Linux应用程序调试技术

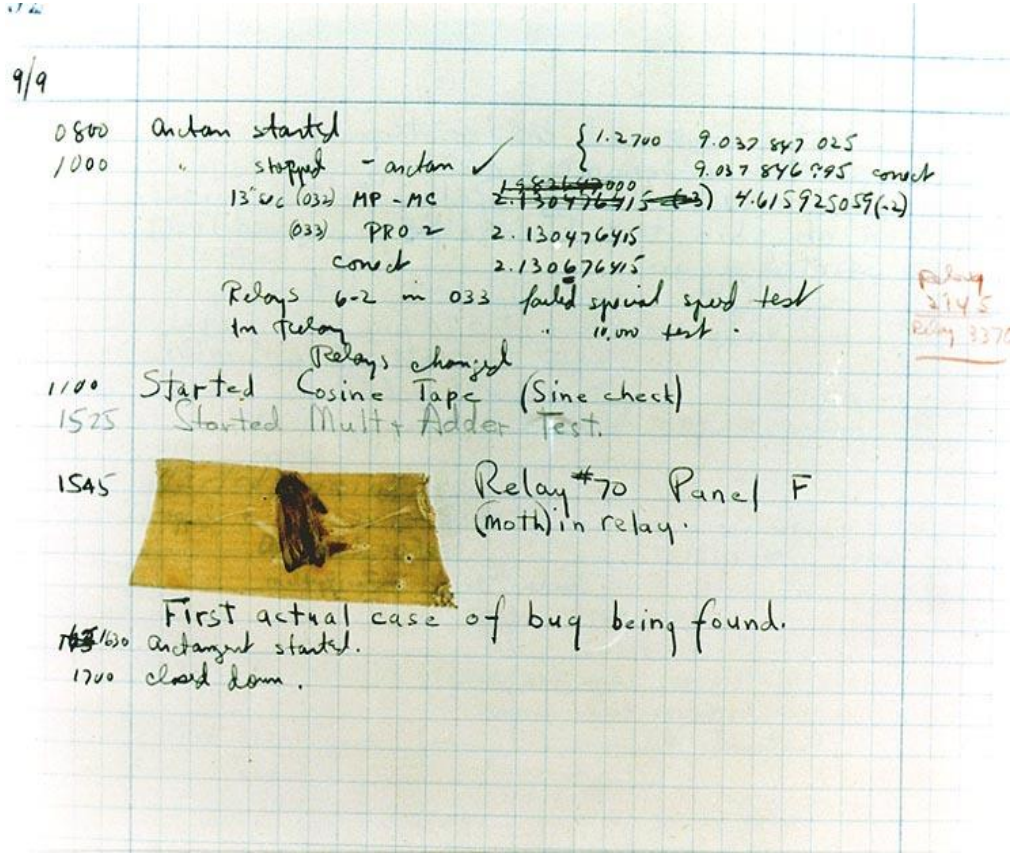
郑立松

Albert Zheng <lisong.zheng@gmail.com>

2011年12月

“Bug”的由来

Grace Hopper (1906~1992)，计算机科学先驱，美国海军准将。COBOL语言的设计者。
她还有一个重要的荣誉，就是1945年9月9日抓住了计算机历史上第一个bug。这只真正的臭虫，如今还荣幸地存放在计算机档案馆里供人瞻仰。



本课程包含的内容

- 以实际遇到的bugs为例，介绍调试的基本思路和方法；
- 涉及GDB等调试器的使用方法；
- 转储文件（Core Dump）文件的查看方法；
- 发生SIGSEGV时，应用程序异常停止时的调试；
- 发生死循环或CPU负载高时的调试(OProfile调试工具)；
- Valgrind进行各种内存非法使用的检测。

