpoint already hit 1 time
(gdb) run                 #7 运行程序，run 可简写为 r
Starting program:
/home/kwarph/Training/Module02/context/gdb/shell_sort

Breakpoint 1, print (vec=..., hint=...) at
shell_sort.cpp:16
16      std::cout << hint;
```

使用 gdb - 一个简短的演示

- 简单调试（续）

```
(gdb) next              #8 执行下一句，next 可简写为 n
17      for (size_t i = 0; i < vec.size(); ++i) {
(gdb)                   #9 空回车表示执行上一次命令，大多数命令支持这个动作
18          std::cout << vec[i] << ' ';
(gdb) print i           #10 打印变量的值，print 可简写为 p
$2 = 0
(gdb) step              #11 单步执行，step 可简写为 s
std::vector<int, std::allocator<int> >::operator[]
(this=0xbffff374, __n=0)
    at /usr/include/c++/4.4/bits/stl_vector.h:611
611       { return *(this->_M_impl._M_start + __n); }
(gdb) s
print (vec=..., hint=...) at shell_sort.cpp:17
17      for (size_t i = 0; i < vec.size(); ++i) {
(gdb) display i         #12 自动打印变量 i 的值
1: i = 1
(gdb) n
17      for (size_t i = 0; i < vec.size(); ++i) {
1: i = 1      # 由于自动打印（display），只要程序暂停，都打印 i 的值
```

使用 gdb - 一个简短的演示

- 简单调试（续）

```
(gdb) continue          #13 跳过当前断点，继续运行（如果碰到下个断点，暂停）
Continuing.
original: 90 2 5 120 8

Breakpoint 2, shell_sort (vec=...) at shell_sort.cpp:12
12                          std::swap(vec[j], vec[j + gap]);
(gdb) c
......
(gdb)                   # 空回车，执行上一次命令 continue
Continuing.
sorted: 2 5 8 90 120

Program exited normally.
(gdb) quit              #14 退出 gdb， quit 可简写为 q
$
```

使用 gdb - 执行 gdb 命令

- 执行 gdb 命令的方式
  - gdb prog

    调试应用程序 prog
  - gdb prog core_file

    调试程序 prog 和由 prog 异常终止产生的 core dump 文件
  - gdb prog pid

    调试一个正在运行的程序， pid 是进程的 id
  - gdb

    进入 gdb 后执行 file prog，调试程序 prog

使用 gdb - 执行 gdb 命令

- 用 help 命令获取帮助

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping
the program
user-defined -- User-defined commands

......
```

使用 gdb - 执行 gdb 命令

- 用 help 命令获取帮助 (续)

```
(gdb) help break
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an
address.
If a line number is specified, break at start of code for
that line.
If a function is specified, break at start of code for that
function.
If an address is specified, break at that exact address.
With no LOCATION, uses current execution address of selected
stack frame.
This is useful for breaking on return to a stack frame.

THREADNUM is the number from "info threads".
CONDITION is a boolean expression.
......
```

使用 gdb - 查看程序代码

- 查看程序代码
  - 使用 list 命令

命令	说明
list	列出当前行前后共 10 行（默认）代码
list *n*	列出第 n 行前后共 10 行（默认）代码
list ,*n*	列出当前行到第 n 行前后共 10 行（默认）代码
list *n*,*m*	列出第 n 行到第 m 行前后共 10 行（默认）代码
list +*n* / list -*n*	列出当前行以上、以下 n 行代码
list *file*:*n*	列出文件 file 的第 n 行前后共 10 行（默认）代码
list **address*	列出指定地址的行代码
list *function*	列出函数 function 的代码
list *file*:*function*	列出文件 file 中 function 的代码
注：当前行是指程序运行中正在执行的代码所在的行	

