内存泄漏测试工具介绍

文件修订记录

日期	版本	内容	编写及修订者
2023.09.04	1.0	内存泄漏测试工具介绍	r.zhou & h.wang

一、meminfo.sh

1、介绍

该脚本用于监测内存的使用情况，具体的监测内容根据脚本文件中的内容变动。meminfo.sh的内容如下：

```
#!/bin/sh

while :
do
    cat /proc/meminfo
    free
    echo "------------------------------------------------------------------------
---------------"
    echo "                           Kbytes     PSS   Dirty    Swap"
    echo -n "Launcher"
    pmap -x $(ps aux | grep -w Launcher | awk '{print $1}' | head -1)
    echo "------------------------------------------------------------------------
---------------"
    /media/sda1/procrank
    sleep 3
done
```

- `cat /proc/meminfo`、`free`都显示的是Linux系统内存使用状况。
- `pmap -x $(ps aux | grep -w Launcher | awk '{print $1}' | head -1)`显示Launcher进程的内存映射情况。
- `/procrank`显示所有进程的内存使用情况，前面的`/media/sda1/`表示procrank文件存放的路径。

命令打印出的内容，在相关命令及参数分析进行具体展示。

2、使用方法

1. 将meminfo.sh、procrank文件放到U盘
2. 执行：`./路径/meminfo.sh &`，比如：`./media/sda1/meminfo.sh &`。

注意：

1. meminfo.sh、procrank文件可以放到其它路径，能找到就行，也可以放到/linux/sdk/prebuilt/common/usr/local/bin下，编译到bin里，在/usr/local/bin路径下能找到；
2. 如果执行命令后出现procrank not found，首先检查procrank的路径是否正确，该路径根据实际放置的位置变化，如果路径没问题，那么则考虑toolchain不对，不同的toolchain，对应的procrank文件不一样，需要更换。

   procrank路径：技术领域/系统分析/分析工具/tools

3、相关命令及参数分析

1. pmap -x pid

介绍：查看指定进程的内存映射情况。pmap 从文件 `/proc/<pid>/maps` 中获得相关数据，用来观察系统中的指定进程的地址空间分布和内存状态信息，包括进程各个段的大小。下面是使用该命令打印的Launcher进程的部分映射情况：

```
Address   Kbytes     PSS   Dirty    Swap  Mode  Mapping
00010000      16      16       0       0  r-xp  /tmp/sp/application/bin/Launcher
00023000       4       4       4       0  r-xp  /tmp/sp/application/bin/Launcher
00024000      84       4       0       0  rwxp  /tmp/sp/application/bin/Launcher
00039000    2080    1400    1400     380  rwxp  [heap]
41030000    3136    1184       0       0  r-xp
/tmp/sp/usr/local/qt/lib/libQt5Core.so.5.3.2
41340000      60       0       0       0  ---p
/tmp/sp/usr/local/qt/lib/libQt5Core.so.5.3.2
4134f000      28      20      12       0  r-xp
/tmp/sp/usr/local/qt/lib/libQt5Core.so.5.3.2
41356000      20      20      20       0  rwxp
/tmp/sp/usr/local/qt/lib/libQt5Core.so.5.3.2
```

- Address：内存映像的起始地址。
- Kbytes：内存映像的虚拟内存空间大小（KB）。
- PSS：内存映像实际驻留物理内存的空间大小（KB）。
- Dirty： 脏页的字节数（包括共享和私有的）（KB）。
- Swap：占用交换分区的字节数（KB）。
- Mode：内存权限：read、write、execute、shared、private（写时复制）
- Mapping：占用内存的文件、或[anon]（分配的内存）、或[stack]（栈）。

关注点：选取两个时间点的内存映射情况进行对比，如果较长时间内某个Mapping对应的Kbytes、PSS、 Dirty 、Swap的数值是增长的，则需要留意该Mapping映射位置的内存泄漏问题。

2. procrank

介绍：打印所有进程的内存使用情况，默认是按照PSS降序排列。下面是使用该命令的打印输出：

```
  PID       Vss      Rss      Pss      Uss     Swap  cmdline
  708   193568K   10396K    9722K    9580K    1600K  /application/bin/Launcher
  689   122572K    1152K     497K     324K     192K
/usr/local/bin/resourcemanager
25419     2052K     892K     423K     276K       0K  procrank-arm8
  809   108332K     912K     393K     340K     428K  /usr/local/bin/as_server
  692    50936K     848K     305K     252K     380K
/usr/local/bin/bluetooth_server
  690    33204K     776K     269K     216K     252K
/usr/local/bin/networkmanager
  954      904K     388K     206K      28K       0K  ash
  811    91248K     692K     185K     148K     340K
/usr/local/bin/device_server
  807    49592K     724K     183K     116K     236K  /usr/local/bin/pfc_server
  685      916K     364K     182K       4K      32K  /bin/sh
  767    87208K     588K     166K     124K     720K  /usr/local/bin/gocsdk
  813    58208K     644K     146K     116K     304K
/usr/local/bin/apple_iap_server
  944    56928K     588K     127K     100K     204K  /application/bin/mcusrv
```

```
  808    40516K     584K     124K     100K     188K  /usr/local/bin/log_server
 1016     9736K     464K     114K      96K      40K  ./myvt
  810    32532K     604K     111K      76K     188K
/usr/local/bin/videoin_server
  927    52728K     640K     104K      56K     504K  /usr/local/bin/ps_server
  857    53408K     492K      90K      64K     176K  /usr/local/bin/vs_server
  930    24588K     528K      83K      60K     240K
/usr/local/bin/discplayer_server
  730     4400K     452K      52K      32K     156K  sp_wd_usr
  751    11396K     516K      49K       4K     396K  /usr/local/bin/gocsdk
  693    32700K     532K      41K       8K     272K
/usr/local/bin/bt_init_client
  688     1516K     212K      32K      24K      44K
/usr/local/bin/servicemanager
    1     2132K     340K      19K       4K     260K  /init
  812     1900K     316K      18K       4K     200K  /usr/local/bin/mdnsd
                         ------   ------   ------  ------
                        13651K   12152K  7528448K  TOTAL

RAM: 64064K total, 15640K free, 64K buffers, 8544K cached, 4K shmem, 8084K slab
```