发现Bug

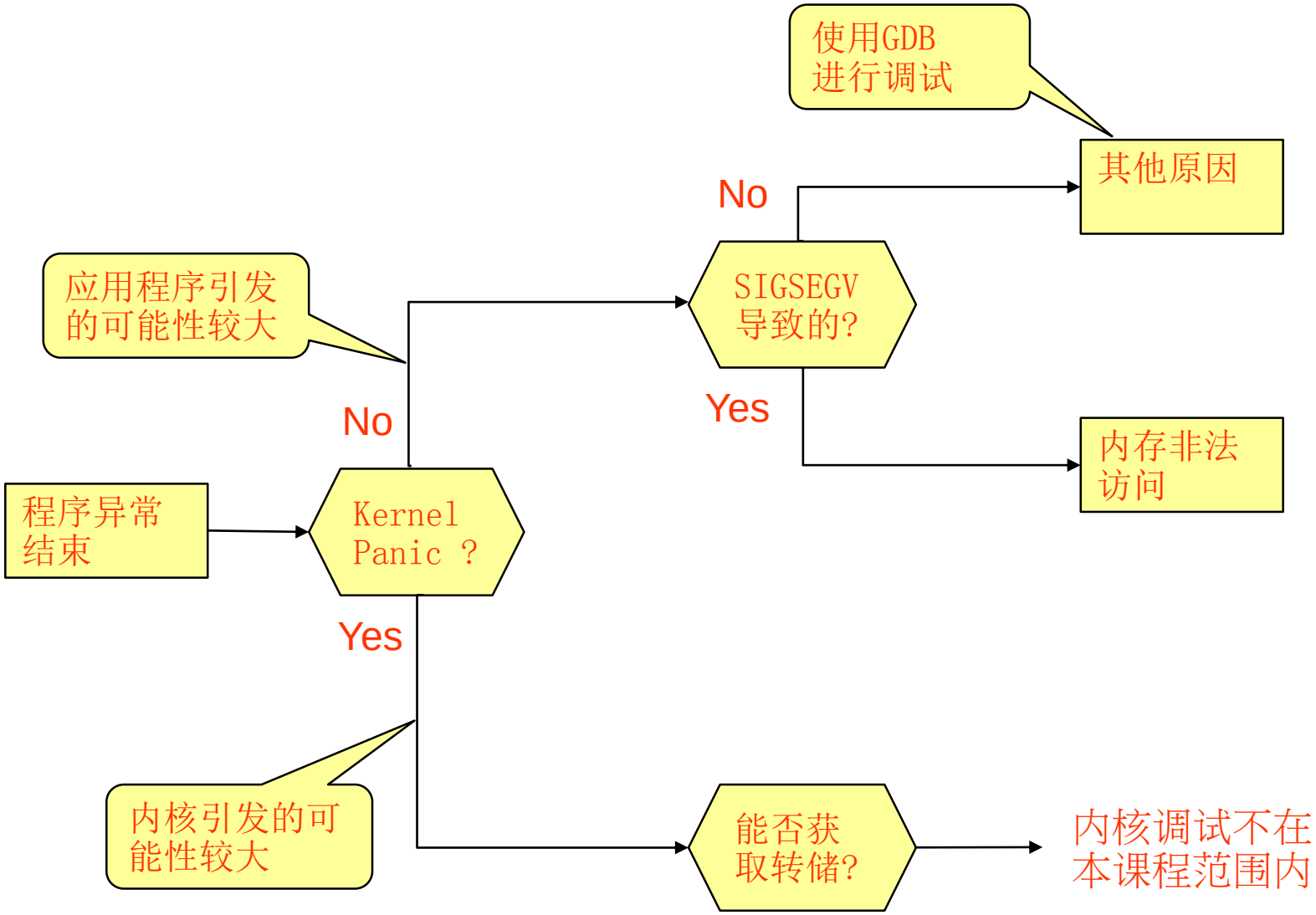
- Bug的发现形式多种多样：
  - 主要会在测试阶段发现：程序行为与期待行为（规格specification）不一致，就称为bug。
  - 也会在平常随意使用时发现
  - 或被自己以外其他人发现

