

*Android **Debug Debug Debug***

: logcat, gdb, addr2line/objdump, DDMS, jdb, kgdb, kprobe/kgtp



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 (<i>addr2line</i>)
0.3	이 충 한	11/08/2011 (<i>kprobe/jprobe</i>)
0.4	이 충 한	11/09/2011 (<i>JNI</i>)
0.5	이 충 한	11/29/2011 (<i>maps, jdb, ANR</i>)
0.6	이 충 한	12/05/2011 (<i>kgtp, kgdb</i>)

목차

- **1. Basic Tools:** *adb, logcat, dumpstate, dumphsys*
- **2. C/C++ Code Debugging**
 - 2.1 *strace*
 - 2.2 *gdb & gdbserver*
 - 2.3 *addr2line & objdump – shared object 에 대한 backtrace 정보가 있는 경우*
 - 2.4 */proc/pid/maps – shared object 에 대한 backtrace 정보가 없는 경우*
 - 2.5 *libc.debug.malloc*
 - 2.6 *JNI Debugging – CheckJNI*
 - 2.7 *Valgrind(TODO)*
- **3. Java Code Debugging**
 - 3.1 *DDMS*
 - 3.2 *jdb*
 - 3.3 *VM Heap*
 - 3.4 *ANR 에 관하여*
- **4. Kernel Code Debugging**
 - 4.1 *dmesg*
 - 4.2 *addr2line & objdump*
 - 4.3 *gdb*
 - 4.4 *kgdb (Only Doc)*
 - 4.5 *kprobe/jprobe 를 사용한 실시간 debugging*
 - 4.6 *kgtp*

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.

(*) 본 문서의 내용은 *ICS4.01*에도 그대로 적용 가능할 것으로 보인다.

1. Basic Tools: adb, logcat, dumpstate, dumpsys(1)

- `# adb shell logcat`
- `# adb shell dumpstate > state.txt`
→ *system, status, counts, and statistics* 정보를 dump
- `# adb shell dumpsys > sys.txt`
- `# adb shell dumpsys 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 dumpsys meminfo pid` ← pid를 갖는 process의 memory 사용정보 출력

(*) 기본적인 명령이라, 자세한 의미는 별도로 기술하지 않았음.

1. Basic Tools: adb, logcat, dumpstate, dumpsys(2)

- `# adb shell ps -t`
→ Thread를 모두 출력
- `# adb shell ps -x`
→ Time 정보를 함께 출력
- `# adb shell ps -p`
→ Priority 정보 출력

</proc/<pid> 디렉토리 아래 유용한 정보>

→ process 별 매우 유용한 정보가 포함되어 있음.

- 1) task : thread 별 정보
- 2) fd: 해당 process가 사용하는 file descriptor 정보(현재 open되어 있는 정보)
- 3) maps: 해당 process를 위한 가상 memory 맵(PC값을 통해 debugging시 유용)
- 4) smaps: maps 정보 보다 상세한 정보 출력(memory 관련)
- 5) ...

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.

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

(*) *strace*로 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) – shared object 에 대한 backtrace 정보가 있는 경우

• <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 fffff84 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

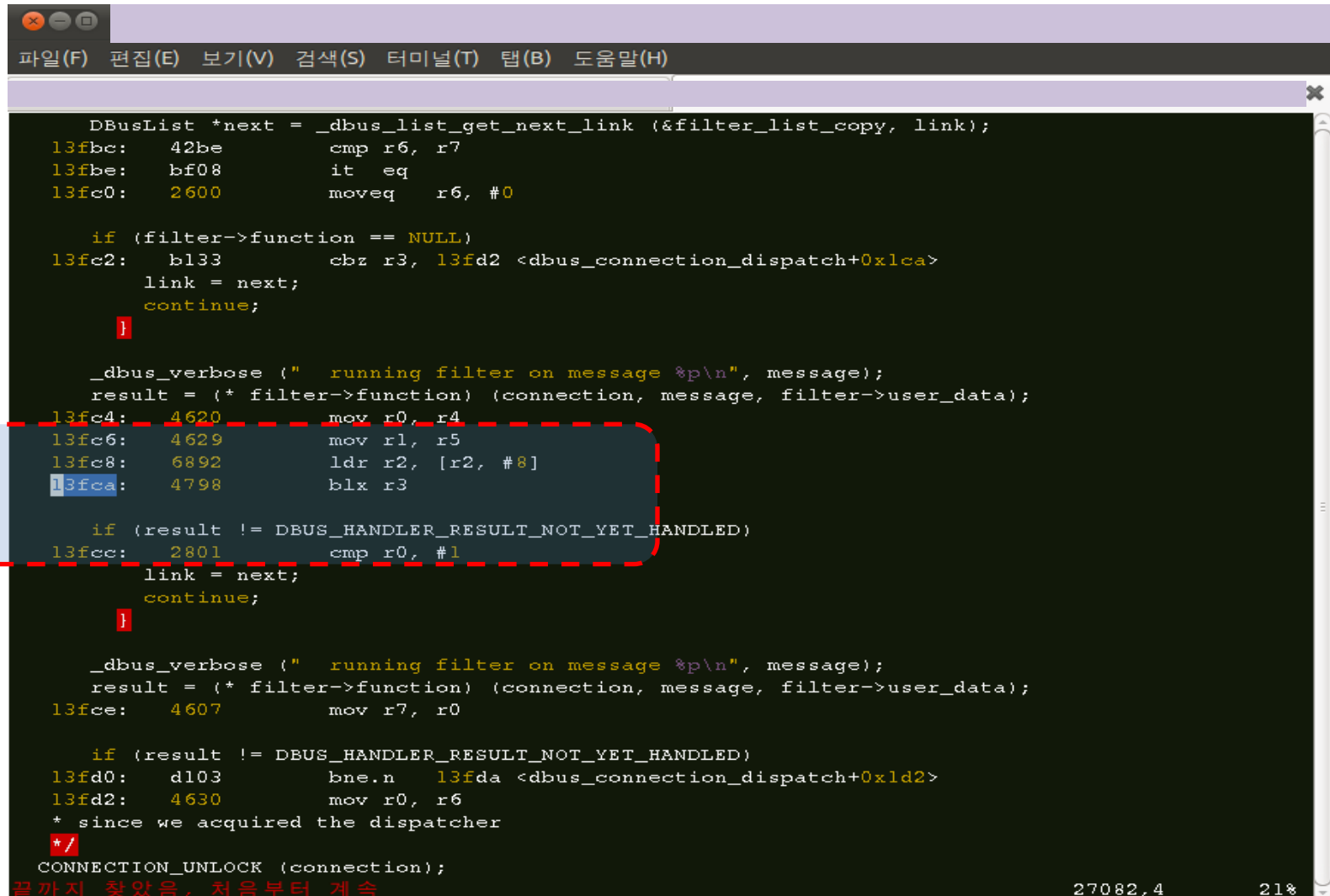
(*) -C option을 사용하면 function name이 깔끔하게 출력될 것임.

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;
}

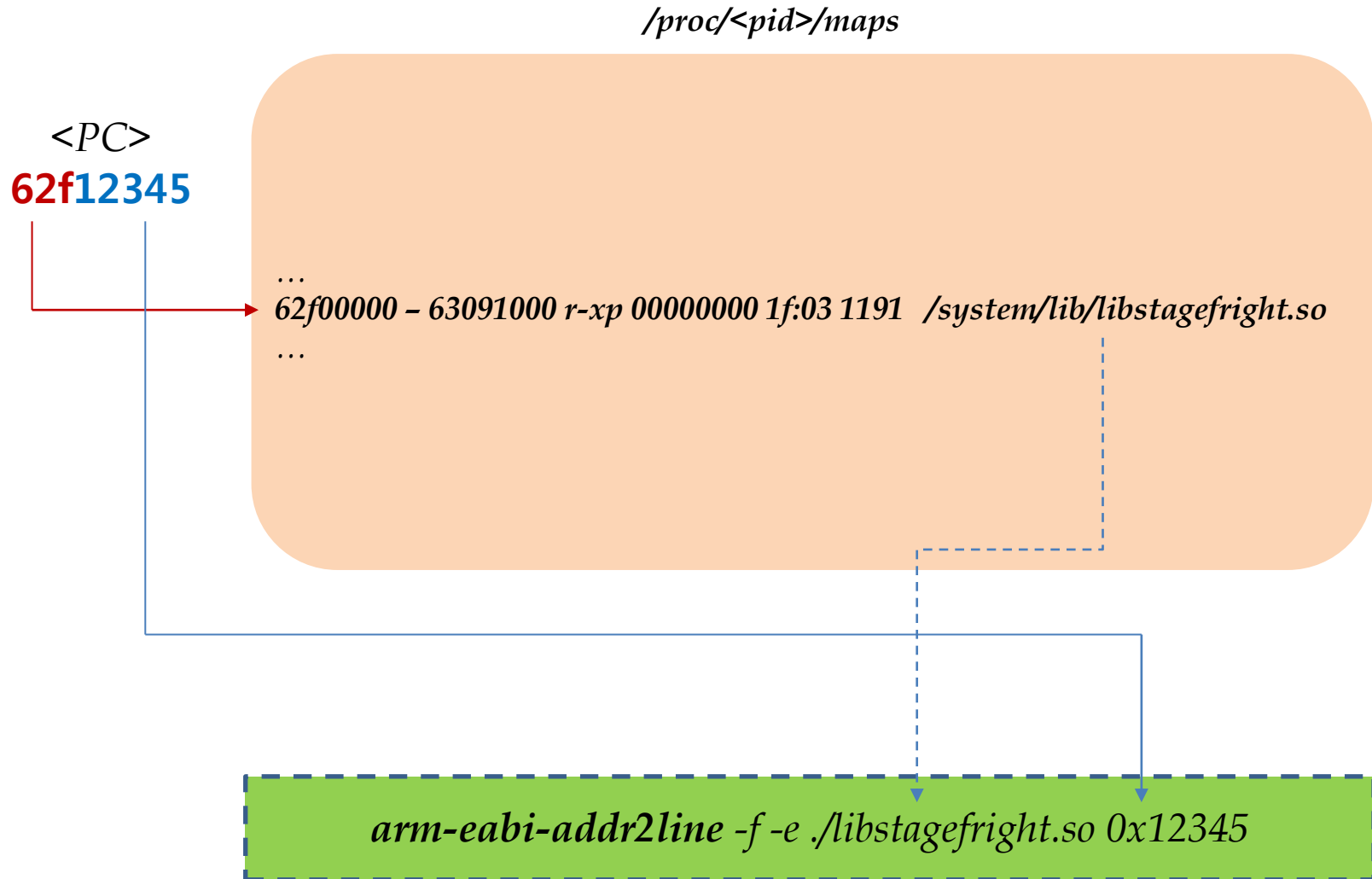
    _dbus_verbose ("    running filter on message %p\n", message);
    result = (* filter->function) (connection, message, filter->user_data);
13fce: 4607      mov r7, r0

    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

2.4 /proc/pid/maps(1) - shared object에 대한 backtrace 정보가 없는 경우



2.4 /proc/pid/maps(2) - shared object에 대한 backtrace 정보가 없는 경우

<조건>

→ Segmentation fault 발생 시, shared object에 대한 backtrace 정보는 없으나, 죽은 thread 이름과 PC(program counter) 정보가 있다.

