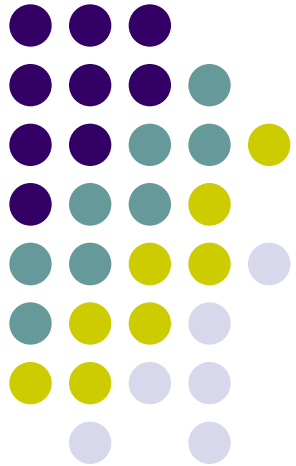
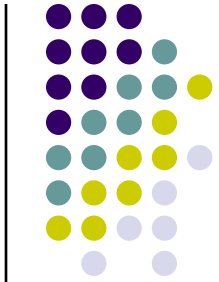
内存调试工具--Valgrind

jeremychen

jeremychen@tencent.com

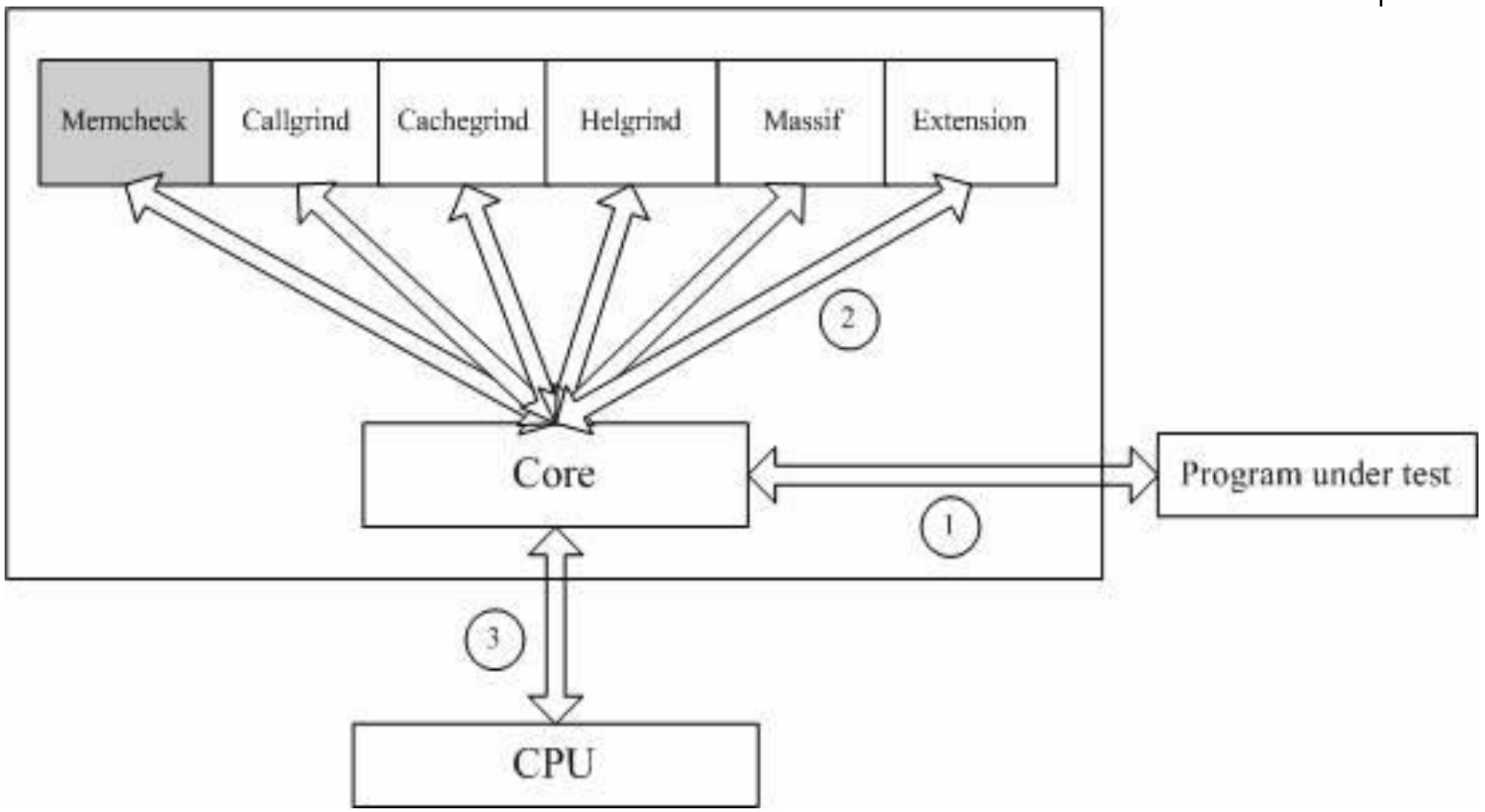


valgrind 的体系结构

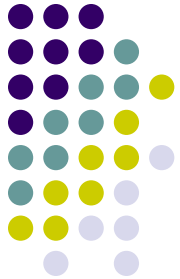


- Valgrind由内核（core）以及基于内核的其他调试工具组成
- 内核类似于一个框架（framework），它模拟了一个CPU环境，并提供服务给其他工具
- 其他工具则类似于插件 (plug-in)，利用内核提供的服务完成各种特定的内存调试任务