- 无论哪种情况，**调试**总是开始于bug被确认之后。

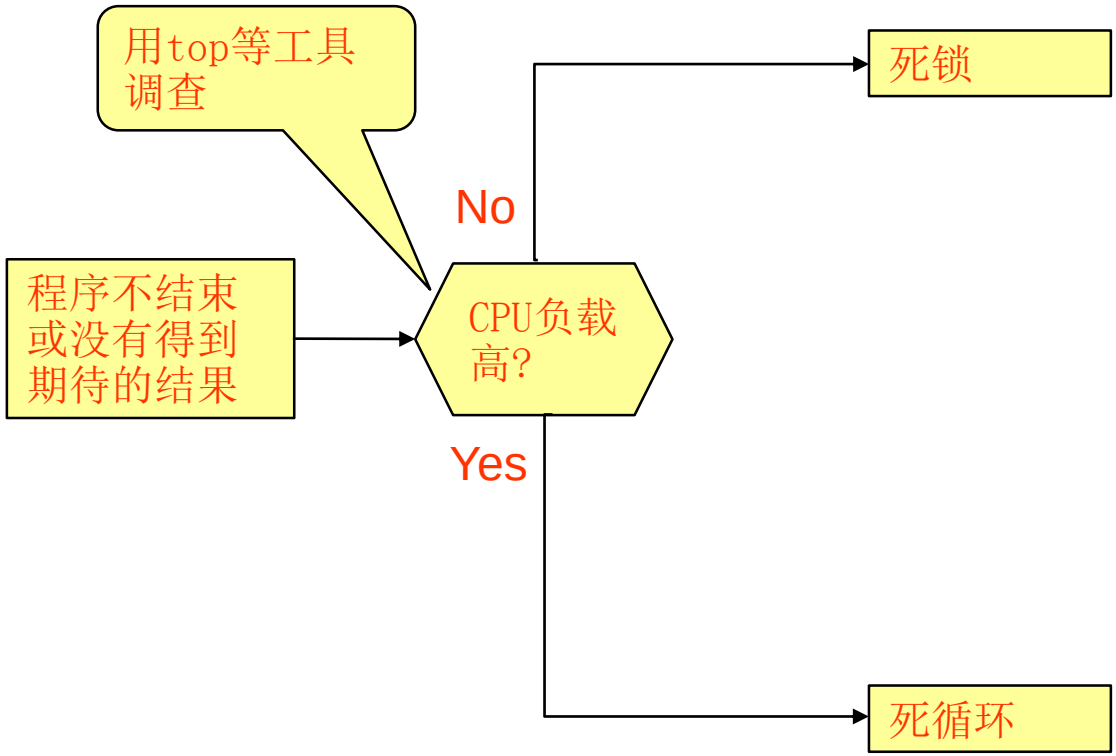
Bug的分类

- 如果把编程简单地看作是提供输入、并获得输出的过程，那么程序的行为可以分为以下3种：
  1. 执行期待的行为并结束。
  2. 不执行期待的行为并结束。
  3. 不结束。

调试的指南图 – 程序异常结束时



调试的指南图 – 程序不结束时



内核有问题的现象

区分方法	结果
ps	显示中途停止，状态为D
ping	不返回响应
键盘	键盘无法输入
kill -9	无法结束进程
strace	无法attach到进程（无响应）
gdb	无法attach到进程（无响应）
查看内核信息	soft lockup等有输出结果

调试前的必知必会 – 获得进程的内核转储

- 获得内核转储(core dump)的最大好处是：它能保存问题发生时的状态。只要有问题发生时程序的可执行文件和内核转储，就可以知道进程当时的状态。

  $ulimit –c

  0

- -c选项表示显示或设置core文件的最大值。上例中限制为0，表示内核转储功能无效。
- 开启内核转储：

  $ulimit –c unlimited

调试前的必知必会 – 设置内核转储的目录

- 一般会开辟一个内核转储专用分区，并在该分区中设置内核转储文件。
- 转储保持位置的完整路径可以通过sysctl变量kernel.core_pattern设置。

```
#cat /etc/sysctl.conf
kernel.core_pattern=/var/core/%t-%e-%p-%c.core
kernel.core_uses_pid=0  #设置为0，因为我们改变了文件名中PID的位置；如果为1，文件名末尾就会自动添加.PID
#sysctl –p
#./a.out
#ls /var/core
1323277483-a.out-8018-18446744073709551615.core
```

