

*Android **Debug Debug Debug***

: logcat, gdb, addr2line, DDMS, kernel



ANDROID

chunghan.yi@gmail.com, slowboot

Revision	작성자	비고
0.1	이 충 한	최초 작성 11/03/2011
0.2	이 충 한	11/07/2011
0.3	이 충 한	11/08/2011
0.4	이 충 한	11/09/2011

목차

- **1. basic tools: logcat, dumpstate, dumphsys**
- **2. C/C++ Code Debugging**
 - 2.1 *strace*
 - 2.2 *gdb & gdbserver*
 - 2.3 *addr2line & objdump*
 - 2.4 *libc.debug.malloc*
 - 2.5 *JNI Debugging - CheckJNI*
 - 2.6 *Valgrind(TODO)*
- **3. Java Code Debugging**
 - 3.1 *DDMS*
 - 3.2 *VM Heap*
 - 3.3 *Java Code Off/stack trace code 추가하기*
- **4. Kernel Code Debugging**
 - 4.1 *dmesg*
 - 4.2 *addr2line & objdump*
 - 4.3 *gdb*
 - 4.4 *kgdb (사용 불가)*
 - 4.5 *kprobe/jprobe를 사용한 실시간 debugging*
- **5. TODO**
-

0. 이 문서에서 다루고자 하는 내용

- 1) Out of memory issue(VM heap, native heap)
- 2) Native SIGSEGV/SEG_MAPERR issue
- 3) Native memory Leak, overrun, double free issue
- 4) kernel panic/oops issue
- ➔ 이런 문제로 죽는 것에 대한 원인을 분석
- ➔ 유용한 *debugging* 기법 소개 및 *back trace* 방법 소개
- ➔ (가능하다면)해결책 제시

(*) 본 문서에 테스트한 내용은 *gingerbread* 2.3.4 및 *Qualcomm* 칩을 기준으로 하였다.

1. Basic tools: logcat, dumpstate, dumphsys

- `# adb shell logcat`
- `# adb shell dumpstate > state.txt`
→ *system, status, counts, and statistics* 정보를 dump
- `# adb shell dumphsys > sys.txt`
- `# adb shell dumphsys meminfo` ← 각각의 process가 사용하는 메모리 내역 출력
- `# adb shell procrank` ← 전체 process의 메모리 사용량 출력
- `# adb shell "top -m 10 -s rss -d 2"` ← 메모리 leak이 발생(??)하는지 모니터링하는 명령
- `# adb shell dumphsys meminfo pid` ← pid를 갖는 process의 memory 사용정보 출력

(*) 기본적인 명령이라, 자세한 의미는 별도로 기술하지 않았음.

2. C/C++ Code Debugging

2.1 strace

- # strace -p pid_of_process
 - ➔ C/C++ process가 호출하는 system call 분석 시 용이
 - ➔ -p option을 사용하여 현재 동작 중인 process에 대해 strace를 돌릴 수 있음.

(*) strace는 debug하려는 program(or process)가 호출하는 각종 system call을 추적할 수 있음.

(*) strace로 debugging하고자 하는 위치를 알아내고, gdb로 breakpoint를 지정한 후, breakpoint까지 trace하는 방법을 활용하면 보다 효과적으로 Debugging이 가능할 수도 있음^^.

2.2 gdb & gdbserver(1)

(*) C/C++로 만들어진 code debugging

- **<phone에서 설정할 내용>**

sudo adb shell ← 물론 이건 PC에서 실행

gdbserver :5039 --attach pid_of_mediaserver ← mediaserver의 예임(실제 ps하여 얻은 pid 값 사용)

← 돌고 있는 mediaserver process를 gdb에 붙이는 방법

- **<PC에서 실행할 내용>**

sudo adb forward tcp:5039 tcp:5039

← phone의 gdb 결과를 PC로 forwarding해주는 설정

- # cd android/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin

./arm-eabi-gdb

~/YOUR_PATH/android/out/target/product/c_{XXXXXXXXXX}/symbols/system/bin/mediaserver

(gdb) set solib-absolute-prefix ~/YOUR_PATH/android/out/target/product/c_{XXXXXXXXXX}/symbols

(gdb) set solib-search-path ~/YOUR_PATH/android/out/target/product/c_{XXXXXXXXXX}/symbols/system/lib

(gdb) target remote :5039

(gdb) c

Continuing.

- ← mediaserver가 죽을 경우, 아래와 유사한 로그 발생

Program received signal SIGSEGV, Segmentation fault.

[Switching to Thread 1272]

0x6fd207b4 in strcasecmp (s1=0x10b80 "LG HBS700",

s2=0x2a000 <Address 0x2a000 out of bounds>) at bionic/libc/string/strcasecmp.c:83

83 while (cm[*us1] == cm[*us2++])

2.2 gdb & gdbserver(2)

- **(gdb) bt**
#0 0x6fd207b4 in strcmp (s1=0x10b80 "LG HBS700",
s2=0x2a000 <Address 0x2a000 out of bounds>) at bionic/libc/string/strcmp.c:83
#1 0x6970956c in android::AudioHardware::setParameters (this=0xb138,
keyValuePairs=<value optimized out>)
at hardware/msm7k/libaudio-qsd8k/AudioHardware.cpp:370
#2 0x6970a78e in android::A2dpAudioInterface::setParameters (
this=<value optimized out>, keyValuePairs=<value optimized out>)
at frameworks/base/services/audioflinger/A2dpAudioInterface.cpp:188
#3 0x68d254aa in android::AudioFlinger::setParameters (this=0xb000, ioHandle=0,
keyValuePairs=...) at frameworks/base/services/audioflinger/AudioFlinger.cpp:881
#4 0x690375ac in android::BnAudioFlinger::onTransact (this=0xb000,
code=<value optimized out>, data=..., reply=0x7ec62b90, flags=16)
at frameworks/base/media/libmedia/IAudioFlinger.cpp:993
#5 0x68d1ff02 in android::AudioFlinger::onTransact (this=<value optimized out>,
code=<value optimized out>, data=..., reply=0x6fd38350, flags=16)
at frameworks/base/services/audioflinger/AudioFlinger.cpp:7023
#6 0x68213566 in android::BBinder::transact (this=0xb004, code=20, data=...,
reply=0x7ec62b90, flags=16) at frameworks/base/libs/binder/Binder.cpp:107
-

2.2 gdb & gdbserver(3)

- **[TODO]** system_server의 C++ code를 debugging하기 위하여 gdb에 붙이려면 ?
 - ➔ init.rc를 수정해야 함 !