<Point>

→ PC 값은 virtual address 정보를 담고 있으며, 앞 3자리는 shared object가 load되는 가상 memory map 주소 정보를 가리키며, 나머지 뒷자리 숫자들(5자리)은 shared object내의 offset을 의미한다.

<단계 1>

- 죽은 thread 이름으로 이와 연관된 process 이름을 유추한 뒤, ps로 해당 process의 pid를 알아낸다.
- 이후, "cat /proc/<pid>/maps" 명령으로 해당 process의 memory map을 확인한다.

<단계 2>

- PC의 처음 3자리 수를 이용하여, 위에서 얻은 /proc/<pid>/maps 정보 중에서 문제가 되는 shared object(library)를 찾아낸다.

<단계 3>

- 위에서 찾은 shared object에 대해, arm-eabi-objdump(addr2line도 가능)과 PC의 남은 숫자를 활용하여 문제가 되는 point를 찾아낸다(2.3 절 내용)^.^.

2.4 /proc/pid/maps(3)

문제가 되는 thread 이름(정보)

```
- Seg Fault Message:  
PV author: unhandled page fault (11) at 0x00000004, code 0x017  
pgd = ccf8000  
[00000004] *pgd=8cbe1031, *pte=00000000, *ppte=00000000  
Pid: 9691, comm: PV author  
CPU: 0 - Not tainted (2.6.29-omap1 #20)  
PC is at 0x81b23140  
LR is at 0x81b23049
```

Program Counter

PV author runs in context of Media server process (PID: 944)

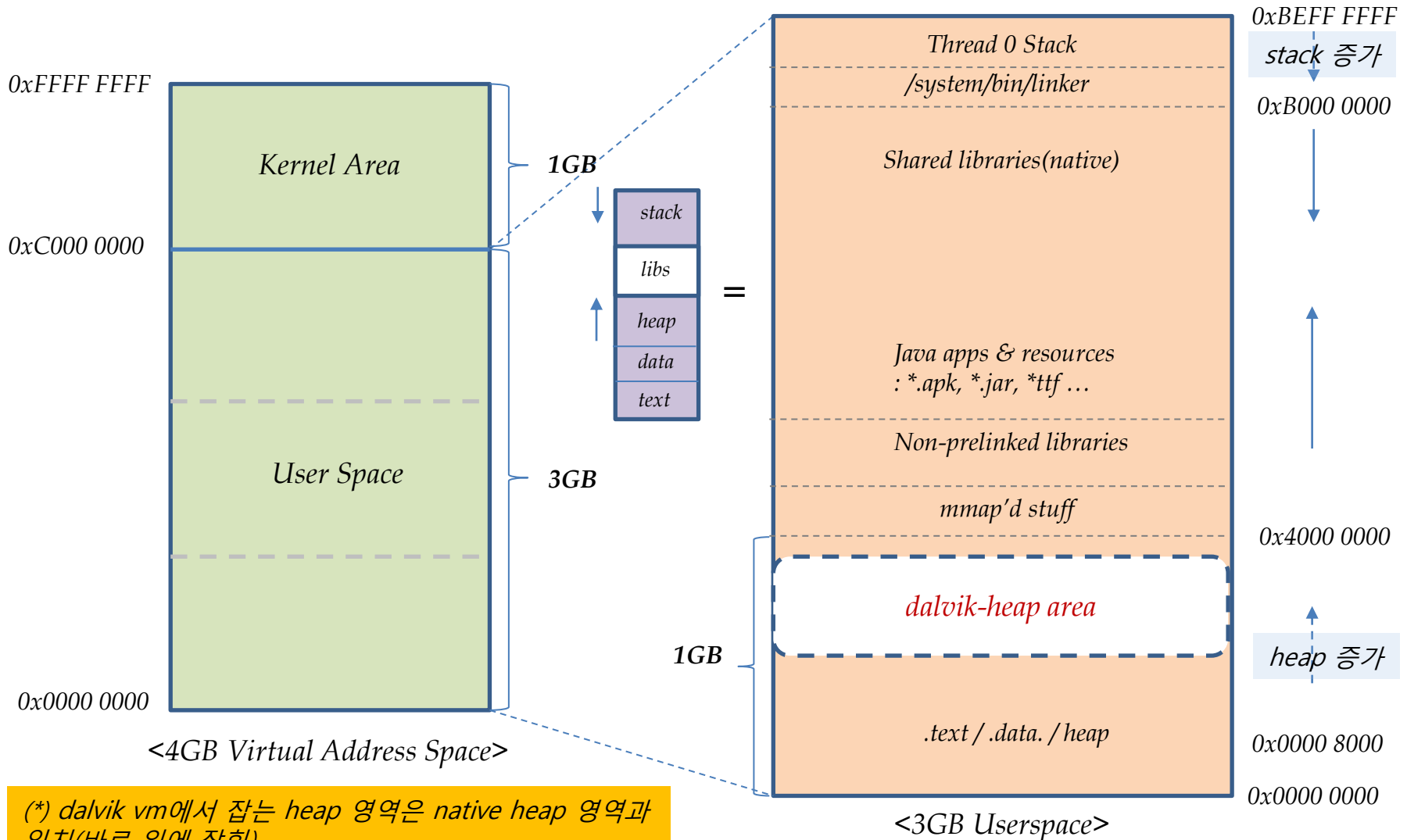
```
Using cat /proc/944/maps we can identify the 'so' loaded in this region  
81b00000-81b2a000 r-xp 00000000 b3:02 34574 /system/lib/libOMX.TI.Video.encoder.so  
So appears to be in OMX TI Video encoder
```

/proc/<pid>/maps 내용 중, PC로 mapping되는 라인의 예

- (*) Gingerbread의 경우는 아니지만, 적당한 로그가 없어, 위의 내용으로 대신함.
- (*) 로그 내용으로 부터, 죽은 process를 적절히 유추해야 함 → pid 획득
- (*) /proc/<pid>/maps 정보와 PC 값으로 부터, 문제의 shared library 획득 !
- (*) 문제의 library를 알아냈으니, 나머지는 2.3절에서 설명한 방법(addr2line/objdump)을 활용 !

2.4 /proc/pid/maps(4) - android virtual address memory map

(*) /proc/<pid>/maps를 이해하기 위해서는 android virtual memory map을 이해할 필요가 있다^^. 보다 자세한 사항은 build/core/prelink-linux-arm.map 파일 참조 !
(*) 아래 User space map 정보는 prelink-linux-arm.map을 참조하여 작성한 것일 뿐, 실제 동작중인 내용(주소 값)은 다르다. (단, 각 영역의 순서/위치는 일치함)



2.4 /proc/pid/maps(5) - *ex) mediaserver*

- 1) .text(code), .data(global var), heap 영역
- 2) code segment는 0x0000 8000에서 시작함.

```
# cat maps
00008000-00009000 r-xp 00000000 1f:03 457 /system/bin/mediaserver
00009000-0000a000 rw-p 00001000 1f:03 457 /system/bin/mediaserver
0000a000-0012f000 rw-p 00000000 00:00 0 [heap]
10000000-10001000 ---p 00000000 00:00 0
10001000-10100000 rw-p 00000000 00:00 0
2aab000-2aab3000 r--s 00000000 00:0c 917 /dev/__properties__ (deleted)
2aab3000-2aab4000 r--p 00000000 00:00 0
2aab4000-2abb2000 r--p 00000000 00:0c 716 /dev/binder
2abb2000-2abb3000 ---p 00000000 00:00 0
2abb3000-2acb2000 rw-p 00000000 00:00 0
2acb2000-2acb3000 rw-s 00000000 1f:05 575 /data/qvss.conf
2acb3000-2acb4000 ---p 00000000 00:00 0
2acb4000-2adb3000 rw-p 00000000 00:00 0
2adb3000-2adb4000 ---p 00000000 00:00 0
2adb4000-2aeb3000 rw-p 00000000 00:00 0
2aeb3000-2afb3000 rw-s 00000000 00:04 1496 /dev/ashmem/decode_fd (deleted)
2afb3000-2afb4000 ---p 00000000 00:00 0
2afb4000-2b0b3000 rw-p 00000000 00:00 0
2b0b3000-2b1b3000 rw-s 00000000 00:04 1506 /dev/ashmem/decode_fd (deleted)
2b1b3000-2b2b3000 rw-s 00000000 00:04 1511 /dev/ashmem/decode_fd (deleted)
2b2b3000-2b3b3000 rw-s 00000000 00:04 1513 /dev/ashmem/decode_fd (deleted)
2b3b3000-2b4b3000 rw-s 00000000 00:04 1521 /dev/ashmem/decode_fd (deleted)
2b4b3000-2b5b3000 rw-s 00000000 00:04 1522 /dev/ashmem/decode_fd (deleted)
2b5b3000-2b6b3000 rw-s 00000000 00:04 1526 /dev/ashmem/decode_fd (deleted)
2b6b3000-2b7b3000 rw-s 00000000 00:04 1524 /dev/ashmem/decode_fd (deleted)
2b7b3000-2b8b3000 rw-s 00000000 00:04 1527 /dev/ashmem/decode_fd (deleted)
2b8b3000-2b9b3000 rw-s 00000000 00:04 1529 /dev/ashmem/decode_fd (deleted)
2b9b3000-2bab3000 rw-s 00000000 00:04 1541 /dev/ashmem/decode_fd (deleted)
2bab3000-2bbb3000 rw-s 00000000 00:04 1990 /dev/ashmem/AudioFlinger::Client (deleted)
)
2bbb3000-2bbb4000 ---p 00000000 00:00 0
2bbb4000-2bcb3000 rw-p 00000000 00:00 0
2bcb3000-2bcb4000 rw-s 00000000 1f:05 575 /data/qvss.conf
```

(*) 여기는 주로, kernel 영역을 application에서 볼 수 있도록 mapping(mmap)해 주는 내용과 연관이 있음.