调试前的必知必会 – kernel.core_pattern

格式符	说明
%%	%字符本身
%p	被转储进程的进程ID(PID)
%u	被转储进程的真实用户ID
%g	被转储进程的真实组ID
%s	引发转储的信号编号
%t	转储时刻(从1970年1月1日0:00开始的秒数)
%h	主机名(同uname(2)返回的nodename)
%e	可执行文件名
%c	转储文件的大小上限(内核26.24以后可以使用)

调试前的必知必会 – 自动压缩内核转储文件

- kernel.core_pattern中可以加入管道符，启动用户模式辅助程序。管道符后面可以写程序名。
- 使用该方法可以自动压缩内核转储文件。

```
#vi /etc/sysctl.conf
kernel.core_pattern=|/usr/local/sbin/core_helper %t %e %p %c
kernel.core_uses_pid=0
```

core_helper脚本的内容：

```
#cat /usr/local/sbin/core_helper
#!/bin/bash
exec gzip - > /var/core/$1-$2-$3-$4.core.gz
```

调试前的必知必会 – 自动开启内核转储

- 开启登录到系统的所有用户的内核转储功能：

```
#vi /etc/profile
……
ulimit –S –c unlimited  >  /dev/null  2>$1
……
```

- 关闭：

```
ulimit –S –c 0 > /dev/null 2>$1
```

GDB的基本用法

- 调试的基本流程：
  1. 采用调试选项编译、构建调试对象。
  2. 启动调试器GDB。
     2.1.设置断点。
     2.2.显示栈帧。
     2.3.显示值。
     2.4.继续执行。

GDB的基本用法 – 准备

- 编译：

  #gcc –Wall –O2 **–g** 源文件

  注：关于调试时是否使用-O2优化选项进行编译，业界有两种不同的声音。

- 启动gdb

  #gdb 可执行文件

GDB的基本用法 – 设置断点

- 断点命令(break，可简写为b)的格式：

  break 断点

- 断点可以是函数名、行号、文件名+行号，文件名+函数名、偏移、地址：

  break 函数名

  break 行号

  break 文件名:行号

  break 文件名:函数名

  break +偏移量　　//现在暂停位置往后x行

  break –偏移量　　//现在暂停位置往前x行

  break *地址　　// e.g. *0x08186fd8

  break　　// 如果没指定位置，则在下一行代码上设置断点

GDB的基本用法 – 查询断点

- 设置好的断点可用 info break 确认：

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400498 in main at core1.c:3
2       breakpoint     keep y   0x0000000000400498 in main at core1.c:4
3       breakpoint     keep y   0x0000000000400498 in main at core1.c:5
```

SIGSEGV的调试

- 应用程序执行了非法访问内存等操作后，就会发生SIGSEGV异常而停止。
- SIGSEGV发生的情况包括：
  1. NULL指针访问
  2. 指针被破坏等原因导致的非法地址访问
  3. 栈溢出导致访问超过了已分配的地址空间

malloc()和free()发生故障

- 应用程序bug中，最常见的就是内存相关库函数的错误使用引起的bug，如，内存的双重释放、访问分配空间之外内存等bug。
- 现在的glibc中有个方便的调试标志，可以利用环境变量MALLOC_CHECK调试发现相关的内存bug。(注：最新的glibc已不再需要设置这个环境变量来激活内存bug检查功能)
  - #export MALLOC_CHECK=1
  - #.membug