valgrind体系结构

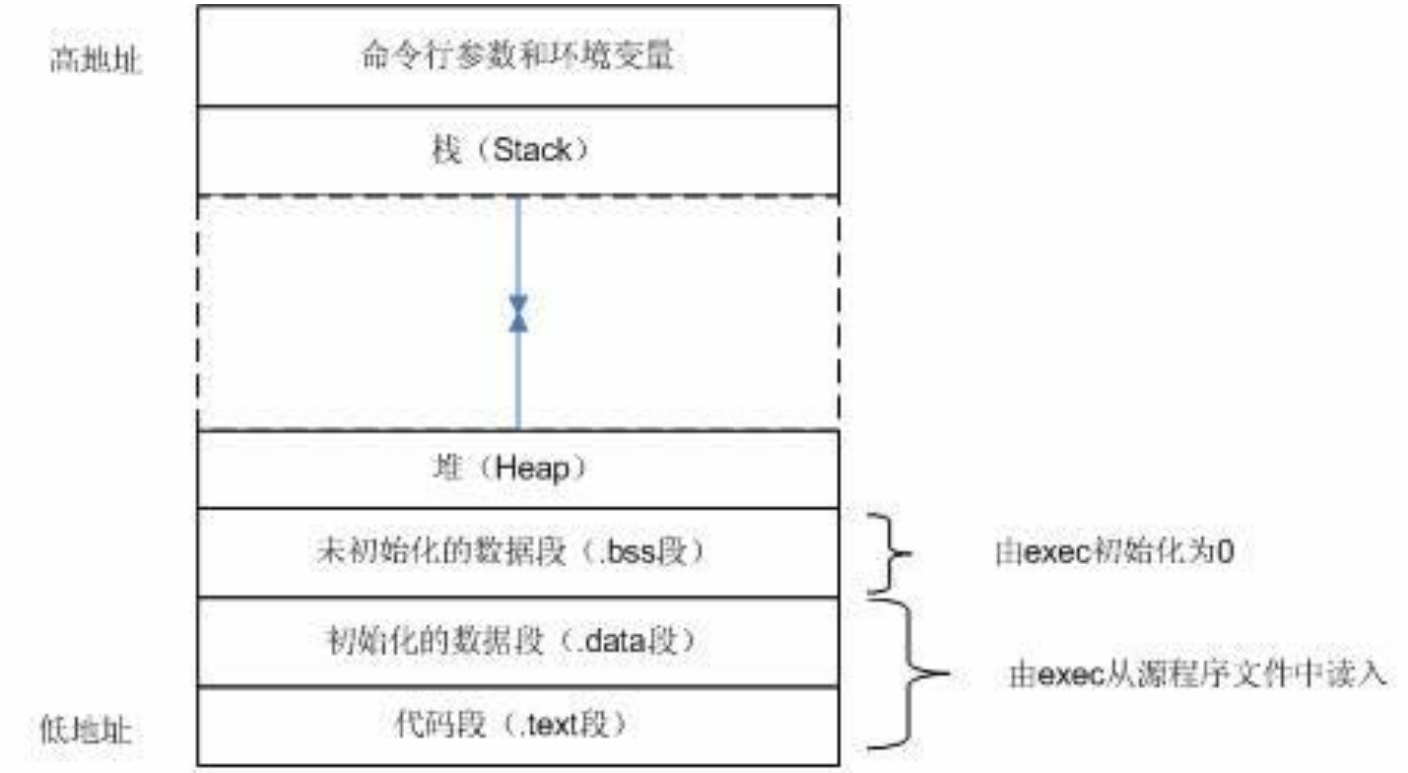


valgrind工具集



- Memcheck：这是valgrind应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况
- Callgrind：它主要用来检查程序中函数调用过程中出现的问题
- Cachegrind：它主要用来检查程序中缓存使用出现的问题。
- Helgrind：它主要用来检查多线程程序中出现的竞争问题
- Massif：它主要用来检查程序中堆栈使用中出现的问题
- Extension：可以利用core提供的功能，自己编写特定的内存调试工具