2.4 /proc/pid/maps(5) - *ex) mediaserver*

(*) native libraries. 이 앞부분(생략)은 java apk, jar, ttf 관련 파일 등이 위치함.

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```
6e300000-6e304000 r-xp 00000000 1f:03 1189 /system/lib/libnetutils.so
6e304000-6e305000 rw-p 00004000 1f:03 1189 /system/lib/libnetutils.so
6e400000-6e402000 r-xp 00000000 1f:03 1215 /system/lib/libwpa_client.so
6e402000-6e403000 rw-p 00002000 1f:03 1215 /system/lib/libwpa_client.so
6e700000-6e743000 r-xp 00000000 1f:03 1265 /system/lib/libdbus.so
6e743000-6e744000 rw-p 00043000 1f:03 1265 /system/lib/libdbus.so
6e900000-6e902000 r-xp 00000000 1f:03 1396 /system/lib/libbluedroid.so
6e902000-6e903000 rw-p 00002000 1f:03 1396 /system/lib/libbluedroid.so
6ee00000-6ee0e000 r-xp 00000000 1f:03 1187 /system/lib/liba2dp.so
6ee0e000-6ee0f000 rw-p 0000e000 1f:03 1187 /system/lib/liba2dp.so
6f000000-6f0ae000 r-xp 00000000 1f:03 1329 /system/lib/libcrypto.so
6f0ae000-6f0be000 rw-p 000ae000 1f:03 1329 /system/lib/libcrypto.so
6f0be000-6f0c0000 rw-p 00000000 00:00 0
6f400000-6f429000 r-xp 00000000 1f:03 1085 /system/lib/libssl.so
6f429000-6f42d000 rw-p 00029000 1f:03 1085 /system/lib/libssl.so
6f700000-6f714000 r-xp 00000000 1f:03 1235 /system/lib/libz.so
6f714000-6f715000 rw-p 00014000 1f:03 1235 /system/lib/libz.so
6f900000-6f90d000 r-xp 00000000 1f:03 1368 /system/lib/libcutils.so
6f90d000-6f90e000 rw-p 0000d000 1f:03 1368 /system/lib/libcutils.so
6f90e000-6f91d000 rw-p 00000000 00:00 0
6fa00000-6fa03000 r-xp 00000000 1f:03 1277 /system/lib/liblog.so
6fa03000-6fa04000 rw-p 00003000 1f:03 1277 /system/lib/liblog.so
6fb00000-6fb16000 r-xp 00000000 1f:03 1336 /system/lib/libm.so
6fb16000-6fb17000 rw-p 00016000 1f:03 1336 /system/lib/libm.so
6fc00000-6fc01000 r-xp 00000000 1f:03 1256 /system/lib/libstdc++.so
6fc01000-6fc02000 rw-p 00001000 1f:03 1256 /system/lib/libstdc++.so
6fd00000-6fd40000 r-xp 00000000 1f:03 1216 /system/lib/libc.so
6fd40000-6fd43000 rw-p 00040000 1f:03 1216 /system/lib/libc.so
6fd43000-6fd4e000 rw-p 00000000 00:00 0
70001000-70009000 r-xp 00001000 1f:03 470 /system/bin/linker
70009000-7000a000 rw-p 00009000 1f:03 470 /system/bin/linker
7000a000-70016000 rw-p 00000000 00:00 0
7e917000-7e938000 rw-p 00000000 00:00 0 [stack]
#
```

(*) linker

(*) stack(thread 0 stack)

2.4 /proc/pid/maps(5) - ex) com.android.phone

1) .text(code), .data(global var), heap 영역

2) code segment는 0x0000 8000에서 시작함. Phone appl의 main routine은 C로 된 app_process !!!

```
# cat maps
00008000-00009000 r-xp 00000000 1f:03 342 /system/bin/app_process
00009000-0000a000 rw-p 00001000 1f:03 342 /system/bin/app_process
0000a000-00341000 rw-p 00000000 00:00 0 [heap]
10000000-10001000 ---p 00000000 00:00 0
10001000-10100000 rw-p 00000000 00:00 0
2aaab000-2aab3000 r--s 00000000 00:0c 917 /dev/__properties__ (deleted)
2aab3000-2aab4000 r--p 00000000 00:00 0
2aab4000-2b106000 rw-p 00000000 00:04 1077 /dev/ashmem/dalvik-heap (deleted)
2b106000-2dcb4000 ---p 00652000 00:04 1077 /dev/ashmem/dalvik-heap (deleted)
2dcb4000-2dd7c000 rw-p 00000000 00:04 1078 /dev/ashmem/dalvik-bitmap-1 (deleted)
2dd7c000-2de44000 rw-p 00000000 00:04 1079 /dev/ashmem/dalvik-bitmap-2 (deleted)
2de44000-2dea9000 rw-p 00000000 00:04 1080 /dev/ashmem/dalvik-card-table (deleted)
2dea9000-2deac000 rw-p 00000000 00:00 0
2deac000-2dead000 ---p 00000000 00:04 1088 /dev/ashmem/dalvik-LinearAlloc (deleted)
2dead000-2e0eb000 rw-p 00001000 00:04 1088 /dev/ashmem/dalvik-LinearAlloc (deleted)
2e0eb000-2e3ac000 ---p 0023f000 00:04 1088 /dev/ashmem/dalvik-LinearAlloc (deleted)
2e3ac000-2e3bd000 rw-p 00000000 00:00 0
2e3bd000-2e3be000 r--s 001c6000 1f:03 1417 /system/framework/core.jar
2e3be000-2e7e6000 r--p 00000000 1f:05 573 /data/dalvik-cache/system@framework@core.
jar@classes.dex
2e7e6000-2e825000 rw-p 00000000 00:00 0
2e825000-2e826000 r--s 00046000 1f:03 1414 /system/framework/bouncycastle.jar
2e826000-2e8d3000 r--p 00000000 1f:05 576 /data/dalvik-cache/system@framework@bounc
ycastle.jar@classes.dex
2e8d3000-2e8d4000 r--s 0007d000 1f:03 1415 /system/framework/ext.jar
2e8d4000-2ea09000 r--p 00000000 1f:05 577 /data/dalvik-cache/system@framework@ext.j
ar@classes.dex
2ea09000-2ea0a000 r--s 0030c000 1f:03 1412 /system/framework/framework.jar
2ea0a000-2f178000 r--p 00000000 1f:05 1580 /data/dalvik-cache/system@framework@frame
work.jar@classes.dex
```

(*) dalvik vm에서 잡는 영역. dalvik-heap 및 dalvik-bitmap이 heap 영역임.

(*) 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 dlmalloc()** 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.

2.5 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.

2.5 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.*

2.6 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.

2.6 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
```

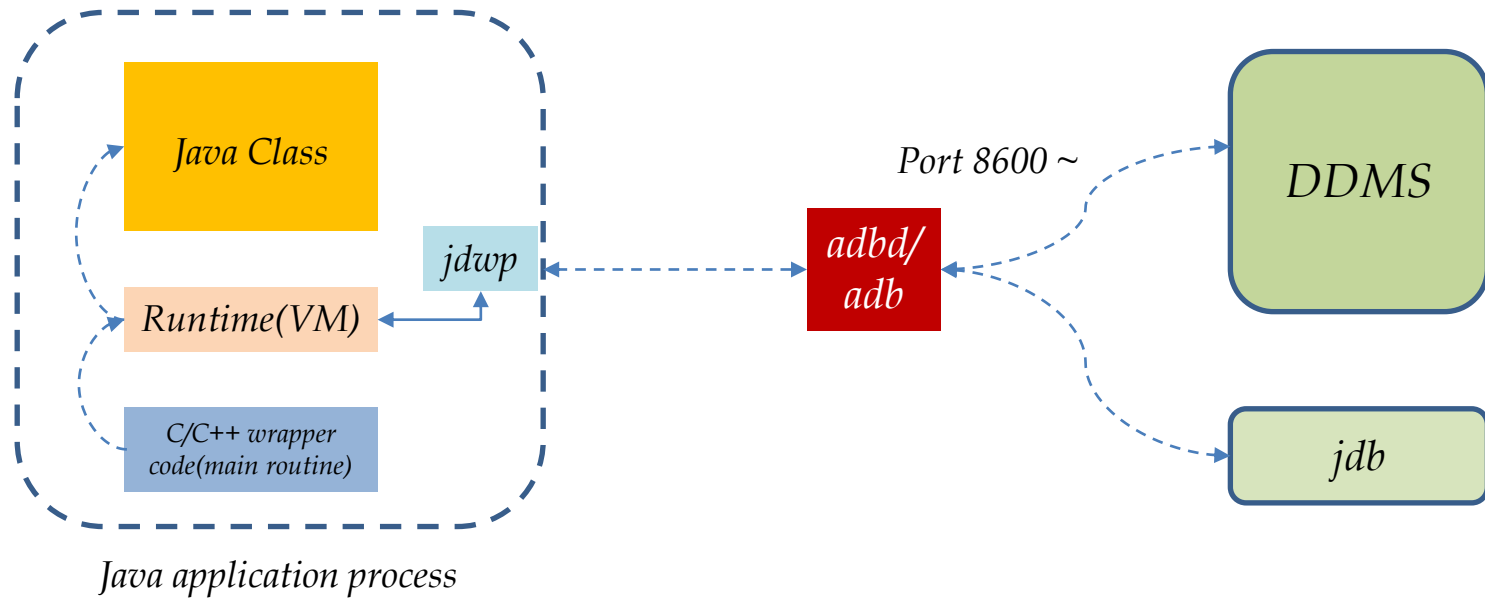
2.7 Valgrind - 메모리 누수, 비정상 메모리 위치 접근, 초기화 안된 영역 읽기, double free, 비정상 stack 조작 등 감지