```
*** glibc detected *** ./membug: double free or corruption (top): 0x0000000000ce5010 ***
======= Backtrace: =========
```

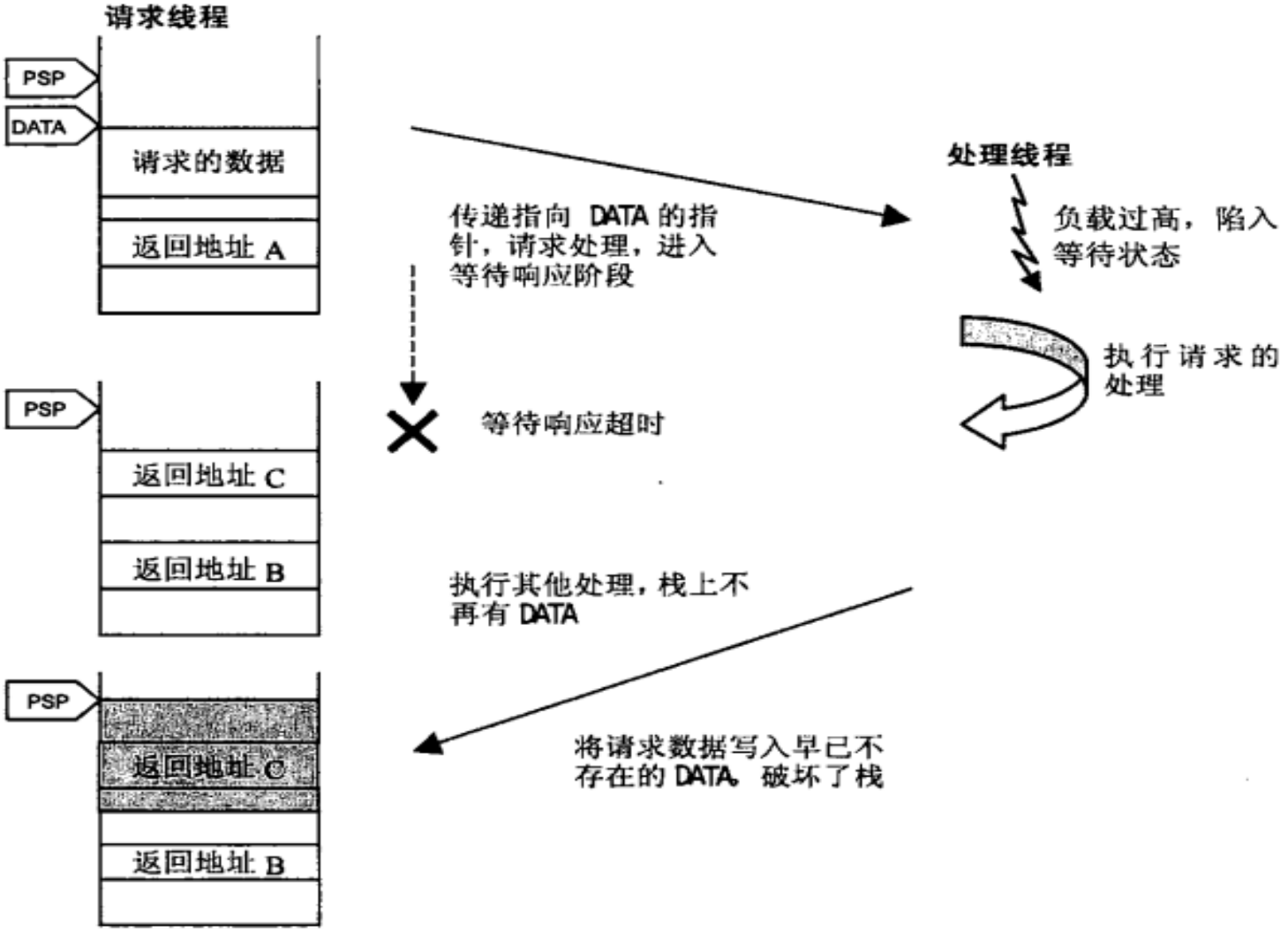
栈溢出 – 缓冲区溢出

- 可以怀疑是缓冲区溢出的情况之一就是：即使指定了-g编译选项，利用gdb读入core并显示backtrace后，栈帧中还是没有显示出符号表。
- 缓冲区溢出的常见情况：strcpy()越界；数组下标越界。
- 例如，如下显示一堆??()