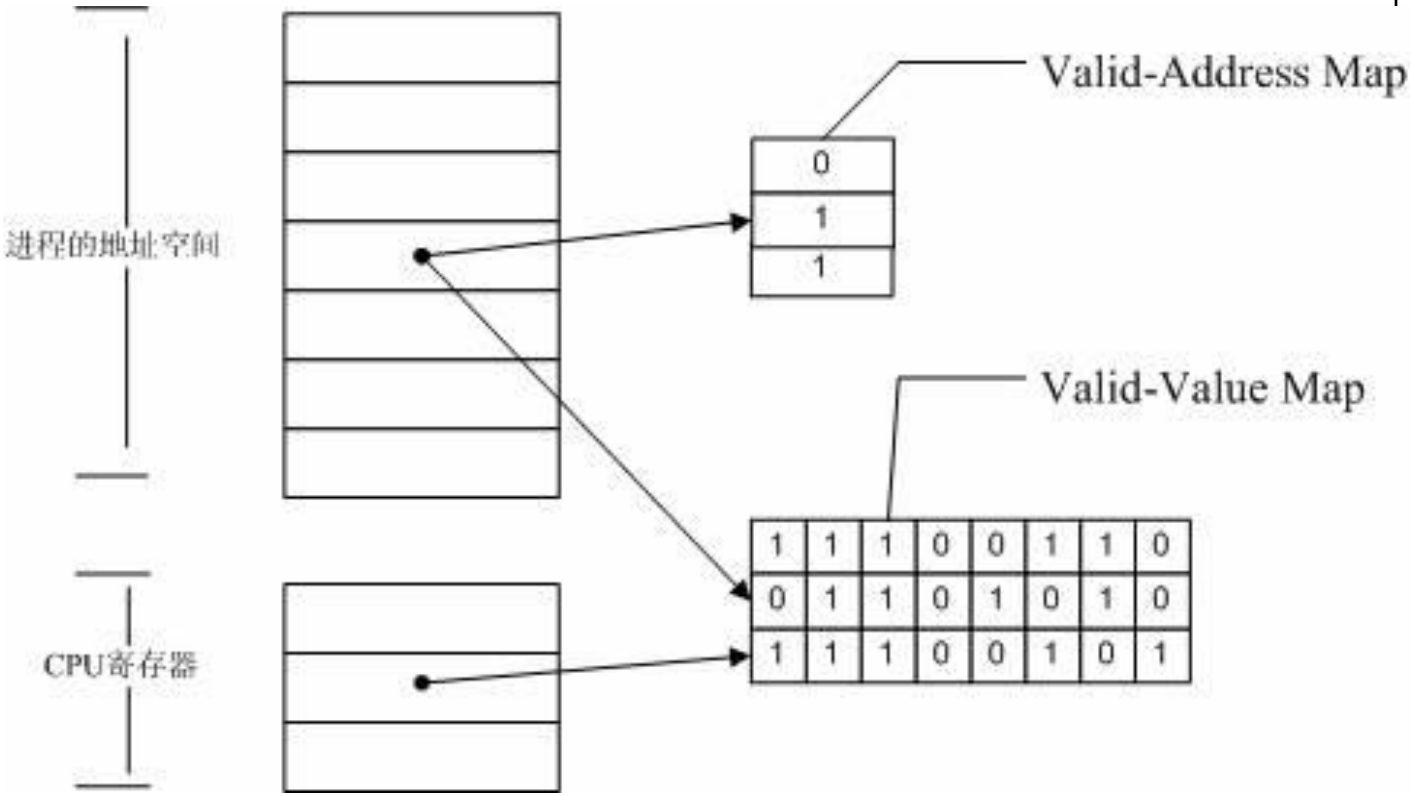
Linux程序的内存布局



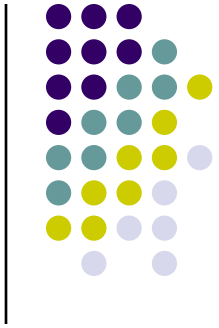
Linux程序的内存布局

- 代码段（.text）：存放的是CPU要执行的指令。代码段是可共享的，相同的代码在内存中只会有一个拷贝，同时这个段是只读的，防止程序由于错误而修改自身的指令。
- 初始化数据段（.data）：存放的是程序中需要明确赋初始值的变量，例如位于所有函数之外的全局变量：int val=100。需要强调的是，以上两段都是位于程序的可执行文件中，内核在调用exec函数启动该程序时从源程序文件中读入。
- 未初始化数据段（.bss）：位于这一段中的数据，内核在执行该程序前，将其初始化为0或者null。例如出现在任何函数之外的全局变量：int sum;
- 堆（Heap）：这个段用于在程序中进行动态内存申请，例如经常用到的malloc，new系列函数就是从这个段中申请内存。
- 栈（Stack）：函数中的局部变量以及在函数调用过程中产生的临时变量都保存在此段中。

内存检查原理


进程的地址空间
CPU寄存器
Valid-Address Map
0
1
1
Valid-Value Map
1 1 1 0 0 1 1 0
0 1 1 0 1 0 1 0
1 1 1 0 0 1 0 1

内存检查原理



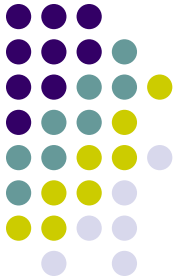
- Memcheck 能够检测出内存问题，关键在于其建立了两个全局表

- Valid-Value 表
  - 对于进程的整个地址空间中的每一个字节(byte)，都有与之对应的 8 个 bits；对于 CPU 的每个寄存器，也有一个与之对应的 bit 向量。这些 bits 负责记录该字节或者寄存器值是否具有有效的、已初始化的值
- Valid-Address 表