- VSS：Virtual Set Size，虚拟内存耗用内存，包含共享库占用的全部内存，以及分配但未使用内存。用处不大。
- RSS：Resident Set Size，实际使用物理内存，包含共享库占用的全部内存。RSS 易被误导的原因在于， 它包括了该进程所使用的所有共享库的全部内存大小。对于单个共享库，尽管无论多少个进程使用，实际该共享库只会被装入内存一次。所以RSS并不能准确反映单进程的内存占用情况。
- PSS：Proportional Set Size，实际使用的物理内存，比例分配共享库占用的内存，按照进程数等比例划分。**可供参考。**
- USS：Unique Set Size，进程独占的物理内存，不计算共享库。当一个进程被销毁后，USS是真实返回给系统的内存。**可供参考。**
- Swap：交换分区，物理内存不够用的时候，会将部分内存上的数据交换到swap空间上。

一般VSS >= RSS >= PSS >= USS。

关注点：如果在较长时间内，某个进程的PSS、USS或者Swap呈增长趋势，则说明该进程可能存在内存泄漏问题。

3. cat /proc/meminfo

介绍：Linux系统内存使用状况主要存储在/proc/meminfo中，”free”、”vmstat”等命令就是通过它获取数据的，所以参考/proc/meminfo中的数据更为准确。下面是使用该命令打印的内容：

```
MemTotal:          64064 kB     // 可供系统支配的内存（即物理内存减去一些预留位和内核的二进制代码大小）
MemFree:           15672 kB     // LowFree与HighFree的总和，系统中未使用的内存
MemAvailable:      21980 kB     // 应用程序可用内存，
MemAvailable≈MemFree+Buffers+Cached，估计值
Buffers:              64 kB     // 缓冲区内存数，作为buffer cache的内存，是块设备的读写缓冲区
Cached:             8544 kB     // 缓存区内存数，作为page cache的内存，文件系统的cache
SwapCached:          740 kB     // 交换文件中的已经被交换出来的内存。与 I/O 相关
Active:             8908 kB     // 经常（最近）被使用的内存
Inactive:           5380 kB     // 最近不常使用的内存。这很容易被系统移做他用
Active(anon):       3180 kB     // 活跃的匿名内存（进程中堆上分配的内存,是用malloc分配的内存）
Inactive(anon):     2504 kB     // 不活跃的匿名内存
```

```
Active(file):       5728 kB     // 活跃的与文件关联的内存（比如程序文件、数据文件所对应的内存页）
Inactive(file):     2876 kB     // 不活跃的与文件关联的内存
Unevictable:           0 kB     // 不能被释放的内存页
Mlocked:               0 kB     // 被系统调用mlock()锁定的内存大，不可回收
SwapTotal:         36860 kB     // 交换空间总大小
SwapFree:          28996 kB     // 空闲的交换空间大小
Dirty:                 0 kB     // 等待被写回到磁盘的大小
Writeback:             0 kB     // 正在被写回的大小
AnonPages:          5616 kB     // 未映射页的大小
Mapped:             7848 kB     // 设备和文件映射大小
Shmem:                 4 kB     // 已经被分配的共享内存大小，包括shared memory、tmpfs和devtmpfs
Slab:               8084 kB     // 内核数据结构缓存大小
SReclaimable:        952 kB     // 可收回Slab的大小
SUnreclaim:         7132 kB     // 不可收回的Slab的大小
KernelStack:        1736 kB     // kernel消耗的内存
PageTables:         1428 kB     // 管理内存分页的索引表的大小
NFS_Unstable:          0 kB     // 不稳定页表的大小
Bounce:                0 kB     // 在低端内存中分配一个临时buffer作为跳转，把位于高端内存的缓存数据复制到此处消耗的内存
WritebackTmp:          0 kB     // 用于临时写回缓冲区的内存
CommitLimit:       68892 kB     // 系统实际可分配内存总量
Committed_AS:     935208 kB     // 当前已分配的内存总量
VmallocTotal:     901120 kB     // 虚拟内存大小
VmallocUsed:           0 kB     // 已经被使用的虚拟内存大小
VmallocChunk:          0 kB     // 在vmalloc区域中可用的最大的连续内存块的大小
CmaTotal:          40960 kB     // 连续可用内存总数
CmaFree:               0 kB     // 空闲的连续可用内存
```