- **TODO**

- *Build는 했으나, 동작상에 문제가 있음^^*

3. Java Code Debugging *: ddms, jdb, ANR...*

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 jdb(1) – java application debugging

```
$ adb shell ps -t
```

➔ *system_server* 의 pid가 171 임을 확인

```
$ adb forward tcp:8000 jdwp:171
```

➔ pid 171 process 관련 debug 정보를 tcp port 8000으로 forwarding 해줌.

➔ 8000 port는 다른 값을 사용할 수 있음.

```
$ jdb -attach localhost:8000
```

➔ 현재 running 중인 *system_server* 를 jdb에 attach 시켜줌(gdb와 유사)

Initializing jdb ...

```
> threads
```

← 선택한 process가 보유한 모든 thread 출력

...

...

```
> suspend
```

← 모든 thread를 일시 정지시킴

```
> where all
```

← 모든 thread의 stack 상태를 출력시킴.

...

(*) *ddms*가 좋기는 하나, 절차가 복잡하고, 느린 문제가 있다. 이럴 때는 *jdb*를 사용하는 편이^^...

(*) 시스템이 이상한 상태(?)에서 의심가는 process를 *suspend* 시킨 후, *stack*을 *dump*해서 보면 문제의 원인을 확인할 수 있는 좋은 출발점이 될 수 있겠다.^^

➔ C/C++ app의 경우는 *gdb*로 비슷한 처리 가능 !!!

(*) 위에서 제시한 옵션 외에도 매우 강력한 기능을 제공하고 있는데, 이를 확인하려면 *jdb* prompt 상태에서 *help*를 치면 된다.

3.2 jdb(2) – java application debugging

\$ adb forward tcp:8000 jdwp:171

→ pid 171 process 관련 debug 정보를 tcp port 8000으로 forwarding 해줌.

→ 8000 port는 다른 값을 사용할 수 있음.

\$ jdb -attach localhost:8000 -sourcepath /android/frameworks/base/services/java

→ source를 확인하고자 할 경우, source path 입력(주의: 실제 java source의 위치가 아니라, 해당 패키지 위치이어야 함)

Initializing jdb ...

> threads

← 선택한 process가 보유한 모든 thread 출력

...

> thread 0xc12aaca1d8

← 특정 thread 선택(threads 실행하여 나오는 결과 중, 첫 번째 열의 숫자 값 지정)

> main[1] suspend 0xc12aaca1d8

← 해당 thread를 일시 정지시킴

> main[1] where

← 해당 thread의 stack 상태를 출력시킴.

> main[1] list

← java code인 경우만 source code 출력

> main[1] up

← stack 이동, down 명령도 사용 가능

> main[1] locals

← stack 상의 local variable 정보 출력

> main[1] stop at <classid>:<line>

← break point 걸기

=> 이걸 사용법을 적은 것임. 실제 classid와 line number를 입력해야 함.

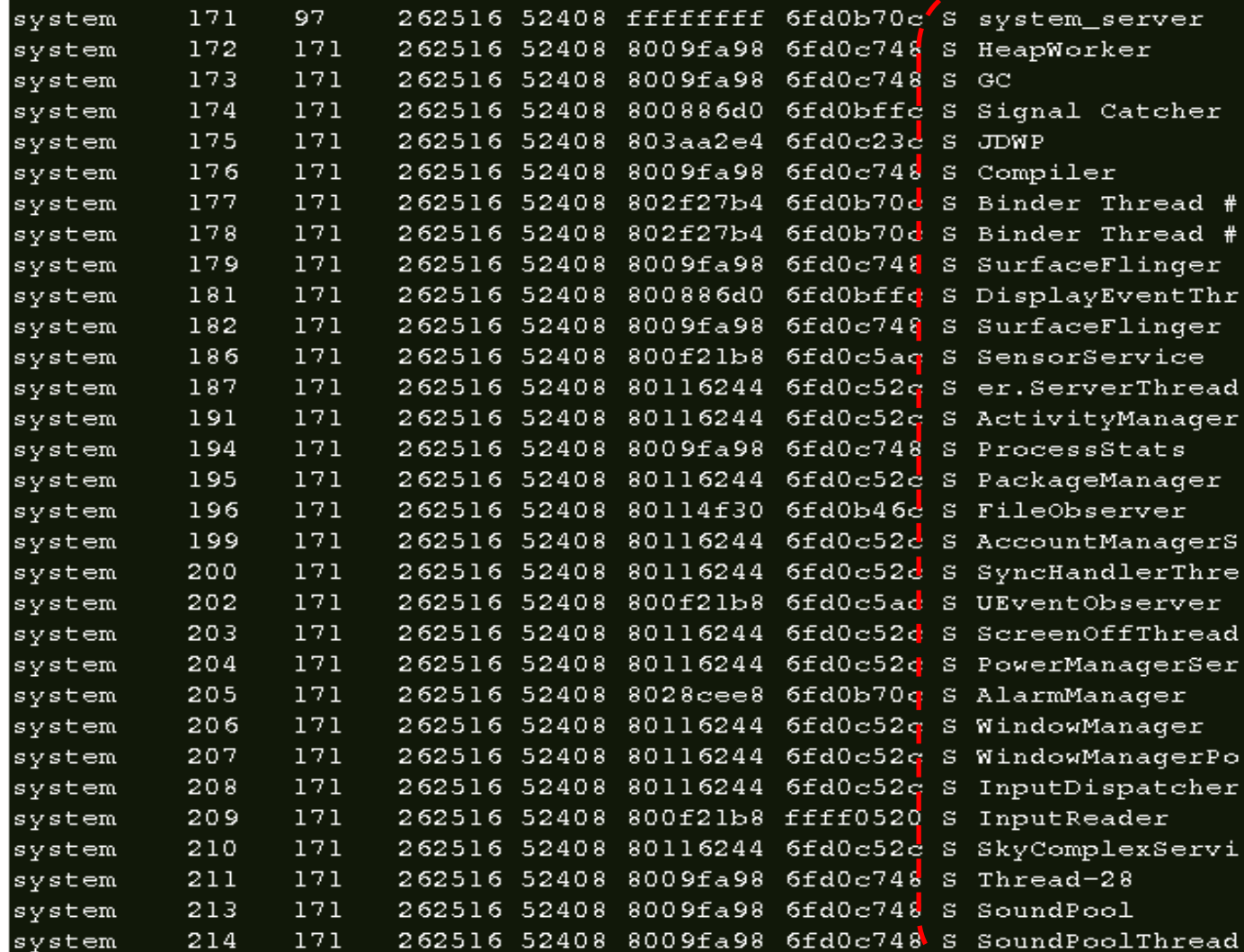
> main[1] resume

← resume하여 다시 실행하기

...

3.2 jdb(3) - adb shell ps -t 0/

(*) system_server 관련 threads



```
system 171 97 262516 52408 ffffffff 6fd0b70c S system_server
system 172 171 262516 52408 8009fa98 6fd0c748 S HeapWorker
system 173 171 262516 52408 8009fa98 6fd0c748 S GC
system 174 171 262516 52408 800886d0 6fd0bffc S Signal Catcher
system 175 171 262516 52408 803aa2e4 6fd0c23c S JDWP
system 176 171 262516 52408 8009fa98 6fd0c748 S Compiler
system 177 171 262516 52408 802f27b4 6fd0b70c S Binder Thread #
system 178 171 262516 52408 802f27b4 6fd0b70c S Binder Thread #
system 179 171 262516 52408 8009fa98 6fd0c748 S SurfaceFlinger
system 181 171 262516 52408 800886d0 6fd0bffc S DisplayEventThr
system 182 171 262516 52408 8009fa98 6fd0c748 S SurfaceFlinger
system 186 171 262516 52408 800f21b8 6fd0c5ad S SensorService
system 187 171 262516 52408 80116244 6fd0c52c S er.ServerThread
system 191 171 262516 52408 80116244 6fd0c52c S ActivityManager
system 194 171 262516 52408 8009fa98 6fd0c748 S ProcessStats
system 195 171 262516 52408 80116244 6fd0c52c S PackageManager
system 196 171 262516 52408 80114f30 6fd0b46c S FileObserver
system 199 171 262516 52408 80116244 6fd0c52c S AccountManagers
system 200 171 262516 52408 80116244 6fd0c52c S SyncHandlerThre
system 202 171 262516 52408 800f21b8 6fd0c5ad S UEventObserver
system 203 171 262516 52408 80116244 6fd0c52c S ScreenOffThread
system 204 171 262516 52408 80116244 6fd0c52c S PowerManagerSer
system 205 171 262516 52408 8028cee8 6fd0b70c S AlarmManager
system 206 171 262516 52408 80116244 6fd0c52c S WindowManager
system 207 171 262516 52408 80116244 6fd0c52c S WindowManagerPo
system 208 171 262516 52408 80116244 6fd0c52c S InputDispatcher
system 209 171 262516 52408 800f21b8 ffff0520 S InputReader
system 210 171 262516 52408 80116244 6fd0c52c S SkyComplexServi
system 211 171 262516 52408 8009fa98 6fd0c748 S Thread-28
system 213 171 262516 52408 8009fa98 6fd0c748 S SoundPool
system 214 171 262516 52408 8009fa98 6fd0c748 S SoundPoolThread
```

