

Android Kernel Hacks

안드로이드를 위한 리눅스 커널 해스



2013.7.16 ~ 2013.9.27

이 충 한(chunghan.yi@gmail.com, slowboot)

머리말

1장. 안드로이드 소개

2장. 주요 커널 프로그래밍 기법 1

3장. 주요 커널 프로그래밍 기법 2

4장. ARM 보드 초기화 과정 분석

5장. 파워 관리(Power Management) 기법

6장. 주요 스마트폰 디바이스 드라이버 분석

7장. **커널 디버깅 기법 소개**

인덱스

커널 디버깅 기법 소개



이번 장에서는 주요 리눅스 커널 디버깅 기법 중 Android 단말에서 실제 적용이 가능한 기법을 위주로 소개해 보고자 한다.

- dmesg
 - Kernel Panic/Oops
 - addr2line & objdump
- Panic 시, dmesg 추출 방법
 - Kexec/Kdump 기법[참고용]
 - [Android Ram Console](#)[Google에서 제안한 방법]
 - [Ram Dump](#)[참고용]
- 대화식(Interactive) 디버깅 기법
 - [kgdb & USB 기반 디버깅\(S/W 방식\)](#)
 - [OpenOCD & JTAG 기반 디버깅\(H/W 방식\)](#)
 - [TRACE32 & JTAG 기반 디버깅\(H/W 방식\)](#)
- Kernel Trace 기법
 - kprobe/jprobe/kretprobe
 - kgtp
- 사용자 영역 디버깅 기법

효과적인 커널 디버깅 방법은 과연 무엇일까? 아마 대부분의 커널 개발자가 고민하는 문제가 아닐까 싶다. 필자의 경우도 이 문제에 관하여 그 동안 많은 고민을 해 왔다고 생각하는데, 본 장에서는 필자가 생각하는 효과적인 커널 디버깅 기법에 관하여 소개해 보고자 한다.

<효과적인 커널 디버깅 가이드>

1) 커널 코드(device driver 포함)를 제대로 이해하고 있어야 한다.

- 본 서의 2, 3장 내용 활용
- 정답은 커널 코드 안에 있다. 하지만, 내가 작성한 코드가 아닌 경우가 더 많으며, 복잡한 상황에서 문제가 발생하는 경우가 대부분이다. 그래도 커널 코드를 훑히 들여다 볼 수 있는 상태를 항상 만들도록 노력해야 한다.

2) dmesg(kernel debug message)를 제대로 분석할 수 있어야 한다.

- Kernel panic & oops를 제대로 분석할 수 있어야 함.
- Stack trace – 대개의 경우 stack trace 정보가 문제의 원인을 찾는 데 도움이 되지만, 항상 해답을 주는 것은 아님에 주의.

3) 항상 dmesg를 확인할 수 있는 것은 아니므로, 문제가 발생할 당시의 dmesg 정보를 추출해 낼 수 있어야 한다.

- Ram Console & Ram Dump
- 문제 발생 당시의 dmesg 정보를 추출할 수 있다면, 문제의 절반을 해결한 셈.

4) 문제가 재현이 가능할 경우라면, 대화식 디버깅 기법인 gdb/kgdb(USB 기반의 S/W 방식, JTAG 기반의 H/W 방식)를 활용할 수 있어야 한다.

- USB 기반의 KGDB – S/W 방식이므로, 약간의 노력(porting)만으로 활용 가능. 단, 부팅 초기에 문제가 발생하는 경우(bootloader 디버깅 포함)는 디버깅할 방법이 없음.
- JTAG/OpenOCD 기반의 GDB – H/W 방식이나, 저가이므로 활용도 높을 수 있음. JTAG H/W 장치가 구비되어 있어야 함.
- Trace32 활용 – JTAG 기반의 H/W 방식임. 고가이므로 회사에 TRACE32가 구비되어 있을 경우만 사용 가능함.

5) 기타 다양한 kernel trace 기법을 활용한다.

- Kprobe/Jprobe, Tracepoint, KGTP, LTTng ...
- 시스템 성능이 저하되므로, 양산 버전에서 테스트 하는 것에는 무리가 있다.