  - 对于进程整个地址空间中的每一个字节(byte)，还有与之对应的 1 个 bit，负责记录该地址是否能够被读写

检测原理

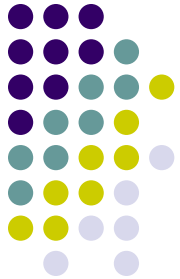
- 当要读写内存中某个字节时，首先检查这个字节对应的 A bit。如果该A bit显示该位置是无效位置，memcheck 则报告读写错误
- 内核（core）类似于一个虚拟的 CPU 环境，这样当内存中的某个字节被加载到真实的 CPU 中时，该字节对应的 V bit 也被加载到虚拟的 CPU 环境中。一旦寄存器中的值，被用来产生内存地址，或者该值能够影响程序输出，则 memcheck 会检查对应的V bits，如果该值尚未初始化，则会报告使用未初始化内存错误

valgrind使用简介



- 为了使valgrind发现的错误更精确，如能够定位到源代码行，建议在编译时加上-g参数，编译优化选项请选择O0，虽然这会降低程序的执行效率。
  - gcc –g –O0 sample.c –o sample
- 利用valgrind调试内存问题，不需要重新编译源程序，它的输入就是二进制的可执行程序。调用Valgrind的通用格式是
  - valgrind [valgrind-options] your-prog [your-prog-options]

valgrind使用简介

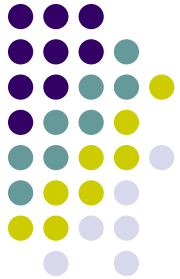


```
1   #include <stdlib.h>
2
3   void fun()
4   {
5           int *p=(int*)malloc(10*sizeof(int));
6           p[10]=0;
7   }
8
9   int main(int argc, char* argv[])
10  {
11          fun();
12          return 0;
13  }
```