2.3 addr2line & objdump (1) – stack trace

• <logcat 내용 중, 문제가 되는 부분>

```
10-24 13:27:46.509 I/DEBUG (16058): pid: 162, tid: 17497 >>> system_server <<<
10-24 13:27:46.509 I/DEBUG (16058): signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr deadd00d
10-24 13:27:46.509 I/DEBUG (16058): r0 fffffe84 r1 deadd00d r2 00000026 r3 00000000
10-24 13:27:46.509 I/DEBUG (16058): r4 6ca9659c r5 0073f720 r6 6ca9659c r7 005a2068
10-24 13:27:46.509 I/DEBUG (16058): r8 00000000 r9 00000000 10 39b8dd50 fp 00000000
10-24 13:27:46.509 I/DEBUG (16058): ip 6ca966a8 sp 39b8dcb0 lr 6fd191e9 pc 6ca3d444 cpsr 20000030
10-24 13:27:46.509 I/DEBUG (16058): d0 74726f6261204d69 d1 617453657669746e
10-24 13:27:46.509 I/DEBUG (16058): d2 4d79746976697467 d3 6553726567616e0a
10-24 13:27:46.509 I/DEBUG (16058): d4 72656469766f7250 d5 61636f4c73704724
10-24 13:27:46.509 I/DEBUG (16058): d6 766f72506e6f6974 d7 6572685472656469
10-24 13:27:46.509 I/DEBUG (16058): d8 0000000000000000 d9 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d10 0000000000000000 d11 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d12 0000000000000000 d13 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d14 0000000000000000 d15 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d16 000000072b626aa0 d17 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d18 0000000000000000 d19 3fee8c97c0000000
10-24 13:27:46.509 I/DEBUG (16058): d20 40713c72c0000000 d21 4039337500000000
10-24 13:27:46.509 I/DEBUG (16058): d22 3ff0000000000000 d23 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d24 0000000000000000 d25 3fee8c97c0000000
10-24 13:27:46.509 I/DEBUG (16058): d26 4039337500000000 d27 3fee8c97c0000000
10-24 13:27:46.509 I/DEBUG (16058): d28 0000000000000000 d29 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): d30 0000000000000000 d31 0000000000000000
10-24 13:27:46.509 I/DEBUG (16058): scr 60000010
```

(*) arm-eabi-addr2line 명령을 이용하면,
Program counter 값과 라이브러리 명을 이용하여
문제가 되는 지점의 code 위치를 알아낼 수 있다.

◀ 아래 내용을 역으로 분석 시도 하고자 함(PC 값을 function name으로 전환 작업) – 죽을 당시의 stack의 내용임 !

```
12) 10-24 13:27:46.569 I/DEBUG (16058): #00 pc 0003d444 /system/lib/libdvm.so
11) 10-24 13:27:46.569 I/DEBUG (16058): #01 pc 00060978 /system/lib/libdvm.so
10) 10-24 13:27:46.569 I/DEBUG (16058): #02 pc 00060c1a /system/lib/libdvm.so
9) 10-24 13:27:46.569 I/DEBUG (16058): #03 pc 00060492 /system/lib/libdvm.so
8) 10-24 13:27:46.569 I/DEBUG (16058): #04 pc 000446b8 /system/lib/libdvm.so
7) 10-24 13:27:46.569 I/DEBUG (16058): #05 pc 0005f408 /system/lib/libandroid_runtime.so
6) 10-24 13:27:46.569 I/DEBUG (16058): #06 pc 0005f520 /system/lib/libandroid_runtime.so
5) 10-24 13:27:46.569 I/DEBUG (16058): #07 pc 0006423c /system/lib/libandroid_runtime.so
4) 10-24 13:27:46.569 I/DEBUG (16058): #08 pc 00013fca /system/lib/libdbus.so
3) 10-24 13:27:46.569 I/DEBUG (16058): #09 pc 000634d0 /system/lib/libandroid_runtime.so
2) 10-24 13:27:46.569 I/DEBUG (16058): #10 pc 000118f4 /system/lib/libc.so
1) 10-24 13:27:46.569 I/DEBUG (16058): #11 pc 000114c0 /system/lib/libc.so
```

pc 0003d444 /system/lib/libdvm.so

파일명, line number,
Function name 추출

arm-eabi-addr2line -f -e ./libdvm.so 0003d444

2.3 addr2line & objdump(2)

- <function name을 추출한 결과 – 위의 빨간색 표시 부분을 아래에서부터 위로 trace한 결과>
- # cd android/out/target/product/ /symbols/system/lib
- 12) arm-eabi-addr2line -f -e ./libdvm.so 0003d444
- dvmAbort
- /home/android/dalvik/vm/Init.c:1716
-
- 11) arm-eabi-addr2line -f -e ./libdvm.so 00060978
- findClassNoInit
- /home/android/dalvik/vm/oo/Class.c:1401
-
- 10) arm-eabi-addr2line -f -e ./libdvm.so 00060c1a
- dvmFindSystemClassNoInit
- /home/android/dalvik/vm/oo/Class.c:1356
-
- 9) arm-eabi-addr2line -f -e ./libdvm.so 00060492
- dvmFindClassNoInit
- /home/android/dalvik/vm/oo/Class.c:1197
-
- 8) arm-eabi-addr2line -f -e ./libdvm.so 000446b8
- FindClass
- /home/android/dalvik/vm/Jni.c:1933
-
- 7) arm-eabi-addr2line -f -e ./libandroid_runtime.so 0005f408
- _ZN7_JNIEnv9FindClassEPKc
- /home/android/dalvik/libnativehelper/include/nativehelper/jni.h:518
-
- 6) arm-eabi-addr2line -f -e ./libandroid_runtime.so 0005f520
- _ZN7android29parse_adapter_property_changeEP7_JNIEnvP11DBusMessage
- /home/android/frameworks/base/core/jni/android_bluetooth_common.cpp:694
-

2.3 addr2line & objdump(3)