关注点：通常关注MemAvailable的数值，如果在较长时间内，MemAvailable的值呈减少趋势，则说明可能出现内存泄漏问题。

4. free

介绍：显示系统内存的使用情况，包括物理内存、交换内存(swap)和内核缓冲区内存。以下是使用该命令打印出的内容：

```
             total       used       free     shared    buffers     cached
Mem:         64064      48344      15720          4         64       8544
-/+ buffers/cache:      39736      24328
Swap:        36860       7864      28996
```

- total：可供系统支配的内存，/proc/meminfo中的MemTotal
- used：已经使用的物理内存，计算方式为 total - free，即/proc/meminfo中的MemTotal-MemFree
- free：未使用的物理内存，/proc/meminfo中的MemFree
- shared：tmpfs使用的内存，/proc/meminfo中的Shmem
- -buffers/cache：used - buffers - cached
- +buffers/cache：free + buffers + cached，可用的内存

关注点：可能需要关注各项参数的变化。

5. pstree

介绍：以树形结构显示程序和进程之间的关系。以下是使用pstree命令打印出的Launcher进程的信息：

```
Launcher-+-{BTCallbackThrea}
         |-{Binder_1}
         |-{Binder_2}
         |-7*[{Launcher}]
         |-2*[{LoadPlaylist}]
         |-{Notifier Worker}
         |-{PhotoDecode}
         |-{QDirectFbInput}
         |-2*[{QThread}]
         |-{RcNotifierWorke}
         `-{notifier listen}
```

关注点：将该语句放到meminfo.sh中，可以动态的观察进程的创建情况。如果在观察过程中，进程个数突然增加而后续又没有减少，或者进程个数持续在增加，那么则可能有内存泄漏问题。

6. cat /proc/slabinfo

介绍：该命令可以看到内核空间小块内存的申请情况，即slab分配的情况。下面为执行该命令部分输出情况：

```
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
: tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs>
<num_slabs> <sharedavail>
kmalloc-65536          3      3  65536    1   16 : tunables    8    4    0 :
slabdata      3      3      0
kmalloc-32768         49     49  32768    1    8 : tunables    8    4    0 :
slabdata     49     49      0
kmalloc-16384          4      4  16384    1    4 : tunables    8    4    0 :
slabdata      4      4      0
kmalloc-8192          11     11   8192    1    2 : tunables    8    4    0 :
slabdata     11     11      0
kmalloc-4096         163    163   4096    1    1 : tunables   24   12    0 :
slabdata    163    163      0
kmalloc-2048         172    172   2048    2    1 : tunables   24   12    0 :
slabdata     86     86      0
kmalloc-1024         239    240   1024    4    1 : tunables   54   27    0 :
slabdata     60     60      0
```

关注点：操作几次后，观察kmalloc-4096、kmalloc-8192的<active_objs>的数值变化，因为常创建4k、8k大小的空间，如果该数值在持续增大，则说明驱动上可能存在内存泄漏。

二、自动化测试工具myvt

1、介绍

录制需要反复执行的UI操作后，让工具自动执行操作，适用于需要反复点击的操作。

2、使用方法

1）./myvt -R

2）输入重复操作次数

3）操作

4）操作完成后，输入大写字母A，回车

按照上述流程执行后，平台会根据用户刚刚点击的步骤进行自动重复。缺点：该自动化测试工具运行后，无法在后台执行命令，需要kill才能停止。

3、问题记录

问题一

问题描述：如果执行上述的第四步后，显示You have recorded 0 records。

问题原因：可能是因为平台给QT的event不一样。

解决方法：

1）查看平台给QT的event类型

方法一：打开ininit.gui.qt.rc文件，找到下图划红线的地方，8368P是/dev/input/event2，8368u是/dev/input/event3。

```
#Setup env variable for QT
export QT_QPA_FONTDIR /application/fonts
export QT_QPA_EVDEV_KEYBOARD_PARAMETERS /dev/input/event1:/dev/input/event3:keymap=/application/keymap/keymap.data
export QT_QPA_GENERIC_PLUGINS evdevkeyboard
export QT_CUSTOMIZE_EVDEV_IR_PARAMETERS keymap=/application/keymap/Irkeymap.data
export QT_CUSTOMIZE_TOUCH_CONF /media/flash/nvm/touch_conf
```

方法二：在串口中输入`cat /proc/kmsg`然后点击一下屏幕会出现多条以下类似信息，说明是type3。

```
<7>evbug: Event. Dev: input3, Type: 3, Code: 57, Value: 0
<7>evbug: Event. Dev: input3, Type: 3, Code: 53, Value: 2070
```

2）修改event类型

在myvt的main.c中的main函数中，把下图中划红线的部分修改了重新编译即可。

```
353     fd = open("/dev/input/event2",O_RDWR);
354     if(fd<=0)
355     {
356         printf("%s\n",strerror(errno)) ;
357         return -1;
358     }
```

问题二

问题描述：执行命令后，出现myvt not found。

问题原因：toolchain不一样，或者文件的路径不对。

解决方法：先检查文件路径是否正确，文件是否存在，如无问题，则可能是toolchain不对，需要修改myvt的Makefile，重新编译生成myvt文件。

Makefile

```
CC=../../../build/tools/arm-9.2_eabihf/bin/arm-none-linux-gnueabihf-gcc

LIBPATH=-L../../../build/tools/arm-9.2_eabihf/arm-none-linux-gnueabihf/libc/usr/lib