```
(gdb) bt
#0  0x00000000004004fe in func () at bufov.c:11
#1  0x20656c626f6d6f74 in ?? ()
#2  0x0000007972727563 in ?? ()
#3  0x00007f38cfe05c4d in __libc_start_main () from /lib/libc.so.6
#4  0x0000000000400429 in _start ()
```

课后习题：多线程下的栈破坏

- 用后面课程即将讲授的pthread多线程编程，编写个类似生产者-消费者的demo程序，复原如下最棘手的栈破坏bug。



内存非法使用的调试 – Valgrind工具

- Valgrind能检测出内存的非法使用，对缓存、堆进行profile, 检测POSIX线程的冲突等。
- Valgrind的特征之一就是也是“非侵入式”，检测对象程序在编译时无须指定特别的选项，也不需要链接特别的函数库。
- 本教程介绍Valgrind最典型的用途 – 内存非法使用的检测。基本用法：

```
#valgrind --tool=memcheck --leak-check=yes program
```

- 检测内存非法使用(memcheck)是默认启动的选项，可以忽略不输入：

```
#valgrind --leak-check=yes program
```

Valgrind – 检测内存泄漏

#valgrind --leak-check=yes ./test1

```
int main(void)
{
   char *p = malloc(10);
   return EXIT_SUCCESS;
}
```

```
root@debian6:~/lecture/valgrind.work# valgrind --leak-check=yes ./test1
==12154== Memcheck, a memory error detector
==12154== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==12154== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==12154== Command: ./test1
==12154==
==12154==
==12154== HEAP SUMMARY:
==12154==     in use at exit: 10 bytes in 1 blocks
==12154==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==12154==
==12154== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12154==    at 0x4C244E8: malloc (vg_replace_malloc.c:236)
==12154==    by 0x4004F5: main (test1.c:6)
==12154==
==12154== LEAK SUMMARY:
==12154==    definitely lost: 10 bytes in 1 blocks
==12154==    indirectly lost: 0 bytes in 0 blocks
==12154==      possibly lost: 0 bytes in 0 blocks
==12154==    still reachable: 0 bytes in 0 blocks
==12154==         suppressed: 0 bytes in 0 blocks
==12154==
==12154== For counts of detected and suppressed errors, rerun with: -v
==12154== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Valgrind – 检测对非法内存地址的访问

#valgrind ./test2

```
int main(void)
{
   char *p = malloc(10);
   p[10] = 1;
   free(p);
   return EXIT_SUCCESS;
}
```

```
==12182== Command: ./test2
==12182==
==12182== Invalid write of size 1
==12182==    at 0x400552: main (test2.c:7)
==12182==  Address 0x518a04a is 0 bytes after a block of size 10 alloc'd
==12182==    at 0x4C244E8: malloc (vg_replace_malloc.c:236)
==12182==    by 0x400545: main (test2.c:6)
==12182==
==12182==
==12182== HEAP SUMMARY:
==12182==     in use at exit: 0 bytes in 0 blocks
==12182==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==12182==
==12182== All heap blocks were freed -- no leaks are possible
==12182==
==12182== For counts of detected and suppressed errors, rerun with: -v
==12182== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Valgrind – 检测读取未初始化区域

#valgrind --track-origins=yes ./test3

```
int main(void)
{
   char *x = malloc(sizeof(int));
   int a = *x + 1;
   free(x);
   return a;
}
```

