Linux如何调试内存泄漏

程序喵大人 高性能服务器开发 2020-05-25

内存泄漏是指由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

我们平时开发过程中不可避免的会遇到内存泄漏问题，你是如何排查的呢？估计你是使用下面这几个工具吧?

- valgrind
- mtrace
- dmalloc
- ccmalloc
- memwatch
- debug_new

这里程序喵向大家推荐新的一个排查内存泄漏的工具：AddressSanitizer(ASan)，该工具为gcc自带，4.8以上版本都可以使用，支持Linux、OS、Android等多种平台，不止可以检测内存泄漏，它其实是一个内存错误检测工具，可以检测的问题有：

- 内存泄漏
- 堆栈和全局内存越界访问
- free后继续使用
- 局部内存被外层使用
- Initialization order bugs(中文不知道怎么翻译才好，后面有代码举例，重要)

使用方法直接看我下面的代码：

检测内存泄漏

内存泄漏代码：

```
#include <stdlib.h>

void func1() { malloc(7); }

void func2() { malloc(5); }

int main() {
   func1();
   func2();
   return 0;
}
```

编译and输出：

```
g++ -fsanitize=address -g test_leak.cc && ./a.out

=================================================================
==103==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 7 byte(s) in 1 object(s) allocated from:
   #0 0x7f95b231eb40 in __interceptor_malloc (/usr/lib/x86_64-linu
   #1 0x7f95b36007f7 in func1() /home/wangzhiqiang/test/test_leak.
   #2 0x7f95b3600814 in main /home/wangzhiqiang/test/test_leak.cc:
   #3 0x7f95b1e61b96 in __libc_start_main (/lib/x86_64-linux-gnu/l

Direct leak of 5 byte(s) in 1 object(s) allocated from:
   #0 0x7f95b231eb40 in __interceptor_malloc (/usr/lib/x86_64-linu
   #1 0x7f95b3600808 in func2() /home/wangzhiqiang/test/test_leak.
   #2 0x7f95b3600819 in main /home/wangzhiqiang/test/test_leak.cc:
   #3 0x7f95b1e61b96 in __libc_start_main (/lib/x86_64-linux-gnu/l

SUMMARY: AddressSanitizer: 12 byte(s) leaked in 2 allocation(s).
```

编译方式很简单，只需要添加-fsanitize=address -g就可以检测出具体产生内存泄漏的位置以及泄漏空间的大小。

检测堆栈内存越界访问

示例：

```
#include <iostream>

int main() {
   int *array = new int[100];
   array[0] = 0;
   int res = array[100];  // out of bounds
   delete[] array;
   return res;
}
```

编译and输出：

```
g++ -fsanitize=address -g test_leak.cc && ./a.out

=================================================================
==110==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x
READ of size 4 at 0x6140000001d0 thread T0
   #0 0x7f0e06400d2d in main /home/wangzhiqiang/test/test_leak.cc:
   #1 0x7f0e048d1b96 in __libc_start_main (/lib/x86_64-linux-gnu/l
   #2 0x7f0e06400bb9 in _start (/mnt/d/wzq/wzq/util/test/a.out+0xb

0x6140000001d0 is located 0 bytes to the right of 400-byte region
allocated by thread T0 here:
   #0 0x7f0e05120608 in operator new[](unsigned long) (/usr/lib/x8
   #1 0x7f0e06400cab in main /home/wangzhiqiang/test/test_leak.cc:
   #2 0x7f0e048d1b96 in __libc_start_main (/lib/x86_64-linux-gnu/l

```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/wangzhiqiang
Shadow bytes around the buggy address:
 0x0c287fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c287fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c287fff8000: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
 0x0c287fff8010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c287fff8020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c287fff8030: 00 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa
 0x0c287fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c287fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c287fff8060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c287fff8070: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c287fff8080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes
 Addressable:           00
 Partially addressable: 01 02 03 04 05 06 07
 Heap left redzone:       fa
 Freed heap region:       fd
 Stack left redzone:      f1
 Stack mid redzone:       f2
 Stack right redzone:     f3
 Stack after return:      f5
 Stack use after scope:   f8
 Global redzone:          f9
 Global init order:       f6
 Poisoned by user:        f7
 Container overflow:      fc
 Array cookie:            ac
 Intra object redzone:    bb
 ASan internal:           fe
 Left alloca redzone:     ca
 Right alloca redzone:    cb
==110==ABORTING
```

可以方便定位到堆栈内存越界访问的错误。

全局内存越界访问：

示例：