```

三、mtrace

1、介绍

mtrace 工具的主要思路是在我们的调用内存分配和释放的函数中装载 “钩子（hook）” 函数，通过 “钩子（hook）” 函数打印的日志来帮助我们分析对内存的使用是否存在问题。 具体的做法是 mtrace() 函数中会为那些和动态内存分配有关的函数（譬如 malloc()、realloc()、memalign() 以及 free()）安装 “钩子（hook）” 函数，这些 hook 函数会为我们记录所有有关内存分配和释放的跟踪信息，而 muntrace() 则会卸载相应的 hook 函数。基于这些 hook 函数生成的调试跟踪信息，我们就可以分析是否存在 “内存泄漏” 这类问题。

2、使用

使用到的相关函数

```
#include <mcheck.h>

void mtrace(void);          //开启内存分配跟踪;
void muntrace(void);        //取消内存分配跟踪;
```

设置日志生成路径

有如下两种方法:

```
1.在代码里设置，"/media/sda1/mtrace.log"为生成log的位置
setenv("MALLOC_TRACE", "/media/sda1/mtrace.log", 1);

2.在串口设置环境变量
export MALLOC_TRACE=/media/sda1/mtrace.log
```

编译

```
gcc -g xxxx.c -o xxx
eg.gcc -g test.c -o test
```

使用举例

```
#include <mcheck.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    setenv("MALLOC_TRACE", "/media/sda1/mtrace.log", 1);
    mtrace();       // 开始跟踪

    char *p = (char *)malloc(100);
```

```
    free(p);
    p = NULL;

    p = (char *)malloc(100);

    muntrace();   // 结束跟踪，并生成日志信息

    return 0;
}
```

日志

程序运行后，会在指定路径下生成对应的log，打开看到一下类似内容：

```
root@Gemini:/tmp/sp/mnt/sda1# cat mtrace.log
= Start
@ ./mdemo:[0x104df] + 0x225d8 0x64
@ ./mdemo:[0x104e9] - 0x225d8
@ ./mdemo:[0x104f3] + 0x225d8 0x64
= End
```

从记录内存可以看出，其格式如下：

@ 程序名称:[内存分配调用的地址] +/- 操作的内存地址 申请内存大小

+ 表示分配；- 表示释放

泄漏分析

使用mtrace工具分析日志信息

把日志传到Linux PC中，执行`mtrace 可执行文件路径 日志文件路径`命令，比如：`mtrace mdemo ./mtrace.log`，出现以下内容：

```
r.zhou@cdoa03:~/file/memoryLeak_demo$ mtrace mdemo ./mtrace.log

Memory not freed:
-----------------
           Address     Size     Caller
0x00000000000225d8     0x64  at /home/r.zhou/file/memoryLeak_demo/mtrace_demo1.cpp:15
r.zhou@cdoa03:~/file/memoryLeak_demo$
r.zhou@cdoa03:~/file/memoryLeak_demo$
```

使用addr2line工具定位源码位置

addr2line：是将指令的地址和可执行映像转换为文件名、函数名和源代码行数的工具，可用于快速定位出错的位置。

使用：addr2line -e 可执行文件 内存分配调用的地址

```
r.zhou@cdoa03:~/file/memoryLeak_demo$ addr2line -e mdemo 0x104e9
/home/r.zhou/file/memoryLeak_demo/mtrace_demo1.cpp:13
r.zhou@cdoa03:~/file/memoryLeak_demo$
r.zhou@cdoa03:~/file/memoryLeak_demo$
r.zhou@cdoa03:~/file/memoryLeak_demo$
r.zhou@cdoa03:~/file/memoryLeak_demo$ addr2line -e mdemo 0x104f3
/home/r.zhou/file/memoryLeak_demo/mtrace_demo1.cpp:15
r.zhou@cdoa03:~/file/memoryLeak_demo$
```

3、问题记录

问题一

问题描述：mtrace()、muntrace()不能反复调用，即如果在接口内部中使用mtrace()、muntrace()，则该接口不能反复调用，否则会带来内存泄漏。

解决方法：在该接口需要多次调用时，在接口调用之前使用mtrace()追踪，接口结束之后使用muntrace()结束追踪。

问题二

问题描述：mtrace可以通过addr2line定位内存泄漏的位置，但显示"??"，查询后发现原因可能是没有 -g 编译后的调试信息，但在MakeFile中发现已经有-g参数。

```
r.zhou@cdoa03:~/QtTest/audioPlay/audioPlay$ mtrace audioPlay qttrace.log
- 0x00000000b3901d00 Free 2 was never alloc'd 0x416aa0cb
- 0x00000000000dbc60 Free 123 was never alloc'd 0x416c2567
- 0x00000000000cf048 Realloc 166 was never alloc'd 0x4111ba6b
- 0x00000000000cf060 Realloc 168 was never alloc'd 0x4111ba7f

Memory not freed:
-----------------
           Address     Size     Caller
0x000000000011d930     0x54  at 0x4186cab1
0x00000000b3901410     0x38  at 0x4186cab1
0x00000000000cd870      0xc  at 0x4186cab1
0x00000000000cf048     0x10  at 0x4111ba6b
0x00000000000cf060     0x20  at 0x4111ba7f
r.zhou@cdoa03:~/QtTest/audioPlay/audioPlay$ addr2line -e audioPlay 0x000000000011d930
??:0
r.zhou@cdoa03:~/QtTest/audioPlay/audioPlay$
```