- **5) arm-eabi-addr2line -f -e ./libandroid_runtime.so 0006423c**
- _ZN7androidL12event_filterEP14DBusConnectionP11DBusMessagePv
- /home/android/frameworks/base/core/jni/android_server_BluetoothEventLoop.cpp:838
-
- **4) arm-eabi-addr2line -f -e ./libdbus.so 00013fca**
- dbus_connection_dispatch
- /home/android/external/dbus/dbus/dbus-connection.c:4366
-
- **3) arm-eabi-addr2line -f -e ./libandroid_runtime.so 000634d0**
- _ZN7androidL13eventLoopMainEPv
- /home/android/frameworks/base/core/jni/android_server_BluetoothEventLoop.cpp:615
-
- **2) arm-eabi-addr2line -f -e ./libc.so 000118f4**
- __thread_entry
- /home/android/bionic/libc/bionic/pthread.c:207
-
- **1) arm-eabi-addr2line -f -e ./libc.so 000114c0**
- pthread_create
- /home/android/bionic/libc/bionic/pthread.c:343

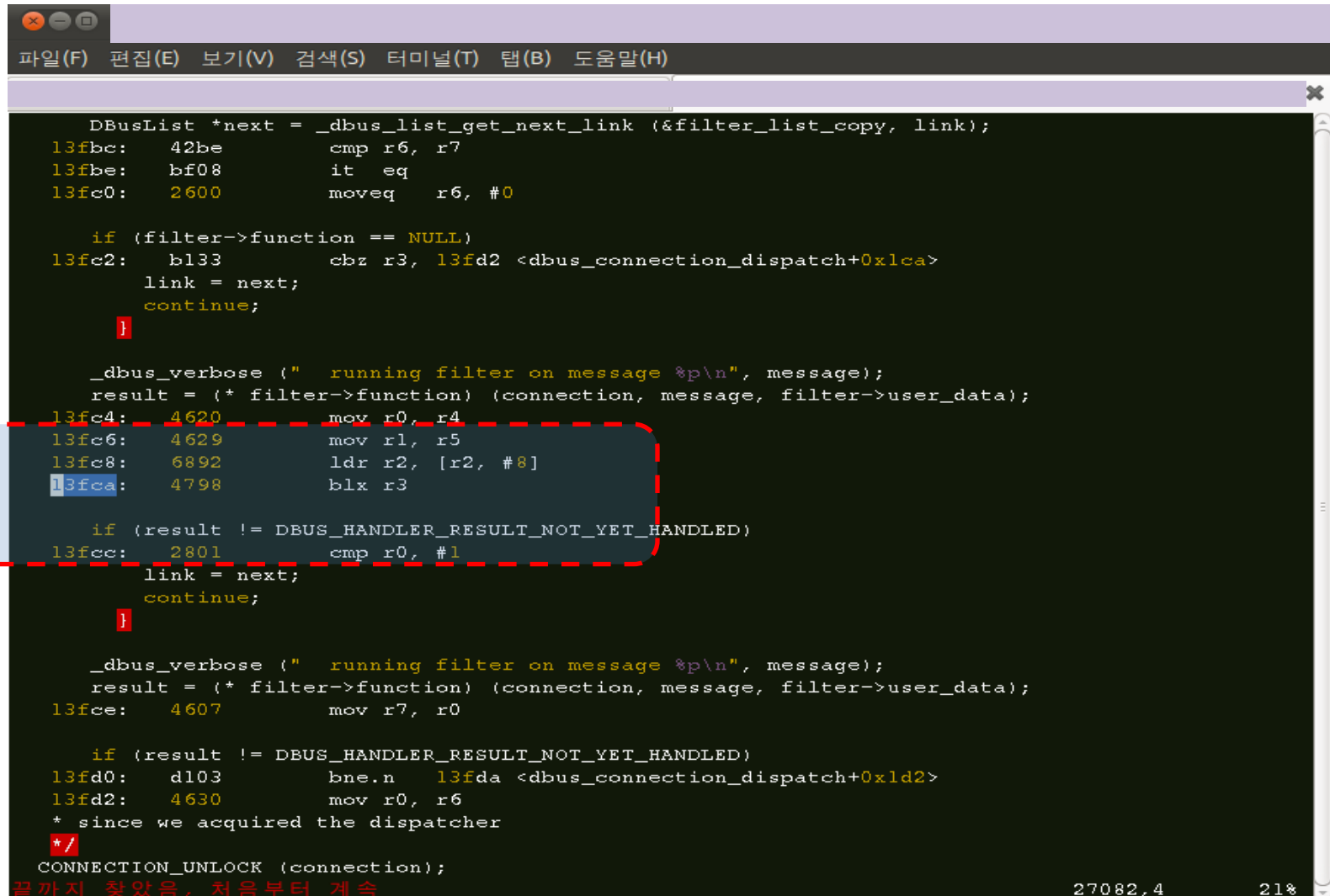
(*) -C option을 사용하면 function name이 깔끔하게 출력될 것임.

2.3 addr2line & objdump(4)

(*) *arm-eabi-objdump*를 사용하면 해당 *library*에 대한 *disassemble* 결과를 얻을 수 있으므로 *arm-eabi-addr2line* 보다 자세한 *debugging*이 가능하다.

- <logcat 내용 중, stack trace 하고자 하는 부분>
 - 4) 10-24 13:27:46.569 I/DEBUG (16058): #08 pc 00013fca /system/lib/libdbus.so
 - 3) 10-24 13:27:46.569 I/DEBUG (16058): #09 pc 000634d0 /system/lib/libandroid_runtime.so
 - 2) 10-24 13:27:46.569 I/DEBUG (16058): #10 pc 000118f4 /system/lib/libc.so
 - 1) 10-24 13:27:46.569 I/DEBUG (16058): #11 pc 000114c0 /system/lib/libc.so
 - ...
- # cd android/out/target/product/[redacted]/symbols/system/lib
- # arm-eabi-objdump -S ./libdbus.so > aaa ← -S는 역어셈블 옵션임
- # vi aaa
 - 13fca로 출력된 내용 검색
 - (다음 페이지 참조 – 검색된 부분의 주변을 살펴 보면, 파일 및 함수 이름을 찾을 수 있음)
- (*) *dump*하려는 *library*가 C++로 작성되었을 경우에는 -C 옵션도 함께 사용한다.
- # arm-eabi-objdump -S -C ./libdbus.so
- (*) *arm-eabi-objdump*는 *android/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin* 아래에 있음.