```
==12209== Command: ./test3
==12209==
==12209== Syscall param exit_group(status) contains uninitialised byte(s)
==12209==    at 0x4EC9B18: _Exit (_exit.c:33)
==12209==    by 0x4E5F58D: __run_exit_handlers (exit.c:93)
==12209==    by 0x4E5F634: exit (exit.c:100)
==12209==    by 0x4E47C53: (below main) (libc-start.c:260)
==12209==  Uninitialised value was created by a heap allocation
==12209==    at 0x4C244E8: malloc (vg_replace_malloc.c:236)
==12209==    by 0x400545: main (test3.c:6)
==12209==
==12209==
==12209== HEAP SUMMARY:
==12209==     in use at exit: 0 bytes in 0 blocks
==12209==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==12209==
==12209== All heap blocks were freed -- no leaks are possible
==12209==
==12209== For counts of detected and suppressed errors, rerun with: -v
==12209== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Valgrind – 检测访问已释放的区域

#valgrind ./test4

```
int main(void)
{
   char *x = malloc(sizeof(int));
   free(x);
   int a = *x + 1;
   return a;
}
```

```
==12217== Command: ./test4
==12217==
==12217== Invalid read of size 1
==12217==    at 0x40055A: main (test4.c:8)
==12217==  Address 0x518a040 is 0 bytes inside a block of size 4 free'd
==12217==    at 0x4C240FD: free (vg_replace_malloc.c:366)
==12217==    by 0x400555: main (test4.c:7)
==12217==
==12217==
==12217== HEAP SUMMARY:
==12217==     in use at exit: 0 bytes in 0 blocks
==12217==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==12217==
==12217== All heap blocks were freed -- no leaks are possible
==12217==
==12217== For counts of detected and suppressed errors, rerun with: -v
==12217== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Valgrind – 检测内存双重释放

#valgrind ./test5

```
int main(void)
{
   char *x = malloc(sizeof(int));
   free(x);
   free(x);
   return EXIT_SUCCESS;
}
```

```
==12258== Command: ./test5
==12258==
==12258== Invalid free() / delete / delete[]
==12258==    at 0x4C240FD: free (vg_replace_malloc.c:366)
==12258==    by 0x400561: main (test5.c:8)
==12258==  Address 0x518a040 is 0 bytes inside a block of size 4 free'd
==12258==    at 0x4C240FD: free (vg_replace_malloc.c:366)
==12258==    by 0x400555: main (test5.c:7)
==12258==
==12258==
==12258== HEAP SUMMARY:
==12258==     in use at exit: 0 bytes in 0 blocks
==12258==   total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==12258==
==12258== All heap blocks were freed -- no leaks are possible
==12258==
==12258== For counts of detected and suppressed errors, rerun with: -v
==12258== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Valgrind – 检测非法栈操作

#valgrind ./test6

```
int main(void)
{
   int a;
   int* p = &a;
   p -= 0x20;
   *p = 1;   // 向比栈指针更低的地址写入数据
   return EXIT_SUCCESS;
}
```

```
==12279== Command: ./test6
==12279==
==12279== Invalid write of size 4
==12279==    at 0x4004A9: main (test6.c:9)
==12279==  Address 0x7ff000634 is just below the stack ptr.  To suppress, use: --workaround-gcc296-bugs=yes
==12279==
==12279==
==12279== HEAP SUMMARY:
==12279==     in use at exit: 0 bytes in 0 blocks
==12279==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==12279==
==12279== All heap blocks were freed -- no leaks are possible
==12279==
==12279== For counts of detected and suppressed errors, rerun with: -v
==12279== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Valgrind – 无法检测的错误

- Valgrind在检测非法使用内存方面效果非凡，但并不是万能的。例如，如下这种在栈上生成的内存区域的非法访问就无法检测到了。

#valgrind ./test_a

```
int main(void)
{
    char p[10];
    p[100] = 1;   // 这种错误检测不了
    return EXIT_SUCCESS;
}
```

停止响应时的解决方法

- 一般情况下，程序停止响应时要先确认是失去控制？或是在等待？用ps ax命令查看进程状态可确认。
  - 如果进程状态为R，表示进程仍在执行，失控的可能性很大。
  - 如果进程状态为S，表示在睡眠，如果睡眠时间很长了，大都情况下是陷入死锁了。