暂无解决方法。

可以尝试使用c++filt解析函数名称：

Memory not freed显示的是内存未释放的部分，复制Caller中的地址，到日志中查找。比如0x4186cab1，搜索之后有以下内容：

```
Y:\QtTest\audioPlay\audioPlay\qttrace.log (130 hits)
   Line 3: @ /lib/libstdc++.so.6:(_Znwj+0x14)[0x4186cab1] + 0xcd1c0 0x2c
   Line 4: @ /lib/libstdc++.so.6:(_Znwj+0x14)[0x4186cab1] + 0xdd338 0xa8
   Line 5: @ /lib/libstdc++.so.6:(_Znwj+0x14)[0x4186cab1] + 0xcf280 0x40
   Line 21: @ /lib/libstdc++.so.6:(_Znwj+0x14)[0x4186cab1] + 0xcd1c0 0x2c
   Line 22: @ /lib/libstdc++.so.6:(_Znwj+0x14)[0x4186cab1] + 0xdd338 0xa8
   Line 23: @ /lib/libstdc++.so.6:(_Znwj+0x14)[0x4186cab1] + 0xcf280 0x40
```

0x4186cab1对应的的_Znwj是编译器根据**Name Mangling**的方式，重新组织源代码中定义的函数名，使用c++filt解析函数名称，如下图所示，执行`c++filt _Znwj`，得到还原函数名，则可以确定哪个函数没有释放。

```
r.zhou@cdoa03:~/QtTest/audioPlay/audioPlay$
r.zhou@cdoa03:~/QtTest/audioPlay/audioPlay$ c++filt _Znwj
operator new(unsigned int)
r.zhou@cdoa03:~/QtTest/audioPlay/audioPlay$
```

四、Valgrind

1、介绍

Valgrind 是运行在Linux 上的多用途代码剖析和内存调试软件，它包括如下一些工具：

(1) 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，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。本文重点介绍的部分。

(2) Callgrind，它主要用来检查程序中函数调用过程中出现的问题。

(3) Cachegrind，它主要用来检查程序中缓存使用出现的问题。

(4) Helgrind，它主要用来检查多线程程序中出现的竞争问题。

(5) Massif，它主要用来检查程序中堆栈使用中出现的问题。

(6) Extension，可以利用core提供的功能，自己编写特定的内存调试工具。

更多信息见参考资料中第19项。

2、移植

1. `cd 源码路径`
2. `./autoconfig.sh`生成configure
3. `./configure`生成makefile，需要自行设置CC、CXX、AR、--host、--prefix
   - 以xu为例：`./configure CC=arm-none-linux-gnueabihf-gcc CXX=arm-none-linux-gnueabihf-g++ AR=arm-none-linux-gnueabi-ar --host=armv7-linux --prefix=/home/r.zhou/valgrind/install`
     - CC、CXX、AR在build/tools/**arm-9.2_eabihf**/bin/下可以找到，其中arm-9.2_eabihf目录根据toolchain不同而不同
     - --host=armv7-linux是8368xu下指定的运行主机，8368p则为--host=armv8-linux
     - --prefix则为安装路径，根据自己需要更改
     - 经过试验，AR非必须选项
4. make
5. make install

3、使用方法

1. 输出路径下的 bin/ 和 /libexec到U盘，原因：valgrind编译后的lib目录很大，开发板没有过多空间存放，所以放置到U盘中。

bin、libexec放置位置：Valgrind生成的bin、libexec(xu)

2. 串口输入命令，声明环境变量：

```
export VALGRIND_LIB=xxxx（指明拷贝到开发板下的libexec目录）
eg：export VALGRIND_LIB=/mnt/sda1/valgrind/libexec/valgrind
```

3. 串口输入命令，运行：

```
valgrind --tool=xxx 参数 /程序路径
eg：/mnt/sda1/valgrind/bin/valgrind --tool=memcheck --leak-check=yes --leak-resolution=low /application/bin/Launcher
```

- 在运行Valgrind时，你必须指明想用的工具，如果省略工具名，默认运行memcheck
- --tool=memcheck，表示使用内存检测工具
- --leak-check=yes，表示开启内存泄漏检测功能，给出每一个独立的泄露的详细信息
- --leak-resolution=low，确定memcheck怎么考虑不同的栈是相同的情况。当设置为low时，只需要前两层栈匹配就认为是相同的情况；当设置为med，必须要四层栈匹配，当设置为high时，所有层次的栈都必须匹配
- 更多参数见参考资料第19条：Valgrind官网

4、日志格式

内存泄漏类型：申请空间后未释放

实例代码：

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

int main(int argc,char *argv[])
{
    int *p = (int*)malloc(sizeof(int));
    *p = 100;

    return 0;
}
```

使用Valgrind的Memcheck工具，日志如下：

```
root@Gemini:/# export VALGRIND_LIB=/mnt/sda1/valgrind/libexec/valgrind
root@Gemini:/#
root@Gemini:/# /mnt/sda1/valgrind/bin/valgrind --tool=memcheck --leak-check=yes -
-leak-resolution=low /mnt/sda1/not_free
==1087== Memcheck, a memory error detector
==1087== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1087== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==1087== Command: /mnt/sda1/not_free
==1087==
==1087==
==1087== HEAP SUMMARY:
==1087==     in use at exit: 4 bytes in 1 blocks
==1087==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==1087==
==1087== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1087==    at 0x4818E6C: malloc (vg_replace_malloc.c:431)
==1087==    by 0x1031EB: main (not_free.cpp:6)
==1087==
==1087== LEAK SUMMARY:
==1087==    definitely lost: 4 bytes in 1 blocks
==1087==    indirectly lost: 0 bytes in 0 blocks
==1087==      possibly lost: 0 bytes in 0 blocks
==1087==    still reachable: 0 bytes in 0 blocks
==1087==         suppressed: 0 bytes in 0 blocks
==1087==
==1087== For lists of detected and suppressed errors, rerun with: -s
==1087== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