2.3 addr2line & objdump(5)



```
DBusList *next = _dbus_list_get_next_link (&filter_list_copy, link);
13fbc: 42be      cmp r6, r7
13fbe: bf08      it eq
13fc0: 2600      moveq r6, #0

    if (filter->function == NULL)
13fc2: b133      cbz r3, 13fd2 <dbus_connection_dispatch+0x1ca>
    link = next;
    continue;
}

    _dbus_verbose ("    running filter on message %p\n", message);
    result = (* filter->function) (connection, message, filter->user_data);
13fc4: 4620      mov r0, r4
13fc6: 4629      mov r1, r5
13fc8: 6892      ldr r2, [r2, #8]
13fca: 4798      blx r3

    if (result != DBUS_HANDLER_RESULT_NOT_YET_HANDLED)
13fcc: 2801      cmp r0, #1
    link = next;
    continue;
}

    _dbus_verbose ("    running filter on message %p\n", message);
    result = (* filter->function) (connection, message, filter->user_data);
13fce: 4607      mov r7, r0

    if (result != DBUS_HANDLER_RESULT_NOT_YET_HANDLED)
13fd0: d103      bne.n 13fda <dbus_connection_dispatch+0x1d2>
13fd2: 4630      mov r0, r6
* since we acquired the dispatcher
*/
CONNECTION_UNLOCK (connection);
```

끝까지 찾았음, 처음부터 계속

27082,4 218

2.3 addr2line & objdump(6) – stack back trace에 대한 의견

- (*) **Signal 11 (SIGSEGV)** is the signal sent to a process when it makes an invalid memory reference or segmentation fault.
- (*) **SEGV_MAPERR** - Address not mapped to object(It's a segmentation fault which happens during malloc)
- (*) **Aborting or crashing in dmalloc()** usually indicates that the native heap has become corrupted. This is usually caused by native code in an application doing something bad.
-
- (*) malloc(), free()의 연장선상에서 SIGSEGV가 발생한 경우, 라이브러리 함수의 버그를 의심하기 전에 사용방법에 문제가 없는지, 특히 이중해제를 하지 않는지, 할당 영역 범위 밖의 메모리를 사용하지는 않는지 확실히 확인이 필요함 !
- (*) SIGSEGV/SEG_MAPERR는 Native code쪽에서 발생한 것임.
- (*) stack을 dump한 내용의 경우, 문제를 추적하는 가장 확실한 방법이기도 하나, (stack이 파괴되어) 잘못된 정보를 주거나, 대개의 경우 전혀 다른 곳에서 문제가 된 것에 대한 여파로 발생한 사실만을 보여주고 있어, debugging이 간단하지 않다.

2.4 libc.debug.malloc - Native code에 대한 memory 문제 검출 방법(1)

- # adb shell stop ← 전체 system을 내림
- # adb shell setprop libc.debug.malloc 10
 - 1 - perform leak detection
 - 5 - fill allocated memory to detect overruns
 - 10 - fill memory and add sentinels to detect overruns
- # adb shell start ← 전체 system을 다시 올림

(*) stack back trace 결과로 문제의 원인을 못 찾을 경우(SIG_SEGV/SEG_MAPERR 등), Memory 누수(leak), buffer overrun 등을 체크해 보아야 한다.

(*) 위의 명령을 실행하면, android framework(runtime)이 내려갔다, 다시 올라가게 된다.

(*) 또한, 시스템이 느려질 수 있다.

(*) leak or overrun error가 발생할 경우, log에 stack strace 정보가 출력된다.

2.4 libc.debug.malloc – Native code에 대한 memory 문제 검출 방법(2)

- < # adb shell setprop libc.debug.malloc 1 했을 때의 로그 출력 내용 >
- I/libc (12165): sh using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12166): /system/bin/chmod using MALLOC_DEBUG = 1 (leak checker)
- W/SurfaceFlinger(12145): [WJMIN] dipsw.bytes=65535, dipsw.dmterminal=1
- I/libc (12167): sh using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12168): /system/bin/dumpstate using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12169): top using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12170): procrank using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12171): logcat using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12173): logcat using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12175): logcat using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12177): netcfg using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12178): dmesg using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12179): vdc using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12180): vdc using MALLOC_DEBUG = 1 (leak checker)
- D/VoldCmdListener(87): asec list
- I/libc (12181): ps using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12182): /system/bin/cnd using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12183): ps using MALLOC_DEBUG = 1 (leak checker)
- I/libc (12184): librank using MALLOC_DEBUG = 1 (leak checker)

(*) TODO: 실제 memory leak이나 buffer overrun 문제를 보유한 process가 발견될 경우, 어떠한 형태로 출력되는지 테스트해 보아야 함 !!!

➔ Code 상으로는 stack back trace 결과가 출력되는 것으로 되어 있음. Stack trace 정보에 Symbol name이 출력되지 않으면, addr2line이나 objdump를 활용하면 된다.

2.5 JNI Debugging – CheckJNI(1)

What CheckJNI can do

CheckJNI 시, 매우 유용한 기능이 enable됨^^

To help, there's CheckJNI. It can catch a number of common errors, and the list is continually increasing. In Gingerbread, for example, CheckJNI can catch all of the following kinds of error:

- Arrays: attempting to allocate a negative-sized array.
- Bad pointers: passing a bad jarray/jclass/jobject/jstring to a JNI call, or passing a NULL pointer to a JNI call with a non-nullable argument.
- Class names: passing anything but the "java/lang/String" style of class name to a JNI call.
- Critical calls: making a JNI call between a GetCritical and the corresponding ReleaseCritical.
- Direct ByteBuffers: passing bad arguments to NewDirectByteBuffer.
- Exceptions: making a JNI call while there's an exception pending.
- JNIEnv*s: using a JNIEnv* from the wrong thread.
- jfieldIDs: using a NULL jfieldID, or using a jfieldID to set a field to a value of the wrong type (trying to assign a StringBuilder to a String field, say), or using a jfieldID for a static field to set an instance field or vice versa, or using a jfieldID from one class with instances of another class.
- jmethodIDs: using the wrong kind of jmethodID when making a Call*Method JNI call: incorrect return type, static/non-static mismatch, wrong type for 'this' (for non-static calls) or wrong class (for static calls).
- References: using DeleteGlobalRef/DeleteLocalRef on the wrong kind of reference.
- Release modes: passing a bad release mode to a release call (something other than 0, JNI_ABORT, or JNI_COMMIT).
- Type safety: returning an incompatible type from your native method (returning a StringBuilder from a method declared to return a String, say).
- UTF-8: passing an invalid [Modified UTF-8](#) byte sequence to a JNI call.

2.5 JNI Debugging - CheckJNI(2)

(*) 테스트를 해 보았으나, 정상 동작하는 것인지 아닌지 확인할 길이 묘연^^
(*) 아래 두 방법은 동일한 내용이 아닌가 함 !!!