```
root@debian6:~/lecture/31_app_stall.work# ps ax -L|grep astall
 5332  5332 pts/0    Sl+    0:00 ./astall
 5332  5333 pts/0    Sl+    0:00 ./astall
 5342  5342 pts/1    S+     0:00 grep astall
```

- 接下来用gdb attach到这个astall进程上,调查哪里在睡眠。
- 使用gdb astall –x debug.cmd来剖析出到底是怎么死锁的。

死循环/CPU负载高时的解决方法

- OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.

- 非侵入式，无需改动代码，无需重新编译系统。
- 整个系统层面的Profile，All code is profiled。
- 利用硬件计数器。
- 低overhead。

OProfile准备

- 准备我们的程序：
  - 我们的程序,包括内核驱动都需要有符号信息:
  - 应用程序这样编译: gcc -g foo.c ...
  - 查看内核的导出的符号信息: cat /proc/kallsyms
- VMware下需采用timer interrupt for profiling模式：
  - #opcontrol --deinit
  - #modprobe oprofile timer=1
- OProfile可以观测的事件列表:
  - opcontrol --list-events
- 常用的事件有以下几种:
  - CPU_CLK_UNHALTED: CPU执行时间
  - LLC_MISSES: 末级Cache miss
  - DTLB_MISSES: 数据TLB miss

OProfile的用法

- 初始化和设置OProfile

  #opcontrol --init

  #opcontrol --setup --no-vmlinux -- event=default

  缺省事件是CPU_CLK_UNHALTED。在VMware下不能使用--event参数.

- 运行死循环测试程序

  #./foo abcd12345678910 &

- 启动OProfile

  #opcontrol --reset && opcontrol --start

  Using 2.6+ OProfile kernel interface.

  Using log file /var/lib/oprofile/samples/oprofiled.log

  Daemon started.

  Profiler running.

OProfile的用法 Cont.1

#opcontrol --status

Daemon running: pid 8238

Separate options: none

vmlinux file: none

Image filter: none

Call-graph depth: 0

#opcontrol –dump

此时马上可用opreport和opannotate查看此刻的剖析报告

- 继续运行一段时间

#opcontrol --stop

#opreport image:foo -l

CPU: CPU with timer interrupt, speed 0 MHz (estimated)

Profiling through timer interrupt

samples % symbol name

7271 96.8950 find_str

233 3.1050 main

OProfile的用法 Cont.2

#opannotate image:foo -s

```
 * CPU: CPU with timer interrupt, speed 0 MHz (estimated)
 * Profiling through timer interrupt
 */
/*
 * Total samples for file : "/root/lecture/oprofile/foo.c"
 *
 *    7504 100.000
 */


                :#include <string.h>
                :
                :const char* find_str(const char* s, int l)
   88  1.1727 :{ /* find_str total:    7271 96.8950 */
  137  1.8257 :  const char* e = s+l;
  819 10.9142 :  while(s < e)
                :  {
 4830 64.3657 :     if (*s == '<')
                :       return s;
 1081 14.4057 :     s++;
                :  }
  316  4.2111 :}
                :
                :int main(int argc, char* argv[])
                :{  /* main total:     233  3.1050 */
                :  char *s = argv[1];
                :  int i, l;
                :  if (argc ==1)
                :    return -1;
                :  l=strlen(s);
                :
                :  //for (i = 0; i < 100000000; i++)
                :  while (1)
  233  3.1050 :    find_str(s, l);
                :
```

其它优秀工具

- 本课程涉及的工具之外，还有许多优秀工具，例如：
  - strace、LTTng、dmalloc、blktrace、lockdep、kgdb、utrace、lockmeter、mpatrol、e1000_dump、git_bisect、kmemcheck等。