valgrind使用简介

```
cdlyangj:/home/workspace/intro # valgrind ./sample
==32372== Memcheck, a memory error detector.
==32372== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==32372== Using LibVEX rev 1854, a library for dynamic binary translation.
==32372== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==32372== Using valgrind-3.3.1, a dynamic binary instrumentation framework.
==32372== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==32372== For more details, rerun with: -v
==32372==
==32372== Invalid write of size 4
==32372==    at 0x80483CF: fun (sample.c:6)
==32372==    by 0x80483EC: main (sample.c:11)
==32372==  Address 0x416f050 is 0 bytes after a block of size 40 alloc'd
==32372==    at 0x4023888: malloc (vg_replace_malloc.c:207)
==32372==    by 0x80483C5: fun (sample.c:5)
==32372==    by 0x80483EC: main (sample.c:11)
==32372==
==32372== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==32372== malloc/free: in use at exit: 40 bytes in 1 blocks.
==32372== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
==32372== For counts of detected errors, rerun with: -v
==32372== searching for pointers to 1 not-freed blocks.
==32372== checked 56,780 bytes.
==32372==
==32372== LEAK SUMMARY:
==32372==    definitely lost: 40 bytes in 1 blocks.
==32372==      possibly lost: 0 bytes in 0 blocks.
==32372==    still reachable: 0 bytes in 0 blocks.
==32372==         suppressed: 0 bytes in 0 blocks.
==32372== Rerun with --leak-check=full to see details of leaked memory.
```

valgrind使用简介



- 左边显示类似行号的数字（32372）表示的是 Process ID。
- 最上面的红色方框表示的是 valgrind 的版本信息。
- 中间的红色方框表示 valgrind 通过运行被测试程序，发现的内存问题。通过阅读这些信息，可以发现：
- 这是一个对内存的非法写操作，非法写操作的内存是4 bytes。
- 发生错误时的函数堆栈，以及具体的源代码行号。
- 非法写操作的具体地址空间。
- 最下面的红色方框是对发现的内存问题和内存泄露问题的总结。内存泄露的大小（40 bytes）也能够被检测出来。
- 示例程序显然有两个问题，一是fun函数中动态申请的堆内存没有释放；二是对堆内存的访问越界。这两个问题均被valgrind发现。

利用Memcheck检测内存问题

- 使用未初始化的内存
  - 数组a是局部变量，其初始值为随机值，而在初始化时并没有给其所有数组成员初始化，如此在接下来使用这个数组时就潜在有内存问题。

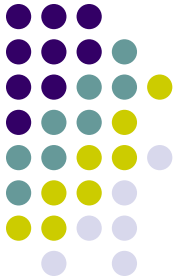
```
1  #include <stdio.h>
2
3  int main ( void )
4  {
5      int a[5];
6      int i, s;
7      a[0] = a[1] = a[3] = a[4] = 0;
8      s = 0;
9      for (i = 0; i < 5; i++)
10         s += a[i];
11     if (s == 377)
12         printf("sum is %d\n", s);
13     return 0;
14 }
```

利用Memcheck检测内存问题

- 输出结果显示，在该程序第11行中，程序的跳转依赖于一个未初始化的变量。准确的发现了上述程序中存在的问题

```
Conditional jump or move depends on uninitialised value(s)
   at 0x8048409: main (badloop.c:11)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
malloc/free: in use at exit: 0 bytes in 0 blocks.
malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
For counts of detected errors, rerun with: -v
All heap blocks were freed -- no leaks are possible.
```

利用Memcheck检测内存问题



- 内存读写越界
  - pt是一个局部数组变量，其大小为4，p初始指向pt数组的起始地址，但在对p循环叠加后，p超出了pt数组的范围，如果此时再对p进行写操作，那么后果将不可预期

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6          int len=4;
7          int* pt=(int*)malloc(len*sizeof(int));
8          int* p=pt;
9
10         for(int i=0;i<len;i++)
11         {
12                 p++;
13         }
14
15         *p=5;
16         printf("the value of p equal:%d",*p);
17         return 0;
18 }
```

利用Memcheck检测内存问题

- 输出结果显示，在该程序的第15行，进行了非法的写操作；在第16行，进行了非法读操作。准确地发现了上述问题。