- **# adb shell setprop debug.checkjni 1**
 - → 현재 동작 중인 app에는 영향을 주지 못하며, 새로 구동되는 app에만 효력 발생함.
 - → logcat 내용 중에 "D Late-enabling CheckJNI" 라는 문구가 출력될 것임.
-
- # adb shell stop
 - **# adb shell setprop dalvik.vm.checkjni true** ← 위의 내용과 동일한 기능이 아닌가 싶음!
 - # adb shell setprop dalvik.vm.jniopts forcecopy ← 위의 명령과 비슷(?)한 것으로 보임.
 - # adb shell setprop dalvik.vm.enableassertions all ← non-system class에 대한 assertion을 enable한다.
 - # adb shell start

<실제로는 SIGSEGV로 죽을 문제인데, CheckJNI가 잡아낸 예>

```
W JNI WARNING: method declared to return 'Ljava/lang/String;' returned '[B'
W      failed in LJUnitest;.exampleJniBug
I "main" prio=5 tid=1 RUNNABLE
I   | group="main" sCount=0 dsCount=0 obj=0x40246f60 self=0x10538
I   | sysTid=15295 nice=0 sched=0/0 cgrp=default handle=-2145061784
I   | schedstat=( 398335000 1493000 253 ) utm=25 stm=14 core=0
I   at JniTest.exampleJniBug(Native Method)
I   at JniTest.main(JniTest.java:11)
I   at dalvik.system.NativeStart.main(Native Method)
I
E VM aborting
```

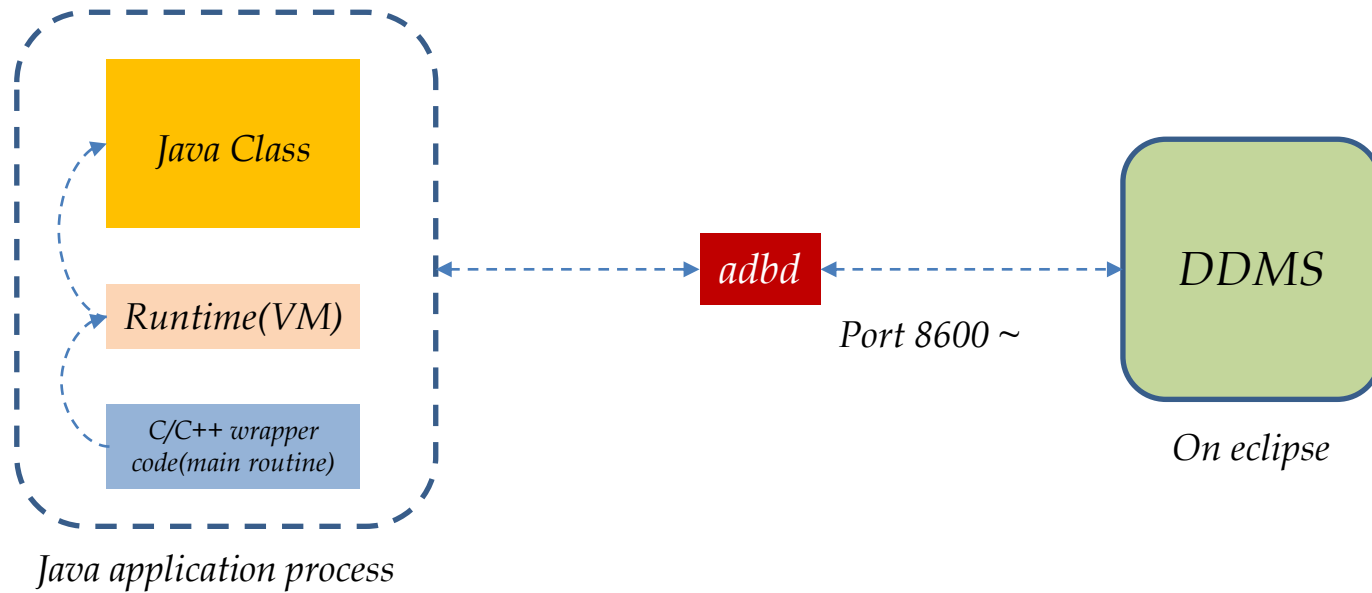
2.6 Valgrind - 메모리 누수, 비정상 메모리 위치 접근, 초기화 안된 영역 읽기, double free, 비정상 stack 조작 등 감지

- **TODO**

- *Build는 했으나, 동작상에 문제가 있음^^*

3. Java Code Debugging

3.1 DDMS(1)



3.1 DDMS(2)

[주의 사항] 아래 과정은 *system memory*가 충분히 있을 경우에만 테스트가 가능하다^^.
내 PC(2GB memory)에서 현재 *running* 중인, *system_server*를 *attaching*하는 것까지는 확인했는데,
*Out of memory*를 뿌리면서 *eclipse*가 죽어 버린다헐... 메모리 늘려줘...

- 1. # export ANDROID_BUILD_TOP=~/.YOUR_PATH/android
- ➔ 자신의 환경에 맞게 적절히 지정
- 2. # ./cts/development/ide/eclipse/genclasspath.sh 실행
- 3. 이클립스에서 File -> New Java Project -> Use default location 체크를 없애고, Browse 버튼을 눌러서 android root 디렉토리 설정 후, Finish 버튼 선택
- ➔ 당연한 얘기지만, 사전에 Eclipse, Android SDK 등은 모두 설치해 두었어야 함
- ➔ 이 단계는 android 전체를 project로 만들므로 다소 시간이 걸림.
- 4. 새로 생성된 프로젝트의 코드 중에 디버깅 할 코드에 Breakpoint 설정
- ➔ *system_server*를 debugging하고자 한다면, 관련 코드 중 하나를 선택하여 breakpoint를 지정해야 함.
- 5. Package Explorer에서 새로 생성된 프로젝트에 대해 마우스 오른쪽 클릭 후 Debug As -> Debug configurations 클릭
- 6. Eclipse 버전마다 약간의 차이는 있을 수 있으나, Remote Java Application을 선택한 후, Host는 localhost, Port는 8600 또는 8700으로 입력 후 Debug 버튼 클릭
- 7. 에러 메시지가 나오나, Proceed를 눌러 진행
- ➔ VM에 연결할 수 없다는 popup이 뜰 경우, 다른 창에서 `sudo adb shell` 하여 단말의 *adbd*를 새로 띄워줌.
- 8. DDMS로 보면, 해당 프로세스에 녹색의 debug 아이콘이 붙어 나오게 됨.
 - ➔ 예를 들어 내가, *system_server* 관련 코드에 breakpoint를 지정했다면, *system_process*에 녹색의 debug icon이 표시되게 됨.