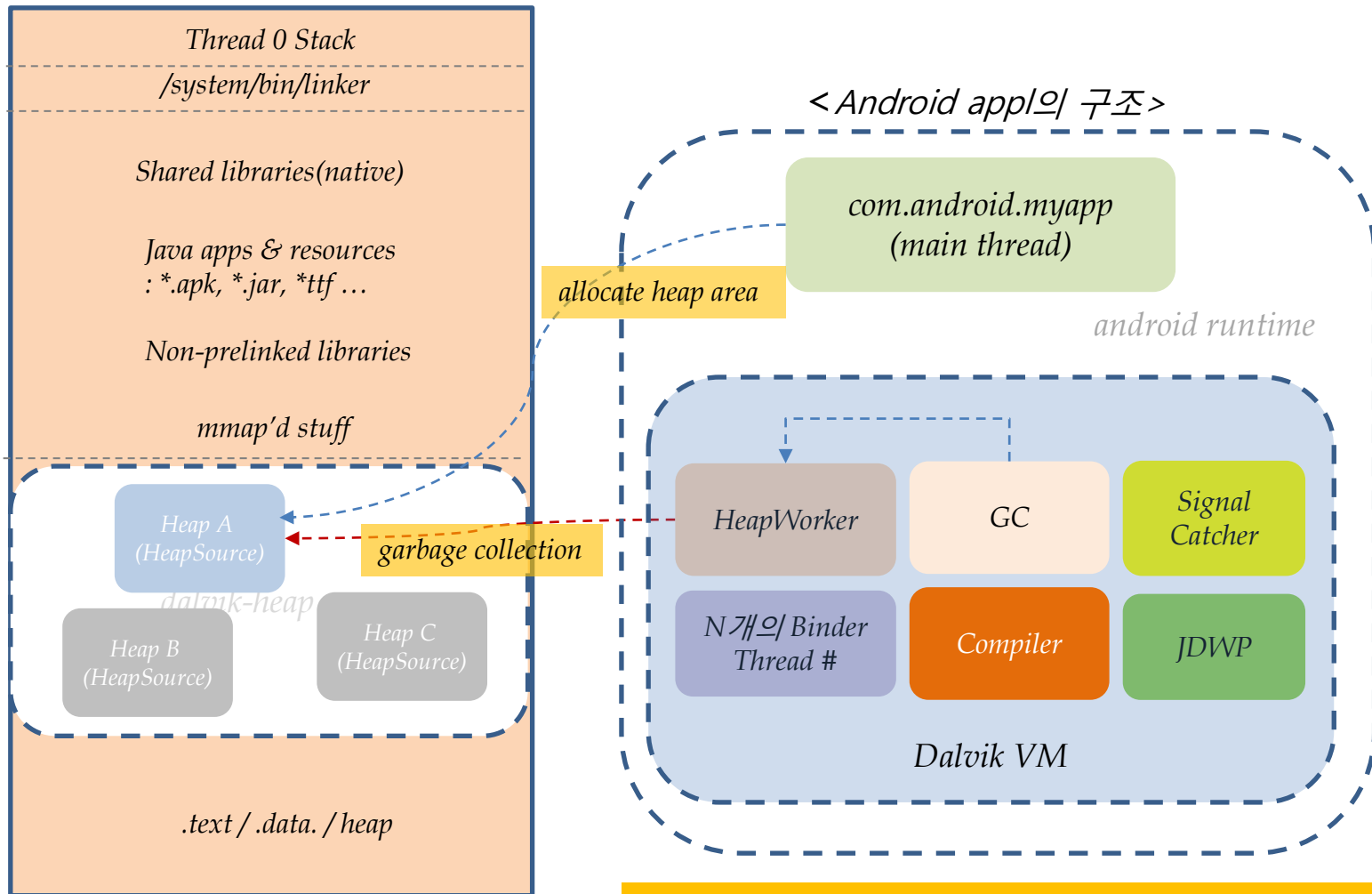
3.2 jdb(4) – stack dump

(*) *system_server*가 보유하는 모든 *thread*의 현재 *stack* 상태를 출력 !

```
> suspend
All threads suspended.
> where all
<1> main:
  [1] com.android.server.SystemServer.init1 (native method)
  [2] com.android.server.SystemServer.main (SystemServer.java:681)
  [3] java.lang.reflect.Method.invokeNative (native method)
  [4] java.lang.reflect.Method.invoke (Method.java:507)
  [5] com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run (ZygoteInit.java:864)
  [6] com.android.internal.os.ZygoteInit.main (ZygoteInit.java:622)
  [7] dalvik.system.NativeStart.main (native method)
<64> Binder Thread #9:
  [1] dalvik.system.NativeStart.run (native method)
<56> android.hardware.SensorManager$SensorThread:
  [1] android.hardware.SensorManager.sensors_data_poll (native method)
  [2] android.hardware.SensorManager$SensorThread$SensorThreadRunnable.run (SensorManager.java:446)
  [3] java.lang.Thread.run (Thread.java:1,019)
<63> 00:0F:E4:C2:52:21:
  [1] android.os.MessageQueue.nativePollOnce (native method)
  [2] android.os.MessageQueue.next (MessageQueue.java:119)
  [3] android.os.Looper.loop (Looper.java:117)
  [4] android.os.HandlerThread.run (HandlerThread.java:60)
<61> Thread-94:
  [1] android.os.MessageQueue.nativePollOnce (native method)
  [2] android.os.MessageQueue.next (MessageQueue.java:119)
  [3] android.os.Looper.loop (Looper.java:117)
  [4] com.google.android.gsf.Gservices$1.run (Gservices.java:78)
<62> DHCP Handler Thread:
  [1] android.os.MessageQueue.nativePollOnce (native method)
```

3.3 VM Heap(1)

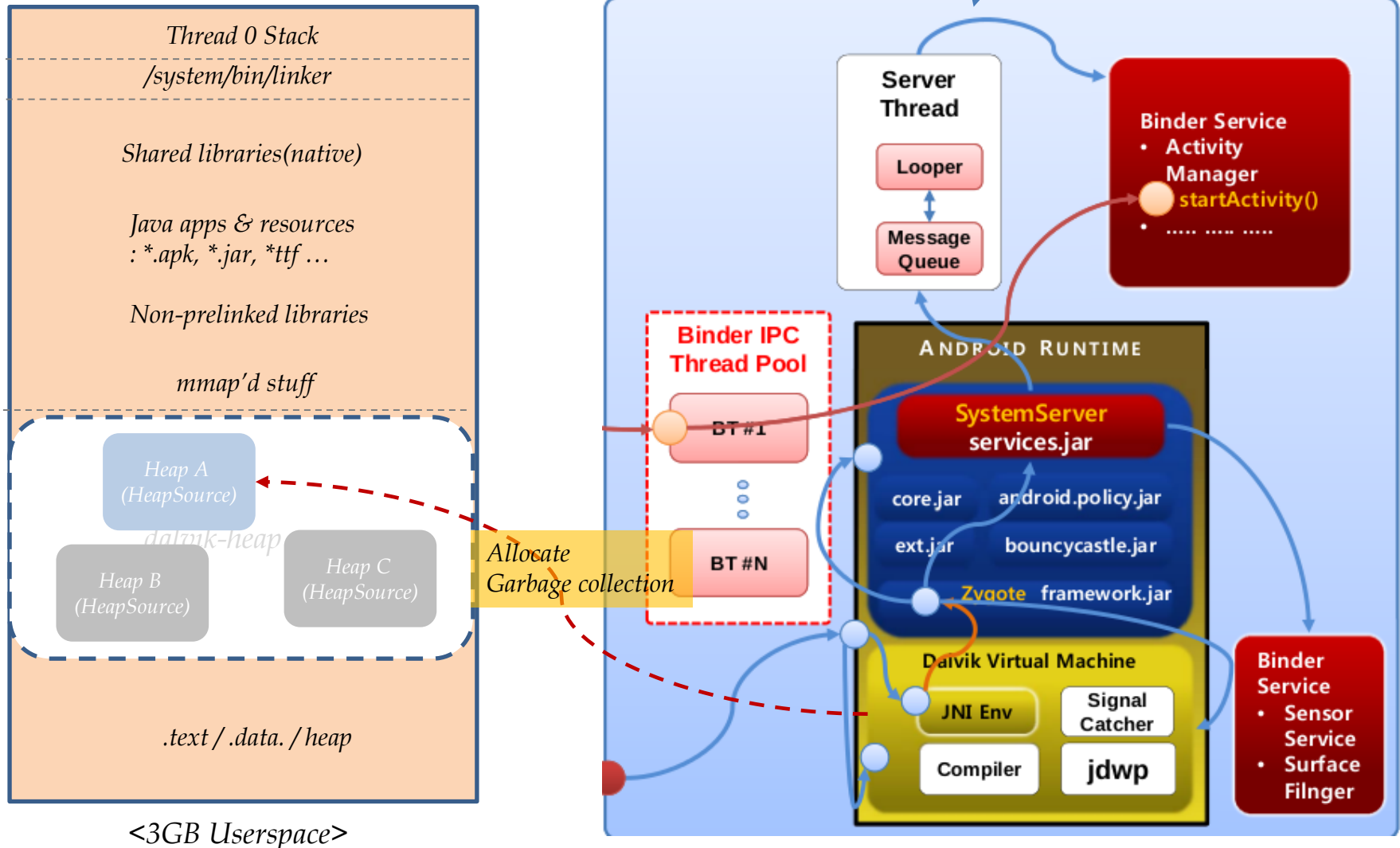
- (*) Davik heap 메모리는 `dldmalloc(open source)`을 사용하여 구현되어 있음.
- (*) Davik VM은 process마다 독자적인 heap을 관리하고 있음.