```
==5064== Invalid write of size 4
==5064==    at 0x804850F: main (badacc.cpp:15)
==5064==  Address 0x4286038 is 0 bytes after a block of size 16 alloc'd
==5064==    at 0x4023888: malloc (vg_replace_malloc.c:207)
==5064==    by 0x80484E9: main (badacc.cpp:7)
==5064==
==5064== Invalid read of size 4
==5064==    at 0x8048518: main (badacc.cpp:16)
==5064==  Address 0x4286038 is 0 bytes after a block of size 16 alloc'd
==5064==    at 0x4023888: malloc (vg_replace_malloc.c:207)
==5064==    by 0x80484E9: main (badacc.cpp:7)
the value of p equal:5==5064==
==5064== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 1)
==5064== malloc/free: in use at exit: 16 bytes in 1 blocks.
==5064== malloc/free: 1 allocs, 0 frees, 16 bytes allocated.
==5064== For counts of detected errors, rerun with: -v
==5064== searching for pointers to 1 not-freed blocks.
==5064== checked 110,028 bytes.
```

利用Memcheck检测内存问题

- 内存覆盖
  - 下面就是一个 src 和 dst 发生重叠的例子。在 15 与 17 行中，src 和 dst 所指向的地址相差 20，但指定的拷贝长度却是 21，这样就会把之前的拷贝值覆盖。第 24 行程序类似，src(x+20) 与 dst(x) 所指向的地址相差 20，但 dst 的长度却为 21，这样也会发生内存覆盖。

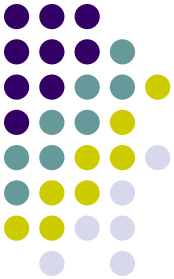
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[])
6  {
7          char x[50];
8          int i;
9          for(i=0;i<50;i++)
10         {
11                 x[i]=i+1;
12         }
13
14         strncpy(x+20, x, 20);     // ok
15         strncpy(x+20, x, 21);     // overlap
16         strncpy(x, x+20, 20);     // ok
17         strncpy(x, x+20, 21);     // overlap
18
19         x[39]='\0';
20         strcpy(x,x+20);           //ok
21
22         x[39]=39;
23         x[40]='\0';
24         strcpy(x,x+20);           //overlap
25
26         return 0;
27 }
```

利用Memcheck检测内存问题

- 输出结果显示上述程序中第15，17，24行，源地址和目标地址设置出现重叠。准确的发现了上述问题

```
==26612== Source and destination overlap in strncpy(0xBEBCC237, 0xBEBCC223, 21)
==26612==    at 0x4025C44: strncpy (mc_replace_strmem.c:291)
==26612==    by 0x804844E: main (badlap.c:15)
==26612==
==26612== Source and destination overlap in strncpy(0xBEBCC223, 0xBEBCC237, 21)
==26612==    at 0x4025C44: strncpy (mc_replace_strmem.c:291)
==26612==    by 0x8048488: main (badlap.c:17)
==26612==
==26612== Source and destination overlap in strcpy(0xBEBCC20E, 0xBEBCC222)
==26612==    at 0x4025D2E: strcpy (mc_replace_strmem.c:268)
==26612==    by 0x80484BE: main (badlap.c:24)
```

利用Memcheck检测内存问题



- 动态内存管理错误

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int i;
7      char* p = (char*)malloc(10);
8      char* pt=p;
9
10     for (i = 0; i < 10; i++)
11     {
12         p[i] = 'z';
13     }
14     delete p;
15
16     pt[1] = 'x';
17     free(pt);
18     return 0;
19 }
```

利用Memcheck检测内存问题

- 输出结果显示，第14行分配和释放函数不一致；第16行发生非法写操作，也就是往释放后的内存地址写值；第17行释放内存函数无效。准确地发现了上述三个问题。

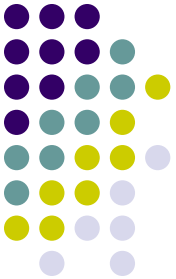
```
Mismatched free() / delete / delete []
   at 0x40230BC: operator delete(void*) (vg_replace_malloc.c:342)
   by 0x8048530: main (badmac.cpp:14)
 Address 0x4286028 is 0 bytes inside a block of size 10 alloc'd
   at 0x4023888: malloc (vg_replace_malloc.c:207)
   by 0x8048500: main (badmac.cpp:7)

Invalid write of size 1
   at 0x8048537: main (badmac.cpp:16)
 Address 0x4286029 is 1 bytes inside a block of size 10 free'd
   at 0x40230BC: operator delete(void*) (vg_replace_malloc.c:342)
   by 0x8048530: main (badmac.cpp:14)