3.1 DDMS (3)

system_process가 system_server를 의미함

The screenshot displays the DDMS interface within the Eclipse IDE. The title bar indicates the current project is `android/frameworks/base/services/java/com/android/server/am/ActivityManagerService.java`. The 'Devices' tab is active, showing a list of virtual devices. The 'system_process' is highlighted in orange, indicating it is the selected process. The 'Heap' tab is also visible, showing heap statistics. The 'LogCat' tab at the bottom shows a list of log messages.

Devices Tab:

Name	State	Version
9990392000495	Online	2.3.4, debug
system_process	180	8600 / 8700
com.android.systemui	205	8601

Heap Tab:

Heap updates are NOT ENABLED f

ID	Heap Size	Allocated	Free	% Used	# Objects
1	11.320 MB	6.621 MB	4.700 MB	58.49%	125.741

LogCat Tab:

Time	PID	Application	Tag	Text
11-04 10:45:52.9	495		wpa_supplicant	[WIFI-WPS] wpa_supplicant_ctrl_iface_pro
11-04 10:45:52.9	495		wpa_supplicant	CMD: DRIVER LINKSPEED
11-04 10:45:52.9	495		wpa_supplicant	[WIFI-WPS] @@@@ DRIVER LINKSPEED @
11-04 10:45:52.9	495		wpa_supplicant	wpa_driver_priv_driver_cmd LINKSPEED len
11-04 10:45:52.9	495		wpa_supplicant	wpa_driver_priv_driver_cmd LinkSpeed 54

3.1 DDMS (4)

- DDMS 사용법 관련하여 보다 자세한 사항은 아래 site를 참고하기 바람.
➔ <http://blog.naver.com/jang2818/20078863663>

3.2 VM Heap & Native Heap

- # setprop dalvik.vm.heapsize 48m ← VM heap size 변경하기
- → 애플을 늘리는게 좋을까 줄이는게 좋을까 ?
- **TODO**
 - (*) Java Heap에서 out of memory가 날 경우는 매우 드물며, 난다면 application code의 로직이 잘 못되었을 가능성이 높다.
 - → Context 관련 leak issue가 많다는군 ...
 - (*) 반면, Native Heap의 경우에 out of memory가 날 가능성이 더 큰데 ...
 - (*) Java Heap size는 해당 process 별로 제한되며 ..
 - (*) Native Heap size는 system 전체적으로 그 크기가 제한되어 있다... thread 1개의 size를 줄여야 하며, thread 개수가 늘어나는 것을 방지해야 한다...

3.3 Java Code에 stack trace code 추가하기

```
// 20110324 branden.lee, printing current stack. Start
StringBuffer stacktrace = new StringBuffer();
StackTraceElement[] stackTrace = new Exception().getStackTrace();
for(int x=0; x<stackTrace.length; x++)
{
    stacktrace.append(stackTrace[x].toString() + " ");
}
Log.e(TAG, "stacktrace");
Log.e(TAG, stacktrace.toString());
// 20110324 branden.lee, printing current stack. End
```

(*) *android framework code*를 *debugging*하는 방법으로 3.1절에서 소개한 방법을 사용할 수도 있으나, 방법이 매우 번거로울 수 있으므로, *debugging*하고자 하는 파일에 위의 코드를 삽입하여 *function call* 흐름을 분석할 수도 있겠다^^

4. Kernel Code Debugging

4.1 dmesg

- dmesg
- ➔ *kernel log 출력*(누구나 아는 내용)
- `cat < /proc/kmsg`
➔ *serial cable이 없을 경우에 유용한 방법*
- `adb shell "cat < /proc/kmsg" | grep "binder"`
➔ *Kernel log 중, 특정 string만을 추출하는 예*

4.2 addr2line or objdump(1)

Unable to handle kernel paging request at virtual address c2800000

pgd = c0004000

[c2800000] *pgd=21c14011, *pte=00000000, *ppte=00000000

Internal error: Oops: 807 [#1]

Modules linked in:

CPU: 0 Not tainted (2.6.24 #135)

PC is at start_kernel+0x2b0/0x350

LR is at 0xc033a3a4

pc : [<c0008ae4>] lr : [<c033a3a4>] psr: 60000053

sp : c0335fd4 ip : c033a3a4 fp : c0335ff4

r10: 2002132c r9 : 41069265 r8 : 20021360

r7 : c0337cd4 r6 : c0022f28 r5 : c035626c r4 : c0355e24

r3 : 12345678 r2 : c2800000 r1 : 00000001 r0 : c02ebf70

Flags: nZCv IRQs on FIQs off Mode SVC_32 ISA ARM Segment kernel

Control: 0005317f Table: 20004000 DAC: 00000017

Process swapper (pid: 0, stack limit = 0xc0334258)

Stack: (0xc0335fd4 to 0xc0336000)

5fc0: c0008470 c0022f28 00053175

5fe0: c0356728 c0022f24 00000000 c0335ff8 20008034 c0008844 00000000 00000000

Backtrace:

[<c0008834>] (start_kernel+0x0/0x350) from [<20008034>] (0x20008034)

r6:c0022f24 r5:c0356728 r4:00053175

Code: eb01240a e5942000 e59f3094 e59f0094 (e5823000)

--[end trace ca143223eefdc828]--

Kernel panic - not syncing: Attempted to kill the idle task!



Kernel Oops message 예임

4.2 addr2line or objdump(2)

- addr2line & objdump 사용하여 위치 추적하기
 - ➔ # arm-eabi-addr2line -f -e ./vmlinux 0xFFFFXXXX
 - ➔ # arm-eabi-objdump -d ./vmlinux > aaa

(*) Oops 메시지 중 "PC is at WWWW + 0xxxxx/0yyyy" 부분을 주목.

위의 objdump 결과로 얻은 파일에서 WWWW 함수를 찾고, 다시 0xxxxx offset 위치의 코드가 문제의 코드임. 0yyyy는 WWWW 함수의 크기를 나타냄.

(*) kernel debugging을 위해서는 반드시 vmlinux가 필요하며, 이는
out/target/product//obj/KERNEL_OBJ 아래에서 얻을 수 있다.