- (*) **HeapWorker**: garbage collection 작업 수행. GC가 깨워줌
- (*) **Signal Catcher**: signal 처리를 위한 thread
- (*) **JDWP**: VM과 Debugger(DDMS)간의 연결 통로 (Java Wired Debug Protocol)

3.3 VM Heap(2) – system_server

(*) 아래 그림은 들풀 양정수님의 문서에서 퍼 온 것임^^ (최고의 그림)
→ 그림 사용해도 되죠^^



3.3 VM Heap(3)

- # 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...

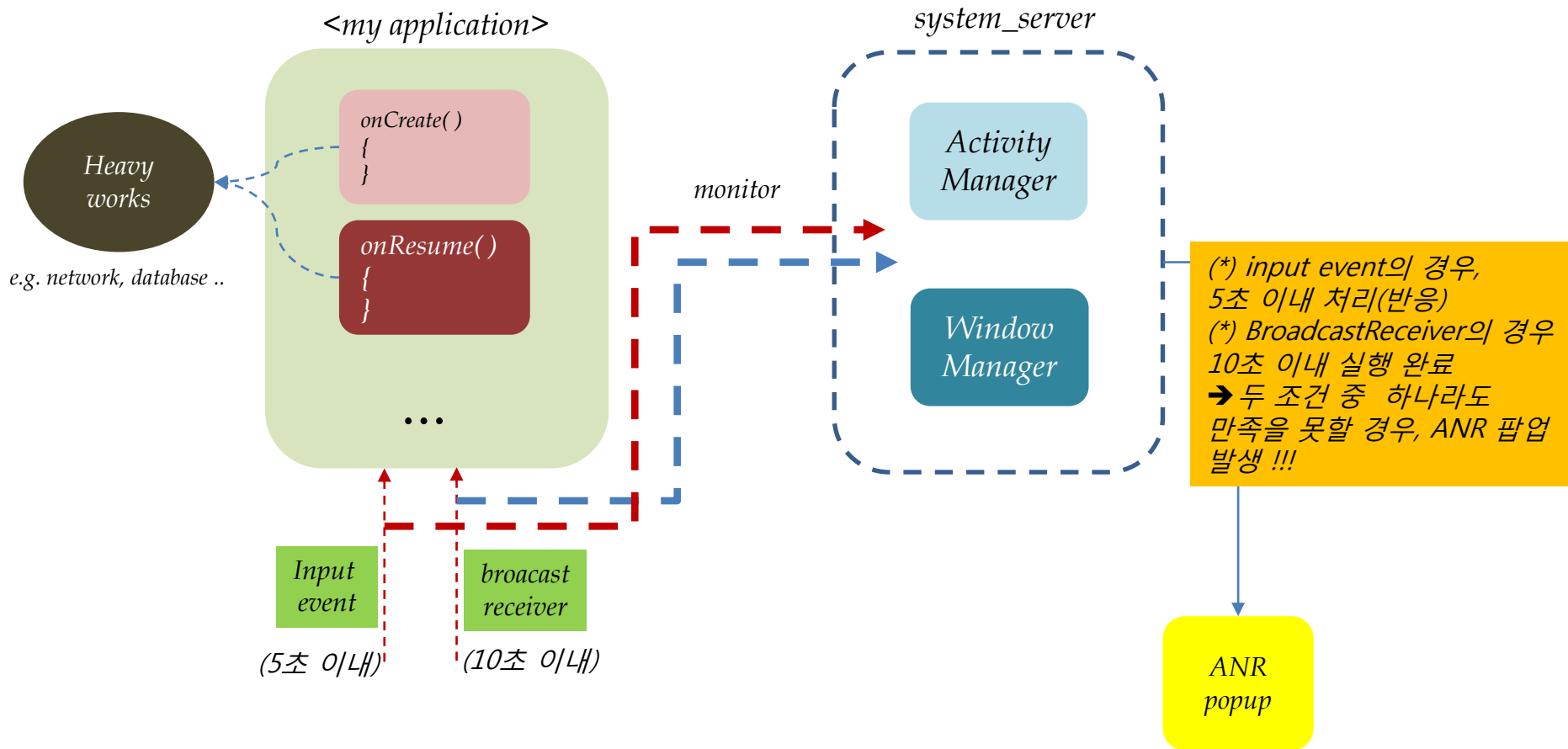
3.4 ANR(Applicatoin Not Responding)에 관하여(1)

- What Triggers ANR ?
 - No reponse to an input event(e.g. key press, screen touch) within 5 seconds.
(5초 이내에 input event에 반응하지 않을 때)
 - A BroadcastReceiver hasn't finished executing within 10 seconds.
(10초 이내에 broadcast receiver의 수행을 끝마치지 못할 때)
- ➔ 위의 두 조건 중 하나에 해당할 때, ANR 팝업이 뜨면서 사용자의 입력(해당 process 강제 종료 혹은 다시 대기)을 기다리게 됨.

(*) For more information, you can see the following site.

<http://developer.android.com/guide/practices/design/responsiveness.html>

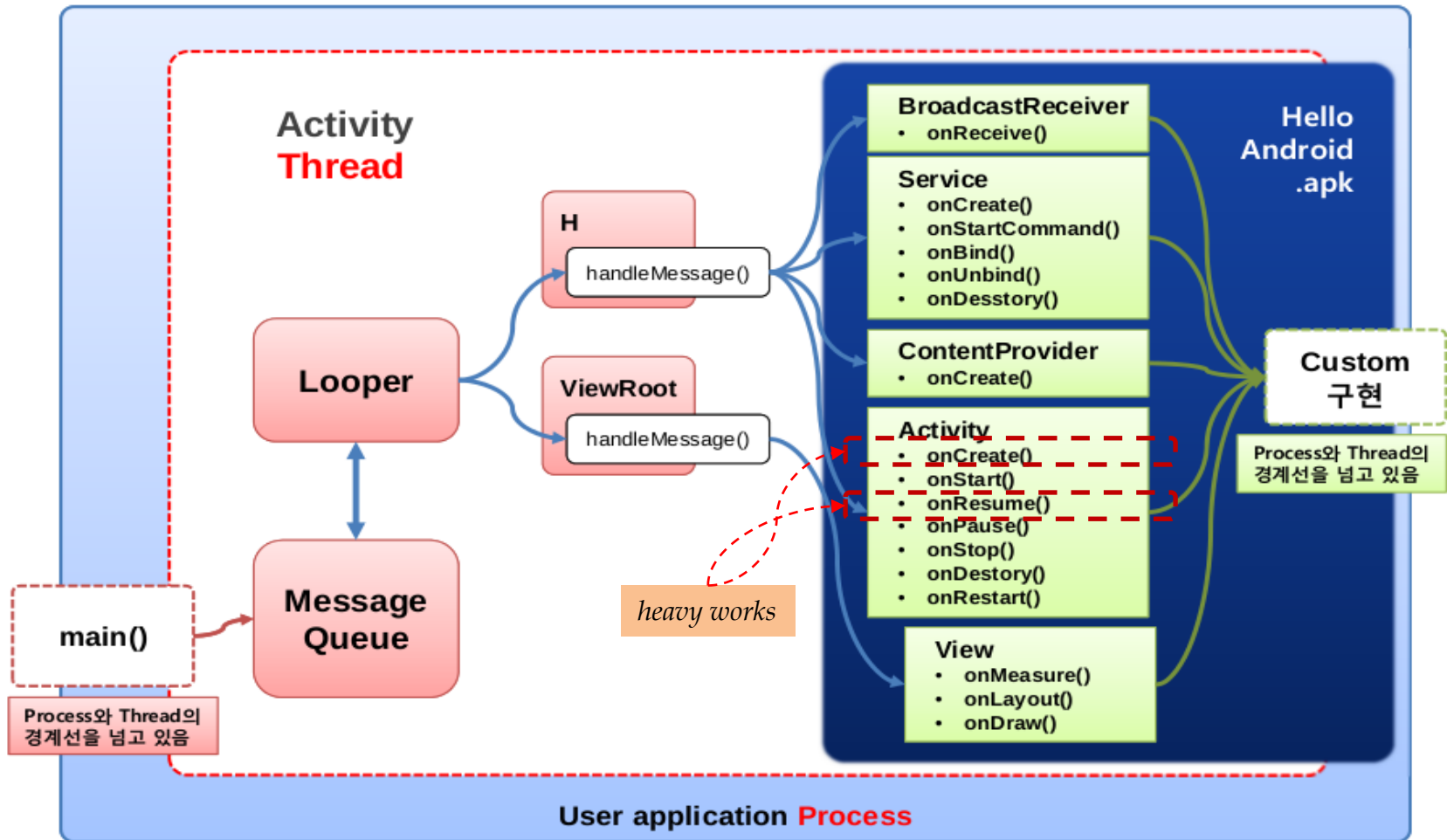
3.4 ANR(Application Not Responding)에 관하여(2)



(*) ANR를 피하는 방법으로는

- 1) onCreate(), onResume() 내부에서 heavy한 작업을 할 경우, child thread로 처리
- 2) Heavy한 작업을 할 경우는 progress bar를 보여 주면 효과적
- 3) StrictMode를 사용하면, 잠재적인 문제를 발견하는데 도움이 됨(개발시)

3.4 ANR(Applicatoin Not Responding)에 관하여(3) - Appl. Message 처리



(*) android appl의 경우, 표면적으로 드러나지는 않지만, 내부적으로 위와 같은 message 처리 scheme을 보유하고 있다. Input Event나 Broadcast intent도 위의 구조를 통해 전달되고 있음^^.

(*) 따라서, ANR 문제는 자신이 처리할 작업이 heavy하여, 위의 방법으로 주변 process(혹은 thread)로부터 message를 받아 처리하는 작업이 늦어질 때 발생하는 것으로 이해하면 될 듯^^

(*) 위의 그림도 들쭉 양정수님의 글에서 퍼온 것임^^.