Invalid free() / delete / delete[]
   at 0x402342C: free (vg_replace_malloc.c:323)
   by 0x8048544: main (badmac.cpp:17)
 Address 0x4286028 is 0 bytes inside a block of size 10 free'd
   at 0x40230BC: operator delete(void*) (vg_replace_malloc.c:342)
   by 0x8048530: main (badmac.cpp:14)
```

利用Memcheck检测内存问题

- 内存泄露
  - 两个部分的接口部分，一个函数申请内存，一个函数释放内存。并且这些函数由不同的人开发、使用，这样造成内存泄露的可能性就比较大了。这需要养成良好的单元测试习惯，将内存泄露消灭在初始阶段

```
1  #ifndef _BADLEAK_
2  #define  _BADLEAK_
3
4  typedef struct _node {
5          struct _node *l;
6          struct _node *r;
7          char v;
8  }node;
9
10 node *mk( node *l,  node *r, char val);
11 void nodefr(node * n);
12
13 #endif
```

利用Memcheck检测内存问题



```c
#include <stdlib.h>
#include "tree.h"

node *mk( node *l,  node *r, char val)
{
        node *f =(node *)malloc(sizeof(*f));
        f->l = l;
        f->r = r;
        f->v=val;
        return f;
}

void nodefr(node * n)
{
        if(n)
        {
                nodefr(n->l);
                nodefr(n->r);
                free(n);
        }
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

int main()
{
        node* tree1,*tree2,*tree3;

        tree1=mk(mk(mk(0,0,'3'),0,'2'),0,'1');

        tree2=mk(0,mk(0,mk(0,0,'6'),'5'),'4');

        tree3=mk(mk(tree1,tree2,'8'),0,'7');

        return 0;
}
```

利用Memcheck检测内存问题

- 该示例程序是生成一棵树的过程，每个树节点的大小为12（考虑内存对齐），共8个节点。从上述输出可以看出，所有的内存泄露都被发现。在上述的例子中，根节点是directly lost，而其他节点是indirectly lost

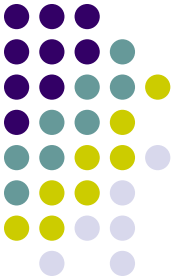
```
96 (12 direct, 84 indirect) bytes in 1 blocks are definitely lost in loss record 2 of
   at 0x4023888: malloc (vg_replace_malloc.c:207)
   by 0x804850F: mk(_node*, _node*, char) (tree.cpp:6)
   by 0x8048614: main (badleak.cpp:13)

LEAK SUMMARY:
   definitely lost: 12 bytes in 1 blocks.
   indirectly lost: 84 bytes in 7 blocks.
     possibly lost: 0 bytes in 0 blocks.
   still reachable: 0 bytes in 0 blocks.
        suppressed: 0 bytes in 0 blocks.
```

绑定调试器

- Valgrind –db-attach=yes –tool=memcheck ./test

```
==22237==
==22237== Invalid write of size 1
==22237==    at 0x8048382: main (test.c:5)
==22237==  Address 0x416B083 is not stack'd, malloc'd or (recently) free'd
==22237==
==22237== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ---- y
starting debugger
==22237== starting debugger with cmd: /usr/bin/gdb -nw /proc/22238/fd/4086
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you ar
welcome to change it and/or distribute copies of it under certain condition
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"...
Attaching to program: /proc/22238/fd/4086, process 22238
Reading symbols from /usr/local/lib/valgrind/x86-linux/vgpreload_core.so...
Loaded symbols for /usr/local/lib/valgrind/x86-linux/vgpreload_core.so
Reading symbols from /usr/local/lib/valgrind/x86-linux/vgpreload_memcheck.s
Loaded symbols for /usr/local/lib/valgrind/x86-linux/vgpreload_memcheck.so
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
0x08048382 in main () at test.c:5
5               x[91] = 'a';
(gdb)
```

Valgrind不能检查的错误



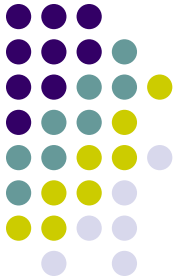
- Valgrind不对静态数组（分配在栈上）进行边界检查