```
#include <iostream>

int global_array[100] = {0};

int main() {
   int res = global_array[100];  // out of bounds
   return 0;
}
```

编译and输出：

```
g++ -fsanitize=address -g test_leak.cc && ./a.out
=================================================================
==116==ERROR: AddressSanitizer: global-buffer-overflow on address (
READ of size 4 at 0x7f42e6e02310 thread T0
   #0 0x7f42e6c00c83 in main /home/wangzhiqiang/test/test_leak.cc:(
   #1 0x7f42e50d1b96 in __libc_start_main (/lib/x86_64-linux-gnu/l
   #2 0x7f42e6c00b69 in _start (/mnt/d/wzq/wzq/util/test/a.out+0xb(

0x7f42e6e02310 is located 0 bytes to the right of global variable
SUMMARY: AddressSanitizer: global-buffer-overflow /home/wangzhiqia
Shadow bytes around the buggy address:
 0x0fe8dcdb8410: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb8420: 00 00 00 00 00 00 00 00 01 f9 f9 f9 f9 f9 f9 f9
 0x0fe8dcdb8430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb8440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb8450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0fe8dcdb8460: 00 00[f9]f9 f9 f9 f9 f9 00 00 00 00 00 00 00 00
```

```
 0x0fe8dcdb8470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb8480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb8490: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb84a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0fe8dcdb84b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes
 Addressable:           00
 Partially addressable: 01 02 03 04 05 06 07
 Heap left redzone:       fa
 Freed heap region:       fd
 Stack left redzone:      f1
 Stack mid redzone:       f2
 Stack right redzone:     f3
 Stack after return:      f5
 Stack use after scope:   f8
 Global redzone:          f9
 Global init order:       f6
 Poisoned by user:        f7
 Container overflow:      fc
 Array cookie:            ac
 Intra object redzone:    bb
 ASan internal:           fe
 Left alloca redzone:     ca
 Right alloca redzone:    cb
==116==ABORTING
```

局部内存被外层使用

示例：

```
#include <iostream>

volatile int *p = 0;

int main() {
```

```
{
   int x = 0;
   p = &x;
}
 *p = 5;
 return 0;
}
```

编译and输出：

```
g++ -fsanitize=address -g test_leak.cc && ./a.out
=================================================================
==243==ERROR: AddressSanitizer: stack-use-after-scope on address 0
WRITE of size 4 at 0x7fffce12a4b0 thread T0
  #0 0x7f3993e00e7d in main /home/wangzhiqiang/test/test_leak.cc:1
  #1 0x7f39922d1b96 in __libc_start_main (/lib/x86_64-linux-gnu/li
  #2 0x7f3993e00c89 in _start (/mnt/d/wzq/wzq/util/test/a.out+0xc8

Address 0x7fffce12a4b0 is located in stack of thread T0 at offset
  #0 0x7f3993e00d79 in main /home/wangzhiqiang/test/test_leak.cc:5

This frame has 1 object(s):
  [32, 36) 'x' <== Memory access at offset 32 is inside this varia
HINT: this may be a false positive if your program uses some custo
    (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-use-after-scope /home/wangzhiqian
Shadow bytes around the buggy address:
0x100079c1d440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x100079c1d490: 00 00 f1 f1 f1 f1[f8]f2 f2 f2 00 00 00 00 00 00
0x100079c1d4a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x100079c1d4b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d4c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d4d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100079c1d4e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes
Addressable:           00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone:       fa
Freed heap region:       fd
Stack left redzone:      f1
Stack mid redzone:       f2
Stack right redzone:     f3
Stack after return:      f5
Stack use after scope:   f8
Global redzone:          f9
Global init order:       f6
Poisoned by user:        f7
Container overflow:      fc
Array cookie:            ac
Intra object redzone:    bb
ASan internal:           fe
Left alloca redzone:     ca
Right alloca redzone:    cb
==243==ABORTING
```

free后被使用

示例：

```
#include <iostream>

int main() {
    int *array = new int[100];
    delete[] array;
```

```
    int a = array[0];  // error
    return 0;
}
```

编译and输出：

```
g++ -fsanitize=address -g test_leak.cc && ./a.out
=================================================================
==282==ERROR: AddressSanitizer: heap-use-after-free on address 0x6
READ of size 4 at 0x614000000040 thread T0
    #0 0x7f209fa00ca9 in main /home/wangzhiqiang/test/test_leak.cc
    #1 0x7f209ded1b96 in __libc_start_main (/lib/x86_64-linux-gnu/
    #2 0x7f209fa00b69 in _start (/mnt/d/wzq/wzq/util/test/a.out+0x