使用 gdb - 查看程序代码

- 查看程序代码（续）
  - 使用 directory 命令指定程序的源文件路径
    - `directory dir1:dir2:...`

      设置源文件搜索路径为 `dir1:dir2:...`，多个目录用：（冒号）分隔
    - `directory`

      清除源文件搜索路径
    - `info directory`

      显示程序的源文件路径

使用 gdb - 查看程序代码

- 查看程序代码（续）
  - 查看代码或函数代码所在的内存地址

```
(gdb) info line sort_test.cpp:12        # 查看指定行的内存地址
Line 12 of "sort_test.cpp" starts at address 0x8048e73
<main+271>
   and ends at 0x8048e85 <main+289>.
(gdb) info line shell_sort.cpp:print    # 查看指定函数的内存地址
Line 14 of "shell_sort.cpp"
   starts at address 0x8048c21
<_Z5printRSt6vectorIiSaIiEERKSs>
   and ends at 0x8048c27 <_Z5printRSt6vectorIiSaIiEERKSs+6>.
```

使用 gdb - 查看程序代码

- 查看程序代码（续）
  - 反汇编：disassemble

```
(gdb) disassemble print
Dump of assembler code for function
_Z5printRSt6vectorIiSaIiEERKSs:
0x08048c21 <_Z5printRSt6vectorIiSaIiEERKSs+0>: push   %ebp
0x08048c22 <_Z5printRSt6vectorIiSaIiEERKSs+1>: mov    %esp,
%ebp
0x08048c24 <_Z5printRSt6vectorIiSaIiEERKSs+3>: sub
$0x28,%esp
0x08048c27 <_Z5printRSt6vectorIiSaIiEERKSs+6>: mov
0xc(%ebp),%eax
0x08048c2a <_Z5printRSt6vectorIiSaIiEERKSs+9>: mov
%eax,0x4(%esp)
0x08048c2e <_Z5printRSt6vectorIiSaIiEERKSs+13>:    movl
$0x804c080,(%esp)
......
```

使用 gdb - 查看程序代码

- 查看程序代码（续）
  - 反汇编：disassemble

```
(gdb) disassemble print
Dump of assembler code for function
_Z5printRSt6vectorIiSaIiEERKSs:
0x08048c21 <_Z5printRSt6vectorIiSaIiEERKSs+0>: push   %ebp
0x08048c22 <_Z5printRSt6vectorIiSaIiEERKSs+1>: mov    %esp,
%ebp
0x08048c24 <_Z5printRSt6vectorIiSaIiEERKSs+3>: sub
$0x28,%esp
0x08048c27 <_Z5printRSt6vectorIiSaIiEERKSs+6>: mov
0xc(%ebp),%eax
0x08048c2a <_Z5printRSt6vectorIiSaIiEERKSs+9>: mov
%eax,0x4(%esp)
0x08048c2e <_Z5printRSt6vectorIiSaIiEERKSs+13>:    movl
$0x804c080,(%esp)
......
```

使用 gdb - 中断程序的运行

- 中断程序的运行
  - 设置断点

命令	说明
break *function*	在函数 function 处设置断点
break *function(type,...)*	在原型为 function(type,...) 的函数处设置断点
break *class::function*	在类 class 的 function 函数处设置断点
break	在下一个指令处设置断点
break *n*	在第 n 行设置断点
break *-n* / break *+n*	在当前行以上、以下 n 行处设置断点
break **address*	在指定内存地址的行处设置断点
break *file:n*	在文件 file 的第 10 行设置断点
break *file:function*	在文件 file 中的函数 function 处设置断点
break expr if condition	符合条件处中断，可与上述各种表达式联合使用
info break / info break n	显示所有 / 第 n 个断点的信息