4.2 addr2line or objdump(3) - 또 다른 예

- **<Kernel log>**
- ```
<6>[319816.736590] sdio_al:sdio_al_sdio_remove: sdio card 4 removed.
<6>[319816.737720] mmc4: card 0002 removed
<0>[319817.405475] Restarting system with command 'androidpanic'.
<5>[319817.406543] Going down for restart now
<3>[319817.406726] allydrop android panic!!!in_panic:0
<0>[319817.406878] Kernel panic - not syncing: android framework error
<0>[319817.406970]
<4>[319817.407245] [<c01083d4>] (unwind_backtrace+0x0/0x164) from [<c07297e8>]
(panic+0x6c/0x11c)
<4>[319817.407550] [<c07297e8>] (panic+0x6c/0x11c) from [<c0178bbc>] (arch_reset+0x120/0x2c8)
<4>[319817.407824] [<c0178bbc>] (arch_reset+0x120/0x2c8) from [<c0102acc>]
(arm_machine_restart+0x40/0x6c)
<4>[319817.408160] [<c0102acc>] (arm_machine_restart+0x40/0x6c) from [<c0102964>]
(machine_restart+0x20/0x28)
<4>[319817.408465] [<c0102964>] (machine_restart+0x20/0x28) from [<c01b8db4>]
(sys_reboot+0x1b4/0x21c)
<4>[319817.408740] [<c01b8db4>] (sys_reboot+0x1b4/0x21c) from [<c01012c0>]
(ret_fast_syscall+0x0/0x30)
```

### **<addr2line 실행 결과>**

pz1944@mars:~/HA\$ /home/android/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-  
addr2line -f -e ./vmlinux 0xc01083d4

unwind\_backtrace

/home/android/kernel/arch/arm/kernel/unwind.c:351

➔ unwind.c 파일의 351 line에 위치한, unwind\_backtrace( ) function에서 죽음 !!!

(\*) 실제로 위의 내용은 사용 방법을 보여주기 위한 예일 뿐이다.

## 4.3 gdb(1)

```
Unable to handle kernel paging request for data at address 0x33343a31
Faulting instruction address: 0xc50659ec
Oops: Kernel access of bad area, sig: 11 [#1]
tpsslr3
Modules linked in: datalog(P) manet(P) vnet wlan_wep wlan_scan_sta ath_rate_samp
NIP: c50659ec LR: c5065f04 CTR: c00192e8
REGS: c2aff920 TRAP: 0300 Tainted: P (2.6.25.16-dirty)
MSR: 00009032 CR: 22082444 XER: 20000000
DAR: 33343a31, DSISR: 20000000
TASK = c2e6e3f0[1486] 'datalogd' THREAD: c2afe000
GPR00: c5065f04 c2aff9d0 c2e6e3f0 00000000 00000001 00000001 00000000 0000b3f9
GPR08: 3a33340a c5069624 c5068d14 33343a31 82082482 1001f2b4 c1228000 c1230000
GPR16: c60f0000 000004a8 c59abbe6 0000002f c1228360 c340d6b0 c5070000 00000001
GPR24: c2aff9e0 c5070000 00000000 00000000 00000003 c2cc2780 c2affae8 0000000f
NIP [c50659ec] mesh_packet_in+0x3d8/0xdac [manet]
LR [c5065f04] mesh_packet_in+0x8f0/0xdac [manet]
Call Trace:
[c2aff9d0] [c5065f04] mesh_packet_in+0x8f0/0xdac [manet] (unreliable)
[c2affad0] [c5061ff8] IF_netif_rx+0xa0/0xb0 [manet]
[c2affae0] [c01925e4] netif_receive_skb+0x34/0x3c4
[c2affb10] [c60b5f74] netif_receive_skb_debug+0x2c/0x3c [wlan]
[c2affb20] [c60bc7a4] ieee80211_deliver_data+0x1b4/0x380 [wlan]
[c2affb60] [c60bd420] ieee80211_input+0xab0/0x1bec [wlan]
[c2affbf0] [c6105b04] ath_rx_poll+0x884/0xab8 [ath_pci]
[c2affc90] [c018ec20] net_rx_action+0xd8/0x1ac
[c2affcb0] [c00260b4] __do_softirq+0x7c/0xf4
[c2affce0] [c0005754] do_softirq+0x58/0x5c
[c2affcf0] [c0025eb4] irq_exit+0x48/0x58
[c2affd00] [c000627c] do_IRQ+0xa4/0xc4
```

## 4.3 gdb(2)

- # arm-eabi-gdb ./vmlinux
- # (gdb) info line 0xc50659ec
  - ➔ 문제가 되는 부분의 함수를 출력할 것임.

(\*) *kernel Oops* 메시지를 보고, 문제가 발생한 위치를 추적하는 방법임.  
(\*) 단말에 돌고 있는 *kernel*을 *attach*하는 방법은 아님^^.

## 4.4 kgdb

- **[TODO]** 단, 이 방법은           의 경우에는 정상 동작하지 않는 듯.

### Debugging an ARM Linux Kernel with GDB

Enable kernel debugging in the menuconfig under Kernel Hacking -> 'Compile the kernel with debug info'

```
$ make menuconfig
```

Make the kernel

```
$ make
```

Start the emulator with debugging enabled in QEMU using the -s switch

```
$ emulator 'normal arguments' -qemu -s
```

Then start gdb by running the following (arm-eabi-gdb comes with Android)

```
$ arm-eabi-gdb ~/src/android/kernel/vmlinux
```

This will launch gdb and load the Linux kernel debugging symbols.

To connect gdb to the qemu gdbserver started by using the -s switch and put a breakpoint in sys\_open run

```
$(gdb) target remote localhost:1234
$(gdb) b sys_open
$(gdb) cont
```

(\*) 위에 기술된 내용은 적절치 않음. 다만, kgdb 관련하여 추후 테스트 목적으로 관련 내용을 정리한 것임^^

(\*) kgdb 관련해서는 [sourceforge.net/apps/mediawiki/usbdevicesuppor/index.php?title=Manual](http://sourceforge.net/apps/mediawiki/usbdevicesuppor/index.php?title=Manual) 참조

## 4.5 kprobe or jprobe를 사용한 실시간 debugging(1)

- kprobe/jprobe/kretprobe

- ➔ Kernel code에 원하는 작업을 동적으로 추가할 수 있는 강력한 기법
- ➔ 동작 중인 kernel 상에서 테스트 가능하며, 코드 수정이 불필요한 매우 유용한 debugging 방식
- ➔ Debugging 하고자 하는 특정 함수의 임의 위치에 probe함수를 삽입할 수 있음(kprobe – chip dependent한 부분이 있어서 사용이 약간 불편함).
- ➔ Debugging 하고자 하는 특정 함수의 앞 부분(jprobe의 경우)에 probe 함수를 삽입(hooking) 하여, 전달되는 argument의 값을 출력하거나, 값을 수정할 수 있음(kprobe에 비해 argument handling이 용이함).
- ➔ 이 밖에도 함수가 return되는 시점에 probe를 삽입할 수 있는 kretprobe도 있음.