- ==1087==：指进程号为1087
- 第一段是Valgrind的基本信息

```
==1087== Memcheck, a memory error detector
==1087== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1087== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==1087== Command: /mnt/sda1/not_free
```

- 第二段是对堆内存分配的总结信息，其中提到程序一共申请了1次内存，其中0次释放，4个字节被分配

```
==1087== HEAP SUMMARY:
==1087==     in use at exit: 4 bytes in 1 blocks
==1087==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
```

- 第三段的内容描述了内存泄露的具体信息，其中有一块内存占用4字节，在调用malloc分配，调用栈中可以看到是mian函数最后调用了malloc，所以这一个信息是比较准确的定位了我们泄露的内存是在哪里申请的

```
==1087== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1087==    at 0x4818E6C: malloc (vg_replace_malloc.c:431)
==1087==    by 0x1103EB: main (not_free.cpp:6)
```

- 第四段的内容是总结，有一块4字节的内存泄露

```
==1087== LEAK SUMMARY:
==1087==    definitely lost: 4 bytes in 1 blocks
==1087==    indirectly lost: 0 bytes in 0 blocks
==1087==      possibly lost: 0 bytes in 0 blocks
==1087==    still reachable: 0 bytes in 0 blocks
==1087==         suppressed: 0 bytes in 0 blocks
==1087==
==1087== For lists of detected and suppressed errors, rerun with: -s
==1087== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

5、问题记录

问题一

问题描述：当执行完使用方法中的第三步后，出现以下信息显示：

```
==1063== Memcheck, a memory error detector
==1063== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1063== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==1063== Command: /application/bin/Launcher
==1063==

valgrind:  Fatal error at startup: a function redirection
valgrind:  which is mandatory for this platform-tool combination
valgrind:  cannot be set up.  Details of the redirection are:
valgrind:
valgrind:  A must-be-redirected function
valgrind:  whose name matches the pattern:      index
valgrind:  in an object with soname matching:   ld-linux-armhf.so.3
valgrind:  was not found whilst processing
valgrind:  symbols from the object with soname: ld-linux-armhf.so.3
valgrind:
valgrind:  Possible fixes: (1, short term): install glibc's debuginfo
valgrind:  package on this machine.  (2, longer term): ask the packagers
valgrind:  for your Linux distribution to please in future ship a non-
valgrind:  stripped ld.so (or whatever the dynamic linker .so is called)
valgrind:  that exports the above-named function using the standard
valgrind:  calling conventions for this platform.  The package you need
```

```
valgrind:  to install for fix (1) is called
valgrind:
valgrind:    On Debian, Ubuntu:                 libc6-dbg
valgrind:    On SuSE, openSuSE, Fedora, RHEL:   glibc-debuginfo
valgrind:
valgrind:  Note that if you are debugging a 32 bit process on a
valgrind:  64 bit system, you will need a corresponding 32 bit debuginfo
valgrind:  package (e.g. libc6-dbg:i386).
valgrind:
valgrind:  Cannot continue -- exiting now.  Sorry.
```

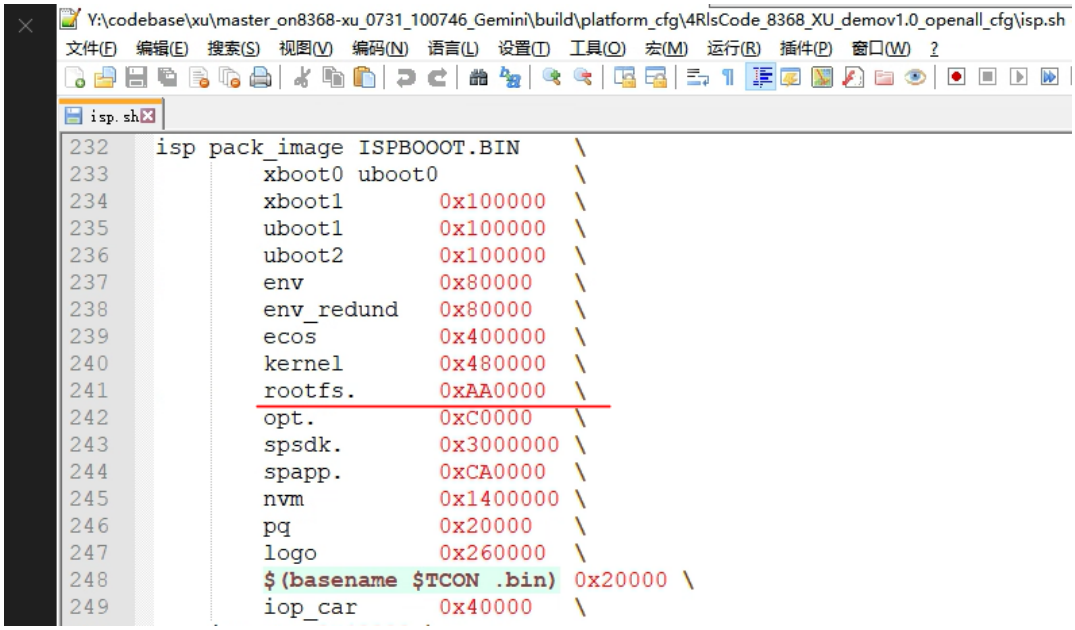
原因：glibc被strip，平台上无法运行。

解决方法：替换被strip的libc-2.30.so , ld-2.30.so(8368-xu)，步骤如下：

1. rootfs/gen_rootfs_lib.sh文件里find .... Xargs ... strip 语句进行了strip，都注释掉，代码如下：