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)

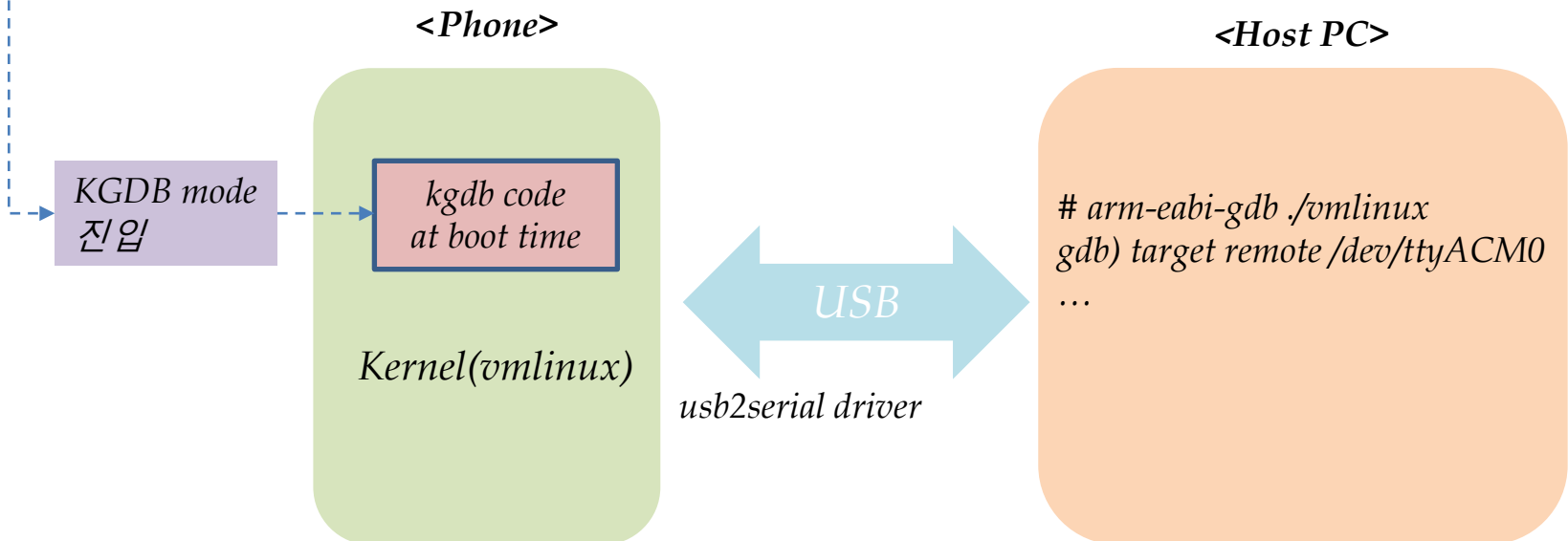
- # arm-eabi-gdb ./vmlinux
- # (gdb) info line 0xc50659ec
  - ➔ 문제가 되는 부분의 함수를 출력할 것임.

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

## 4.4 kgdb(1) - 개요

### <KGDB 모드 진입 가능한 조건>

- 1) **브레이크 포인트를 만날 때**(A break point has been hit)
- 2) **sysrq-q 트리거가 발생할 때**(Sysrq-g has been triggered)  
→ `$ adb shell sh -c "echo -n g>/proc/sysrq-trigger"`
- 3) **system exception(예외 상황)이 발생할 때**(A system exception is caught)
- 4) **부팅 중, kgdbwait 옵션이 kernel 부팅을 중지시킬 때**(The option "kgdbwait" halts the kernel during boot-up)(아직 지원 안함)



(\*) kgdb는 가장 강력한 kernel debugging 기법인 바, (아직 android 단말에서 테스트는 못해 보았으나) 이 장에서 소개하고자 한다. → <http://bootloader.wikidot.com/android:kgdb> site 참조함 !!!

(\*) kgdb를 제대로 사용하기 위해서는 다음 페이지 내용을 참조하여 kernel 작업(드라이버 작업)을 해주어야 한다. 시간 날 때 해 보아야겠다^^.

## 4.4 kgdb(2) – kernel work(드라이버 작업)

### 1) kernel build하기(다음의 configuration을 컴)

CONFIG\_KGDB (for KGDB)  
CONFIG\_HAVE\_ARCH\_KGDB (for KGDB)  
CONFIG\_CONSOLE\_POLL (for Android USB support)  
CONFIG\_MAGIC\_SYSRQ (use sysrq to invoke KGDB)

### 2) CONFIG\_USB\_ANDROID\_ACM kernel configuration 추가 및 관련 작업 수행

acm usb function을 추가하기 위해, arch/arm/mach-msm/board-mahimahi.c 등의 파일 수정 필요함. 아래 내용 추가

```
#ifdef CONFIG_USB_ANDROID_ACM
static char *usb_functions_adb_acm[] = {
 "adb",
 "acm",
};
#endif

static struct android_usb_platform_data android_usb_pdata = {
 .vendor_id = 0x18d1,
 .product_id = 0x4e11,
 .version = 0x0100,
 .product_name = "Nexus One",
 .manufacturer_name = "Google, Inc.",
 .num_products = ARRAY_SIZE(usb_products),
 .products = usb_products,
 .num_functions = ARRAY_SIZE(usb_functions_adb_acm), /* adb + acm */
 .functions = usb_functions_adb_acm,
};
```

(\*) 자세한 사항은 아래 site의 코드를 참조

→ <http://github.com/dankex/kgdb-android>

### 3) kernel command line 수정하기

kgdboc=ttyGS0 kgdbretry=4

← 자신의 단말에 맞게 적절히 수정해야 함.

## 4.4 kgdb(3) - gdb와 kgdb 연결

```
$ arm-eabi-gdb ./vmlinux
```

```
GNU gdb 6.6
```

```
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
...
```

```
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-elf-linux"...
```

```
(gdb) target remote /dev/ttyACM0
```

```
Remote debugging using /dev/ttyACM0
```

```
warning: shared library handler failed to enable breakpoint
```

```
0x800a1380 in kgdb_breakpoint () at
```

```
/.../kernel/arch/arm/include/asm/atomic.h:37
```

```
37 __asm__ __volatile__("@ atomic_set\n"
```

```
(gdb)
```

## 4.4 kgdb(4) - 예

<phone에서 Sysrq 트리거 발생시켜 KGDB 모드로 진입>

```
$ adb shell
cd /data
mkdir kgdb-test
cd kgdb-test
ls
echo -n g>/proc/sysrq-trigger
```

<PC에서 gdb로 kgdb에 접속 후, debugging 시작>

```
$ arm-eabi-gdb ./vmlinux
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it, and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-elf-linux"...
(gdb) target remote /dev/ttyACM0
Remote debugging using /dev/ttyACM0
warning: shared library handler failed to enable breakpoint
0x800a1380 in kgdb_breakpoint () at ../../kernel/arch/arm/include/asm/atomic.h:37
37 __asm__ __volatile__("@ atomic_set\n"
(gdb) l
32 */
33 static inline void atomic_set(atomic_t *v, int i)
34 {
35 unsigned long tmp;
36
37 __asm__ __volatile__("@ atomic_set\n"
38 "1: ldrex %0, [%1]\n"
39 " strex %0, %2, [%1]\n"
40 " teq %0, #0\n"
41 " bne 1b"
(gdb) info br
No breakpoints or watchpoints.
(gdb) br sys_mkdir
Breakpoint 1 at 0x800e033c: file ../../kernel/fs/namei.c, line 2085.
(gdb) c
Continuing.
[New Thread 1092]
[Switching to Thread 1092]
```

## 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 있겠으나, 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.

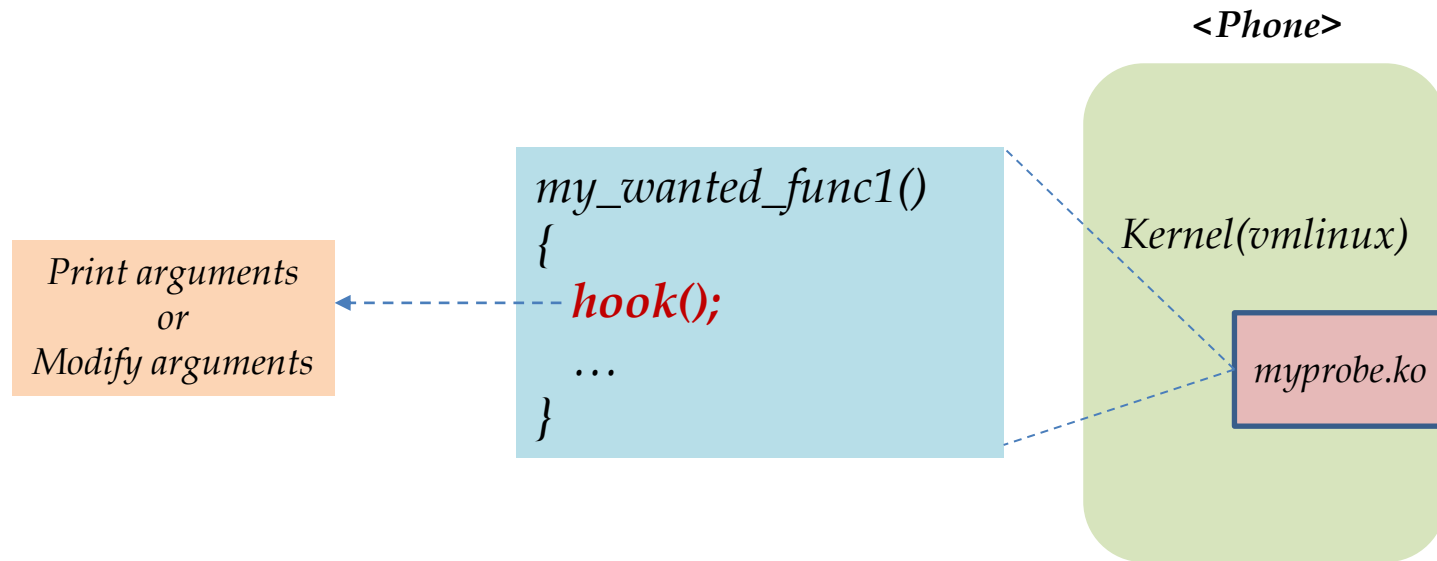
(\*) kprobe/jprobe 관련 자세한 사항은 doc/Documentation/kprobes.txt 파일 참조