(\*) 물론, kernel code에 printk 문을 집어 넣어 직접 debugging하는 방법도 있겠지만, Kernel code를 수정하지 않으면서도, run-time에 특정 함수를 debugging할 수 있는 효과적인 방법임

(\*) debugging하고자 하는 code가 복잡하고 난해하여, 함수의 흐름을 이해하기 어려운 경우에도 매우 유용함.

(\*) kprobe/jprobe 관련 자세한 사항은 doc/Documentation/kprobes.txt 파일 참조

(\*) kprobe/jprobe를 사용하려면, CONFIG\_KPROBES(kernel menuconfig 제일 처음 항목)를 enable시켜야 함.

(\*) kernel/samples/kprobes 아래에 관련 sample code 있음 ^^.

## 4.5 kprobe or jprobe를 사용한 실시간 debugging(2)

```
int register_kprobe(struct kprobe *p);

int register_jprobe(struct jprobe *p);

void unregister_kprobe(struct kprobe *p);

void unregister_jprobe(struct jprobe *p);
```

Table 1. Kernel probes management functions

```
struct kprobe {
 /* elided fields for internal state information */

 kprobe_opcode_t *addr;
 kprobe_pre_handler_t pre_handler;
 kprobe_post_handler_t post_handler;
 kprobe_fault_handler_t fault_handler;

 /* elided fields for internal state information */
};
```

Listing 1. kprobe data structure

```
struct jprobe {
 struct kprobe kp;
 kprobe_opcode_t *entry; /* probe handling code to jump to */
};
```

Listing 2. jprobe data structure

(\*) *kernel/include/linux/kprobes.h* 에 *kprobe/jprobe* 관련 data structure가 정의되어 있음.

## 4.5 kprobe or jprobe를 사용한 실시간 debugging(3) - kprobe code 예

```
• #include <linux/module.h>
 #include <linux/init.h>
 #include <linux/kprobes.h>
 #include <linux/kallsyms.h>

 #define PRCUR(t) printk (KERN_INFO "current->comm=%s, current->pid=%d\n", t->comm, t->pid);

 static char *name = "do_fork";
 module_param(name, charp, S_IRUGO);

 static struct kprobe kp;

 static int handler_pre(struct kprobe *p, struct pt_regs *regs)
 {
 dump_stack();
 printk(KERN_INFO "pre_handler: p->addr=0x%p\n", p->addr);
 PRCUR(current);
 return 0;
 }

 static void handler_post(struct kprobe *p, struct pt_regs *regs,
 unsigned long flags)
 {
 printk(KERN_INFO "post_handler: p->addr=0x%p\n", p->addr);
 PRCUR(current);
 }

 static int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr)
 {
 printk(KERN_INFO "fault_handler:p->addr=0x%p\n", p->addr);
 PRCUR(current);
 return 0;
 }

 static int __init my_init(void)
 {
 /* set the handler functions */

 kp.pre_handler = handler_pre;
 kp.post_handler = handler_post;
 kp.fault_handler = handler_fault;
 kp.symbol_name = name;

 if (register_kprobe(&kp)) {
 printk(KERN_INFO "Failed to register kprobe, quitting\n");
 return -1;
 }

 printk(KERN_INFO "Hello: module loaded at 0x%p\n", my_init);

 return 0;
 }

 static void __exit my_exit(void)
 {
 unregister_kprobe(&kp);
 printk(KERN_INFO "Bye: module unloaded from 0x%p\n", my_exit);
 }

 module_init(my_init);
 module_exit(my_exit);
```

← debugging을 원하는 함수명으로 교체 !!!

(\*) 위의 코드를 build하여 생성한 모듈을 단말에 insmod하게 되면, do\_fork( ) 함수가 호출되기 직전 및 직후에, 원하는 action을 취할 수 있게 됨. fault\_handler는 kprobe가 실행되는 도중 exception이 발생한 경우에 호출됨.

## 4.5 kprobe or jprobe를 사용한 실시간 debugging(4) - jprobe code 예

- ```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

static long mod_timer_count = 0;

static void mod_timer_inst(struct timer_list *timer, unsigned long expires)
{
    mod_timer_count++;
    if (mod_timer_count % 10 == 0)
        printk(KERN_INFO "mod_timer_count=%ld\n", mod_timer_count);
    dump_stack();
    jprobe_return();
}
```
 - ```
static struct jprobe jp = {
 .kp.addr = (kprobe_opcode_t *) mod_timer,
 .entry = (kprobe_opcode_t *) mod_timer_inst,
};

static int __init my_init(void)
{
 register_jprobe(&jp);
 printk(KERN_INFO "plant jprobe at %p, handler addr %p\n", jp.kp.addr,
 jp.entry);
 return 0;
}

static void __exit my_exit(void)
{
 unregister_jprobe(&jp);
 printk(KERN_INFO "jprobe unregistered\n");
 printk(KERN_INFO "FINAL:mod_timer_count=%ld\n", mod_timer_count);
}

module_init(my_init);
module_exit(my_exit);
```
- ◀ mod\_timer 함수가 불리울 때마다, stack trace dump를 시도한다.
- ◀ debugging을 원하는 함수명으로 교체 !!!

(\*) 위의 코드를 build하여 생성한 모듈을 단말에 insmod하게 되면, timer\_inst( ) 함수가 호출될 때마다, 원하는 action을 취할 수 있게 됨.

(\*) 예를 들어, kernel의 특정 함수에서 죽는 문제가 있는데, 누가 문제를 발생시키는지 모를 경우, probe 함수내에 dump\_stack() 함수를 추가하면 debugging에 많은 도움이 될 것이다.



## 5. TODO

- Valgrind
- VM/Native Heap
- kgdb

*Thanks a lot !*