```
find $WOKE_DIR -type f | xargs -i $lstrip '{}' 2>/dev/null
```

2. gen_rootfs_lib.sh使用的lib是从rootfs/rootlib.tar包里面解压出来的，需要解开此包，用build/tools/arm-9.2_eabihf/arm-none-linux-gnueaihf/libc/lib路径下的libc-2.30.so、 ld-2.30.so替换，再打包回去
    - arm-9.2_eabihf路径根据toolchain变化
    - build/tools/arm-9.2_eabihf/arm-none-linux-gnueaihf/libc/lib路径下的libc-2.30.so、 ld-2.30.so未被strip
3. 在rootfs目录下make，重新生成相关的.sqfs image文件
4. 在根目录下面 执行 make -f rootfs/gemini.mk update
5. make rom 打包
    - 若执行make rom，出现Error:Assgined partiton size is less than the image size: rootf., 5242880 < 11071488，则进行以下步骤：
        1. 根据make list中的配置，在build/platform/下找到对应的isp.sh
        2. 如下图所示，rootfs处本来是0x500000(转换为10进制，则为5242880)，小于0xA8F000(转换为10进制，为11071488)，但由于rootfs的值要为0x20000的整数倍，所以更改为0xAA0000
        3. 保存后在根目录再次执行make rom即可



问题二

问题描述：valgrind本身会消耗大量内存，测试Launcher，则测试进程会快会因为内存消耗太大被系统杀掉。

无解决方法，不能使用valgrind来测试Launcher。

五、Address Sanitizer （ASan）

1、介绍

Address Sanitizer是一款面向C/C++语言的内存错误问题检查工具，相比Valgrind，Address Sanitizer（ASan）要快很多，只会拖慢程序两倍左右。

Address Sanitizer由一个编译器instrumentation模块和一个提供malloc()/free()替代项的运行时库组成。

- instrument 静态插桩模块，对栈上对象、全局对象、动态分配的对象分配 redzone，以及针对这些内存做访问检测。
- libasan.so.x 运行时库，替换 `malloc` / `free` / `memcpy` / `memset` 等实现、提供报错函数。

除此之外，还有一些内核工作的开发工具：

- The Kernel Memory Sanitizer (KMSAN)：动态错误检测器，旨在查找未初始化值的使用情况。不适合用于生产环境，因为它会极大地增加内核内存占用并降低整个系统的速度。
- The Undefined Behavior Sanitizer (UBSAN)：运行时未定义行为检查器，使用编译时插桩来捕捉未定义行为.
- Kmemleak，提供了一种以类似于跟踪垃圾收集器的方式检测可能的内核内存泄漏的方法。
- The Kernel Concurrency Sanitizer (KCSAN)，动态竞争检测器，它依赖于编译时检测，并使用基于观察点的采样方法来检测竞争,主要目的是检测数据竞争。
- Kernel Electric-Fence (KFENCE)，低开销的基于采样的内存安全错误检测器，检测堆越界访问、use-after-free和invalidate -free错误。

更多信息可见参考资料13-18条。

2、编译选项

- -fsanitize=address：开启内存越界检测和访问已经释放内存
- -fsanitize-recover=address：一般后台程序为保证稳定性，不能遇到错误就简单退出，而是继续运行，采用该选项支持内存出错之后程序继续运行，需要叠加设置环境变量ASAN_OPTIONS=halt_on_error=0才会生效；若未设置此选项，则内存出错即报错退出
- -fno-omit-frame-pointer：去使能栈溢出保护

3、运行选项

ASAN_OPTIONS是Address-Sanitizier的运行选项环境变量。

- halt_on_error=0：检测内存错误后继续运行
- detect_leaks=1：使能内存泄露检测
- malloc_context_size=15：内存错误发生时，显示的调用栈层数为15

3、注意事项

1. ASAN（Address-Sanitizier）早先是LLVM中的特性，后被加入GCC 4.8，在GCC 4.9后加入对ARM平台的支持。因此GCC 4.8以上版本使用ASAN时不需要安装第三方库，通过在编译时指定编译CFLAGS即可打开开关。
2. 依赖libasan.so，liblsan.so相关库，toolchain自带，需要打包到文件系统中。
3. 如果被检测对象是通过的dlopen方式加载，则需要用LD_PRELOAD方式执行加载对象的libasan.so位置。

4、举例

1、以application为例，可进行以下配置：

编译选项：

添加位置：linux/sdk/prebuilt/qt/standard/host/mkspecs/devices/linux-aarch63-gnu/qmake.conf

```
QMAKE_CFLAGS += -fsanitize=address -fsanitize-recover=address -fno-omit-frame-pointer -fsanitize=leak -g -O1
QMAKE_CXXFLAGS += -fsanitize=address -fsanitize-recover=address -fno-omit-frame-pointer  -fsanitize=leak -g -O1
```

运行选项：

添加位置：rootfs/gen_init_environ_rc.mk