使用 gdb - 中断程序的运行

- 中断程序的运行（续）
  - 设置观察点（ watchpoints ，另外一种形式的中断点）

    观察点是监控指定的变量或表达式的变化，一旦监控的表达式或变量被访问或被改变，程序即暂停

    - `watch expr`

      当 expr 的值发生变化时，程序暂停
    - `rwatch expr`

      当 `expr` 被访问时，程序暂停
    - `awatch expr`

      当 `expr` 被改变或被访问时，程序暂停
    - `info watch`

      显示所有 `watchpoint`

使用 gdb - 中断程序的运行

- 中断程序的运行（续）
  - 清理或禁止中断点（ breakpoints 、 watchpoints ）
    - clear： 清理中断点（只处理 break，不能处理 watch ）
      - 不带参数表示清理当前行中断点
      - 带参数执行方式同 break 指令，如 clear 6 表示清除第 6 行的断点
    - delete： 删除中断点（针对断点号操作）
      - 不带参数表示删除所有断点
      - delete n： 删除断点号为 n 的断点
      - delete n-m： 删除断点号从 n 到 m 的所有断点
    - disable： 禁用中断点（执行方式同 delete ）
    - enable： 启用被禁用的断点（执行方式类似 disable ）

使用 gdb - 中断程序的运行

- 中断程序的运行（续）
  - 为中断点设置命令集

    也就是程序执行到某个中断点后执行命令集内的指令

```
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08048e07 in main at
sort_test.cpp:11
    breakpoint already hit 1 time
        print vec.size()
3       breakpoint     keep y   0x08048e9a in main at
sort_test.cpp:13
(gdb) commands 2     # 为第 2 个断点添加命令集
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>print vec.size()    # 这次只添加一个命令
>end                 #  end 结束命令输入
```

使用 gdb - 中断程序的运行

- 中断程序的运行（续）
  - 为中断点设置命令集（续）

```
(gdb) r
Starting program:
/home/kwarph/Training/Module02/context/gdb/shell_sort

Breakpoint 2, main () at sort_test.cpp:11
11      print(vec, "original: ");
$2 = 5
(gdb) commands 2    # 开始设置命令集
Type commands for when breakpoint 3 is hit, one per line.
End with a line saying just "end".
>end     # commands 与 end 之间未加任何命令，即是清除第 2 个断点下的所有命令
```

使用 gdb - 继续运行程序

- 继续运行程序
  - continue [ignore_n]：同 fp，或简写为 c
    - 无参数表示继续运行到下个中断点，如无中断点则运行到程序结束
    - 带参数 ignore_n 表示继续运行，且忽略随后 ignore_n 次中断点
  - next [n] 和 step [n]
    - 单步执行，无参数表示一句一句执行，带参数 n 表示执行 n 条语句
    - 二者的区别： step 遇到函数调用则进入被调用的函数，而 next 将函数调用当作普通语句，不进入被调用的函数。
    - next 可简写为 n， step 可简写为 s

使用 gdb - 查看程序运行期变量

- 查看程序运行期变量
  - 打印表达式或变量的值： print ， 可简写为 p
    - `print expr`：打印变量或表达式 `expr` 的值

使用 gdb - 查看程序运行期变量

- 查看程序运行期变量（续）
  - 实时打印、关闭实时打印

命令	说明
display expr	自动显示变量或表达式的值
display/fmt expr	按指定的格式自动显示变量或表达式的值
undisplay num1 ...	删除一个或多个自动显示的变量或表达式
delete display num1 ...	删除一个或多个自动显示的变量或表达式
disable display num1 ...	禁止自动显示一个或多个变量或表达式
enable display num1 ...	重新启用被禁止自动显示的一个或多个变量或表达式
info display	显示所有自动显示的变量或表达式

使用 gdb - 查看程序运行期变量

- 查看程序运行期变量（续）
  - 实时打印、关闭实时打印示例

```
(gdb) display i
1: i = 11179008
(gdb) display vec[i]
2: vec[i] = Disabling display 2 to avoid infinite recursion.
(int &) @0xaaf2040: <error reading variable>
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
2:   n  vec[i]
1:   y  I
(gdb) undisplay 2
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   n  i
```