0x614000000040 is located 0 bytes inside of 400-byte region [0x614
freed by thread T0 here:
    #0 0x7f209e721480 in operator delete[](void*) (/usr/lib/x86_64
    #1 0x7f209fa00c72 in main /home/wangzhiqiang/test/test_leak.cc
    #2 0x7f209ded1b96 in __libc_start_main (/lib/x86_64-linux-gnu/

previously allocated by thread T0 here:
    #0 0x7f209e720608 in operator new[](unsigned long) (/usr/lib/x
    #1 0x7f209fa00c5b in main /home/wangzhiqiang/test/test_leak.cc
    #2 0x7f209ded1b96 in __libc_start_main (/lib/x86_64-linux-gnu/

SUMMARY: AddressSanitizer: heap-use-after-free /home/wangzhiqiang/
Shadow bytes around the buggy address:
  0x0c287fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c287fff8000: fa fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd fd
  0x0c287fff8010: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
```

```
  0x0c287fff8020: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff8030: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c287fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==282==ABORTING
```

Initialization order bugs

示例，这里有两个文件：

```
// test_memory1.cc
#include <stdio.h>

extern int extern_global;
int read_extern_global() { return extern_global; }

```

```
int x = read_extern_global() + 1;

int main() {
    printf("%d\n", x);
    return 0;
}
```

```
// test_memory2.cc

int foo() { return 123; }
int extern_global = foo();
```

第一种编译方式输出如下：

```
g++ test_memory1.cc test_memory2.cc && ./a.out
1
```

第二种编译方式输出如下：

```
g++ test_memory2.cc test_memory1.cc && ./a.out
124
```

这种问题我们平时编程过程中可以都不会太注意，然而通过ASan可以检测出这种潜在的bug：

编译and输出：

```
g++ -fsanitize=address -g test_memory1.cc test_memory2.cc

ASAN_OPTIONS=check_initialization_order=true:strict_init_order=tru
=================================================================
==419==ERROR: AddressSanitizer: initialization-order-fiasco on add
READ of size 4 at 0x7f46c20021a0 thread T0
    #0 0x7f46c1e00c27 in read_extern_global() /home/wangzhiqiang/t
```

```
    #1 0x7f46c1e00cb3 in __static_initialization_and_destruction_0
    #2 0x7f46c1e00d0a in _GLOBAL__sub_I__Z18read_extern_globalv /h
    #3 0x7f46c1e00e5c in __libc_csu_init (/mnt/d/wzq/wzq/util/test
    #4 0x7f46c0461b27 in __libc_start_main (/lib/x86_64-linux-gnu/
    #5 0x7f46c1e00b09 in _start (/mnt/d/wzq/wzq/util/test/a.out+0x

0x7f46c20021a0 is located 0 bytes inside of global variable 'exter
  registered at:
    #0 0x7f46c08764a8  (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x3
    #1 0x7f46c1e00e0b in _GLOBAL__sub_I_00099_1__Z3foov (/mnt/d/wz
    #2 0x7f46c1e00e5c in __libc_csu_init (/mnt/d/wzq/wzq/util/test

SUMMARY: AddressSanitizer: initialization-order-fiasco /home/wangz
Shadow bytes around the buggy address:
  0x0fe9583f83e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f83f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8400: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8410: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8420: 00 00 00 00 00 00 00 00 04 f9 f9 f9 f9 f9 f9 f9
=>0x0fe9583f8430: 00 00 00 00[f6]f6 f6 f6 f6 f6 f6 f6 00 00 00 00
  0x0fe9583f8440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe9583f8480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
```

```
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==419==ABORTING
```

注意：这里在运行程序前需要添加环境变量：

```
ASAN_OPTIONS=check_initialization_order=true:strict_init_order=true
```

小总结

ASan是个很好的检测内存问题的工具，不需要配置环境，使用还方便，编译时只需要-fsanitize=address -g就可以，运行程序时候可以选择添加对应的ASAN_OPTIONS环境变量就可以检测出很多内存问题。它的错误信息也很有用，明确指出当前是什么类型的内存错误，如：

- detected memory leaks
- heap-buffer-overflow
- stack-buffer-overflow
- global-buffer-overflow
- heap-use-after-free
- initialization-order-fiasco

具体可以看google的官方文档：https://github.com/google/sanitizers/wiki/AddressSanitizer

如果对后端开发感兴趣，想加入 高性能服务器开发微信交流群 进行交流，可以先加我微信 **easy_coder**，备注"加微信群"，我拉你入群，备注不对不加哦。

点【在看】是最大的支持