```
export LD_PRELOAD /lib/libasan.so
export ASAN_OPTIONS=halt_on_error=0:malloc_context_size=10:detect_leaks=1
```

- 如果没有用到的dlopen流程LD_PRELOAD=/lib/libasan.so可以不要
- “/lib/libasan.so“替换成实际的libasan.so的位置

2、以测试程序为例

问题代码：

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void func()
5 {
6   int *a = (int*)malloc(sizeof(int)*10);
7   if(a != NULL){
8       *a = 1;
9       printf("a is:%d.\n",*a);
10      free(a);
11      *a = 2;
12      printf("error a is:%d.\n",*a);
13  }
14 }
15
16 int main(int argc,char *argv[])
17 {
18  func();
19
20  return 0;
```

```
21 }
```

编译：

```
gcc -fsanitize=address -fsanitize-recover=address -fno-omit-frame-pointer -
fsanitize=leak -g Asan_use_after_free.cpp -o ASan_use_after_free
```

运行：

```
./ASan_use_after_free
```

日志如下：

```
=================================================================
==1074==ERROR: AddressSanitizer: heap-use-after-free on address 0xb4700f50 at pc
0x00010745 bp 0xbefff4c0 sp 0xbefff4c4
WRITE of size 4 at 0xb4700f50 thread T0
    #0 0x10742 in func()
/home/r.zhou/file/memoryLeak_demo/Asan_use_after_free.cpp:11
    #1 0x1076e in main
/home/r.zhou/file/memoryLeak_demo/Asan_use_after_free.cpp:18
    #2 0x41958ae8 in __libc_start_main /tmp/dgboter/bbs/rhev-vm7--
rhe6x86_64/buildbot/rhe6x86_64--arm-none-linux-
gnueabihf/build/src/glibc/csu/libc-start.c:308

0xb4700f50 is located 0 bytes inside of 40-byte region [0xb4700f50,0xb4700f78)
freed by thread T0 here:
    #0 0xb6b40ef8 in __interceptor_free (/lib/libasan.so.5+0xc0ef8)

previously allocated by thread T0 here:
    #0 0xb6b41162 in malloc (/lib/libasan.so.5+0xc1162)

SUMMARY: AddressSanitizer: heap-use-after-free
/home/r.zhou/file/memoryLeak_demo/Asan_use_after_free.cpp:11 in func()
```

- 错误进程号：内存问题发生的Linux进程号。此处为1076。
- 错误类型：heap-use-after-free，释放后使用。
- 错误原因：WRITE of size 4 at 0xb4700f50 thread T0，该处访问了释放后的内存。
- 发生位置：#0 0x10742 in func()
  /home/r.zhou/file/memoryLeak_demo/Asan_use_after_free.cpp:11。
- 内存释放位置：freed by thread T0 here。
- 内存申请位置：previously allocated by thread T0 here。
- 概要说明：SUMMARY: AddressSanitizer: heap-use-after-free
  /home/r.zhou/file/memoryLeak_demo/Asan_use_after_free.cpp:

  11 in func()。

六、Memory Monitor Tool

1、介绍

Memory Monitor Tool, 我们简称为 MMT. 本工具定期会 output 一次 memory information 到 terminal 或者 storage 内。

Memory information 包含了每个 process 的 PID/RSS/AnonPage/Mapped 与获取当下的系统时间 timestamp。

与excel搭配产出可视化图表，详情见：http://wiki2.sunplus.com/dvdsa/index.php/Memory_Monitor_Tool

七、参考资料

1. 内存泄漏定位工具之 mtrace
2. linux内存分析工具pmap
3. Linux学习：使用 procrank 测量系统内存使用情况
4. Linux /proc/meminfo 详解
5. Linux SWAP 深度解读
6. 深度探究Linux内存中的/proc/meminfo参数
7. 内存管理(二)：深入理解内存的动态申请和释放原理
8. https://github.com/google/sanitizers/wiki/AddressSanitizer
9. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
10. AddressSanitizerFlags · google/sanitizers Wiki · GitHub
11. SanitizerCommonFlags · google/sanitizers Wiki · GitHub
12. 查内存泄漏试试AScan
13. KASAN:the kernel address Sanitizer
14. KMSAN:the kernel memory Sanitizer
15. UBSAN:the undefined behavior Sanitizer
16. Kmemleak:kernel memory leak detector
17. KCSAN:the kernel concurrency Sanitizer
18. KFENCE:the kernel Electric-fence
19. Valgrind官网

八、工具放置位置

1. meminfo.sh：SharePoint:Application/性能分析/内存泄漏/测试工具
2. procank（适用xu）：SharePoint:Application/性能分析/内存泄漏/测试工具

   procank（根据平台toolchain更换）：SharePoint:技术领域/系统分析/分析工具/tools
3. myvt（适用xu）：SharePoint:Application/性能分析/内存泄漏/测试工具

   myvt源码：SharePoint:Application/性能分析/内存泄漏/自动化测试myvt源码
4. Valgrind：SharePoint:技术领域/系统分析/分析工具/tools/aarch32_hf_9.2.1/Valgrind
5. Memory Monitor Tool：http://wiki2.sunplus.com/dvdsa/index.php/Memory_Monitor_Tool