(\*) 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)



## 4.5 kprobe or jprobe를 사용한 실시간 debugging(3)

```
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(4) - *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(5) - *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;
}

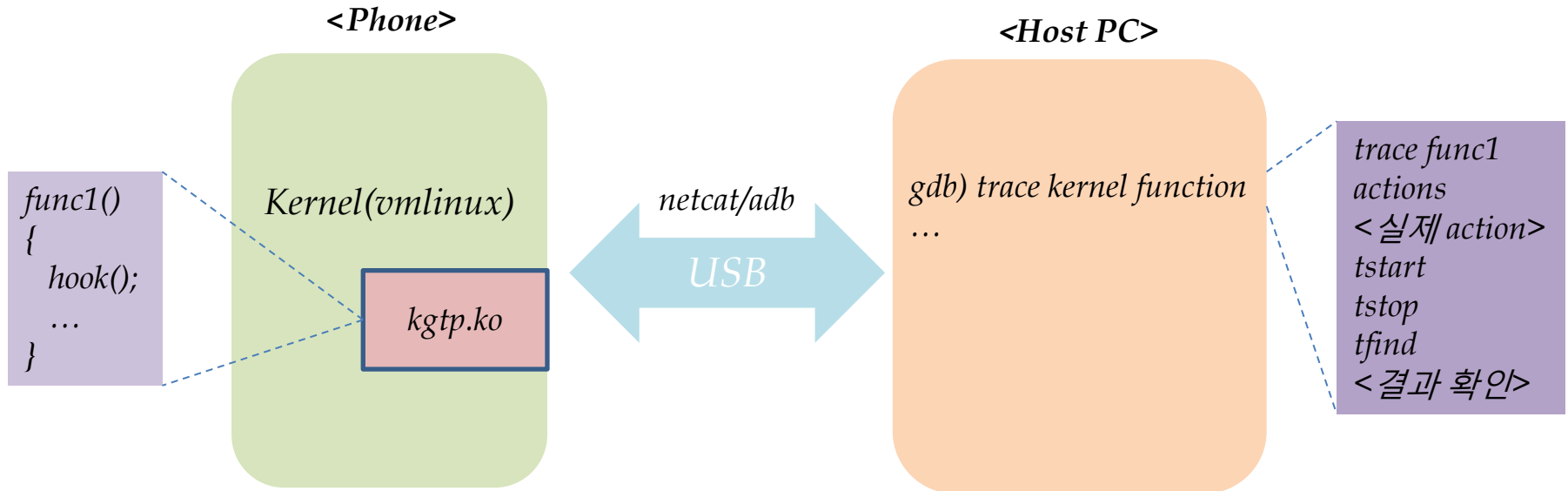
static void __exit my_exit(void)
{
 unregister_jprobe(&jp);
 printk(KERN_INFO "jprobe unregistered\n");
 printk(KERN_INFO "FINAL:mod_timer_count=%ld\n", mod_timer_count);
}

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.

## 4.6 kgtp : kernel GDB tracepoint module(1)



(\*) kgtp(trace point module)는 kprobe에 기반을 둔 kernel debugging 기법으로 gdb(gdb-release)를 활용하여, 현재 동작중인 kernel의 특정 함수를 trace할 수 있는 매우 효과적인 방법임.

(\*) kgtp 관련 자세한 사항은 <http://code.google.com/p/kgtp> 참조 !!!

## 4.6 kgtp : kernel GDB tracepoint module(2) – kernel만 build 하기

- **<PC>**  
# cd android  
# source build/envsetup.sh  
# choosecombo  
[Device] 1  
[release] 1  
[target] 1  
[eng] 3  
  
# cd kernel  
# make ARCH=arm CROSS\_COMPILE=arm-eabi- \_defconfig  
  
# make ARCH=arm CROSS\_COMPILE=arm-eabi- menuconfig  
  
# make -j3 ARCH=arm CROSS\_COMPILE=arm-eabi- zImage  
  
# make -j3 ARCH=arm CROSS\_COMPILE=arm-eabi- modules

General setup →

[\*] Prompt for development and/or incomplete code/drivers

[\*] Kprobes

Kernel hacking →

[\*] Compile the kernel with debug info

[\*] Compile the kernel with frame pointers

(\*) kgtp module을 build할 때, include/linux/config/autoconf.h 파일이 필요하므로, 위와 같이 kernel을 수동으로 build한다.

## 4.6 kgtp : kernel GDB tracepoint module(3) – kgtp.ko build & install

- 1) Makefile 수정(아래 내용 수정)

<PC>

*KERNELDIR := ~/android/kernel*

*CROSS\_COMPILE := ~/CodeSourcery/Sourcery\_G++\_Lite/bin/arm-none-linux-gnueabi-*

→ arm-eabi- toolchain으로 build할 경우, error가 발생하니, CodeSourcery toolchain을 사용하여 build해야 함.

*ARCH := arm*

### 2) kgtp.ko build하기

<PC>

*# make AUTO=0*

→ gtp.ko가 정상적으로 생성됨.

→ AUTO=0을 안줄 경우, insmod시 에러 발생함(Exec format error).

### 3) insmod & netcat으로 port 열어 두기

<PC>

*# sudo adb push ./gtp.ko /data/test*

← /data/test 디렉토리에 gtp.ko 파일 복사

<phone>

*# mount -t debugfs nodev /sys/kernel/debug* ← kgtp를 사용하려면, debugfs가 mount되어 있어야 함.

*# insmod ./gtp.ko*

<phone>

*# nc -l -p 1234 < /sys/kernel/debug/gtp > /sys/kernel/debug/gtp*

→ 종료하지 않고, 대기 상태로 있게 됨.

(\*) kgtp kernel module code를 download([code.google.com/p/kgtp](http://code.google.com/p/kgtp))한 후, 이를 build한다.

## 4.6 kgtp : kernel GDB tracepoint module(4) – *gdb-release* 구동

- **1) gdb-release download 받기**

(\*) kgtp를 사용하기 위해서는 gdb-release version을 사용하여야 함.

For the Ubuntu 10.04 or later, running the following line at a terminal:

**<PC>**

*# sudo add-apt-repository ppa:teawater/gdb-\$(lsb\_release -rs)*

*# sudo apt-get update*

*# sudo apt-get install gdb-release*

- **2) gdb-release 실행**

**<PC>**

*# adb forward tcp:1234 tcp:1234*

→ netcat이 열어둔 1234번 port의 내용을 localhost 1234번으로 forwarding

*# cd android/out/target/product/[redacted]obj/KERNEL\_OBJ*

→ kernel/vmlinux 파일을 이용해도 됨.

*# gdb-release -ex "set gnutarget elf32-littlearm" -ex "file ./vmlinux"*

*(gdb) target remote 127.0.0.1:1234*

Remote debugging using 127.0.0.1:1234

warning: (Internal error: pc 0x0 in read in psymtab, but not in symtab.)

warning: (Internal error: pc 0x0 in read in psymtab, but not in symtab.)

→ 이 에러는 일단 무시.

(\*) kgtp를 사용하기 위해서는 gdb-release 버전을 새로 download 받아야 한다.



## 4.6 kgtp : kernel GDB tracepoint module(5) – *gdb/kgtp ≡ debugging*

- **<PC>**

- **(gdb) trace vfs\_readdir**

Tracepoint 1 at 0xc02289f0: file /build/buildd/linux-2.6.35/fs/readdir.c, line 23.

- **(gdb) actions**

Enter actions for tracepoint 1, one per line.

End with a line saying just "end".

>collect \$reg

>end

- **(gdb) tstart**

(gdb) tstop

(gdb) tfind

Found trace frame 0, tracepoint 1

#0 vfs\_readdir (file=0x0, filler=0x163d8ae3, buf=0x18c0) at /build/buildd/linux-2.6.35/fs/readdir.c:23

23 {

- **(gdb) bt**

#0 vfs\_readdir (file=0x0, filler=0x800f11d4 <generic\_block\_fiemap+20>, buf=0x0)

at [redacted]

[redacted]/android/kernel/fs/readdir.c:23

#1 0x800f1554 in fillonedir (\_\_buf=0x9faa435c, name=<optimized out>, namlen=-2146496044, offset=<optimized out>, ino=2961216046050051823, d\_type=0)

at [redacted]

[redacted]/android/kernel/fs/readdir.c:93

#2 0x000006ee in slip\_exit ()

at [redacted]

[redacted]/android/kernel/drivers/net/slip.c:1365

#3 0x000006f4 in slip\_exit ()

at [redacted]

[redacted]/android/kernel/drivers/net/slip.c:1369

#4 0x000006f4 in slip\_exit ()

at [redacted]

[redacted]/android/kernel/drivers/net/slip.c:1369

Backtrace stopped: previous frame identical to this frame (corrupt stack?)

## 4.6 kgtp : kernel GDB tracepoint module(6)

- TODO

- ➔ Kernel 함수를 debugging 하기는 좋은데, trace가 안되는 함수도 많이 있음^^
- ➔ gdb) source ~/kgtp/getmod.py 를 돌리는데 에러가 발생하여, module의 symbol을 trace하는데 문제 있음.
- ➔ Android 용은 아직까지 문제가 있는 듯 ...
- ➔ Systemtap 에 관해 좀 더 연구^^

- KGTP 사용법

See <http://code.google.com/p/kgtp> for more information.

*Thanks a lot !*



*SlowBoot*