使用 gdb - 查看程序运行期变量

- 查看程序运行期变量（续）
  - 查看栈信息： backtrace ， 简写为 bt

```
(gdb) bt          # 打印当前函数调用的所有 frame 信息
#0  shell_sort (vec=...) at shell_sort.cpp:8
#1  0x08048e7f in main () at sort_test.cpp:12
(gdb) bt full     # 打印当前函数调用的所有 frame 信息，且打印各函数局部变量
#0  shell_sort (vec=...) at shell_sort.cpp:8
        i = 2
        gap = 2
        n = 5
#1  0x08048e7f in main () at sort_test.cpp:12
        vec = {<std::_Vector_base<int, std::allocator<int> >>
= {
            _M_impl = {<std::allocator<int>> =
{<__gnu_cxx::new_allocator<int>> = {<No data fields>}, <No
data fields>}, _M_start = 0x804d040,
              _M_finish = 0x804d054,
              _M_end_of_storage = 0x804d060}}, <No data
fields>}
```

使用 gdb - 改变程序的运行

- 改变程序的运行
  - 设置变量
    - 如：`set var i = 3`，将 i 的值重新赋值为 3
  - 强制函数返回
    - `return`

      结束当前函数的执行，函数中为执行的语句忽略，函数直接返回
    - `return result`

      结束当前函数的执行，函数中为执行的语句忽略，函数直接返回且将 `result` 作为返回值
  - 强制调用函数
    - `call function`

      调用函数 `function`，打印函数返回值

使用 gdb - 调试多线程程序

- 调试多线程程序
  - 查看线程信息：注意带 * 号的为当前线程

```
(gdb) r
Starting program:
/home/kwarph/Training/Module02/context/gdb/thread_test
[Thread debugging using libthread_db enabled]
[New Thread 0xb7fe8b70 (LWP 2855)]
[Switching to Thread 0xb7fe8b70 (LWP 2855)]
......

(gdb) info threads    # 查看线程信息
[New Thread 0xb77e7b70 (LWP 2856)]
  3 Thread 0xb77e7b70 (LWP 2856)  0x007ba7d8 in clone ()
   from /lib/tls/i686/cmov/libc.so.6
* 2 Thread 0xb7fe8b70 (LWP 2855)  count (id=1) at
thread_test.cpp:9
  1 Thread 0xb7fe96d0 (LWP 2852)  0x007ba7d8 in clone ()
   from /lib/tls/i686/cmov/libc.so.6
```

使用 gdb - 调试多线程程序

- 调试多线程程序（续）
  - 切换线程： thread n

```
(gdb) thread 3      # 切换到线程 3
[Switching to thread 3 (Thread 0xb77e7b70 (LWP 2856))]#0
0x007ba7d8 in clone
    () from /lib/tls/i686/cmov/libc.so.6
(gdb) info threads
* 3 Thread 0xb77e7b70 (LWP 2856)  0x007ba7d8 in clone ()
   from /lib/tls/i686/cmov/libc.so.6
  2 Thread 0xb7fe8b70 (LWP 2855)  count (id=1) at
thread_test.cpp:9
  1 Thread 0xb7fe96d0 (LWP 2852)  0x007ba7d8 in clone ()
   from /lib/tls/i686/cmov/libc.so.6
```

  - 接下来可以对当前线程设置断点、等操作，同之前在单线程程序一样

使用 gdb - 调用 shell 命令

- 调用 shell 命令
  - shell： 切换到 shell
  - shell command： 临时执行 shell 命令 command

使用 gdb - 小结

- 软件开发阶段尽量使用 -g 选项编译代码，为 debug 做准备。
- 要习惯通过调试器来查找问题，而不是在代码中书写大量的打印语句
- 常用的 gdb 命令：
  - run、 break、 watch、 print、 display、 next、 step、 continue、 bt
  - info