```
int main()
{
    char x[10];
    x[11] = 'a';
}
```

- 输出结果：

```
==22198== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==22198== malloc/free: in use at exit: 0 bytes in 0 blocks.
==22198== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==22198== For counts of detected errors, rerun with: -v
==22198== All heap blocks were freed -- no leaks are possible.
```

其他工具功能



- Helgrind
  - 主要用来检查多线程程序中出现的竞争问题。Helgrind寻找内存中被多个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步的地方，而且会导致难以发掘的错误。Helgrind实现了名为“Eraser”的竞争检测算法，并做了进一步改进，减少了报告错误的次数。不过，Helgrind仍然处于实验阶段。

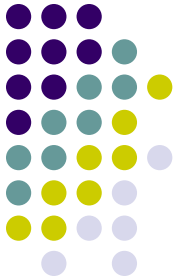
其他工具功能

- Callgrind
  - 和gprof类似的分析工具，但它对程序的运行观察更是入微，能给我们提供更多的信息。和gprof不同，它不需要在编译源代码时附加特殊选项，但加上调试选项是推荐的。Callgrind收集程序运行时的一些数据，建立函数调用关系图，还可以有选择地进行cache模拟。在运行结束时，它会把分析数据写入一个文件。callgrind_annotate可以把这个文件的内容转化成可读的形式。

其他工具功能

```
#include <stdlib.h>
#include <stdio.h>
void f(void)
{

    int* x = malloc(10 * sizeof(int));
    x[10] = 0; // problem 1: heap block overrun
} // problem 2: memory leak -- x not freed

int main(void)
{
    int i;
    f();
    printf("i=%d\n",i); //problem 3: use uninitialised value.
    return 0;
}
```

其他工具功能



```
[root@vss227 mem]# valgrind --tool=callgrind ./test
==22316== Callgrind, a call-graph generating cache profiler.
==22316== Copyright (C) 2002-2006, and GNU GPL'd, by Josef Weidendorfer et al
==22316== Using LibVEX rev 1658, a library for dynamic binary translation.
==22316== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==22316== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==22316== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==22316== For more details, rerun with: -v
==22316==
==22316== For interactive control, run 'callgrind_control -h'.
i=67195744
==22316==
==22316== Events    : Ir
==22316== Collected : 114367
==22316==
==22316== I   refs:      114,367
```

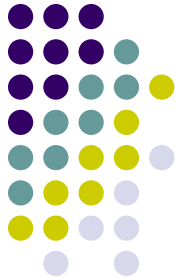
```
[root@vss227 mem]# callgrind_annotate callgrind.out.22316
Possible precedence problem on bitwise & operator at /usr/local/bin/ca
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.22316' (creator: callgrind-3.2.1)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
L2 cache:
Timerange: Basic block 0 - 21668
Trigger: Program termination
Profiled target:  ./test (PID 22316, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
     Ir
--------------------------------------------------------------------------------
114,367  PROGRAM TOTALS

--------------------------------------------------------------------------------
    Ir  file:function
--------------------------------------------------------------------------------
26,654  ???:do_lookup_versioned [/lib/ld-2.3.2.so]
22,844  ???:_dl_relocate_object [/lib/ld-2.3.2.so]
18,249  ???:_dl_lookup_versioned_symbol [/lib/ld-2.3.2.so]
13,968  ???:strcmp [/lib/ld-2.3.2.so]
 4,076  ???:do_lookup [/lib/ld-2.3.2.so]
 3,568  ???:_dl_lookup_symbol [/lib/ld-2.3.2.so]
```

其他工具功能

- Cachegrind
  - Cache分析器，它模拟CPU中的一级缓存I1，Dl和二级缓存，能够精确地指出程序中cache的丢失和命中。如果需要，它还能够为我们提供cache丢失次数，内存引用次数，以及每行代码，每个函数，每个模块，整个程序产生的指令数。这对优化程序有很大的帮助。

其他工具功能

- Massif
  - 堆栈分析器，它能测量程序在堆栈中使用了多少内存，告诉我们堆块，堆管理块和栈的大小。Massif能帮助我们减少内存的使用，在带有虚拟内存的现代系统中，它还能够加速我们程序的运行，减少程序停留在交换区中的几率。



- Thank You !