6) 커널에서 제공하는 debugging option을 활용해 본다.

- 시스템 성능이 저하되므로, 양산 버전에서 테스트 하는 것에는 무리가 있다.

7) 문제의 근원지가 커널이 아니라, 사용자 영역(userspace)일 수도 있음.

- 리눅스 커널의 전반적인 특성(/proc, busybox 활용 등)을 알고 있어야 한다.
- 더불어 사용자 영역의 debugging 기법도 이해하고 있어야 함.

아래 표 7-1은 앞으로, 본 장에서 언급할 주요 디버깅 기법의 개발 보드 및 양산 보드 별 사용 가능 여부를 비교 정리한 것이다.

표 7-1 디버깅 기법 비교

디버깅 기법	사용 가능 여부 (개발 보드)	사용 가능 여부 (양산 보드)	비고
Serial Console – dmesg	OK	NG	시리얼 포트 있어야 함. 실시간 커널 로그 확인 가능
adb - dmesg	OK	OK	adb 연결 안되면 효과 없음
Ram Console	OK	OK	Power off하여 재 부팅하면 안 됨(정보가 사라짐).
Ram Dump	OK	OK	구현해야 할 사항이 좀 있음. (ram dump 기능 구현해야 함)
KGDB & GDB	OK	OK	약간의 포팅 작업 필요함. 커널 부팅 초반 부 debugging 불가(bootloader 포함)
JTAG & GDB	OK	NG (JTAG 포트 없을 경우)	지원하는 JTAG H/W가 있어야 하며, 커널 부팅 초반 부 디버 깅 가능. 가격 저렴.
Trace32	OK	OK (simulation)	고가의 T32 장비가 있어야 함. 커널 부팅 이전 상황까지도 debugging이 가능함(매우 강 력함) – 최고의 디버깅 Tool

1. Kernel 로그 메시지 분석 방법

본 절에서는 Kernel panic 혹은 Oops 발생 시, kernel debug message(/proc/kmsg)를 활용하여 문제의 원인을 분석하는 방법을 먼저 소개하고자 한다.

1.1 Serial Console 연결

타겟 장치에 시리얼 포트가 있다면, 이를 시리얼 콘솔 에뮬레이터로 연결함으로써 커널 로그 메시지를 실시간으로 확인이 가능하다. 그림 7-1은 리눅스용 시리얼 콘솔 에뮬레이터인 미니콤(minicom)을 사용하여 타겟 장치와 연결하는 예를 보여준다(이 밖에도 윈도우로 다양한 에뮬레이터가 있음).

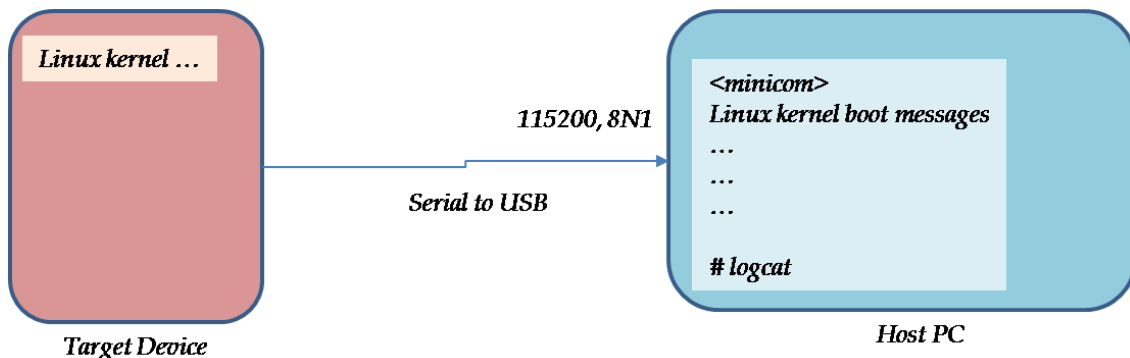


그림 7-1 시리얼 콘솔 연결

1.2 adb로 kernel log 확인하기

만일 타겟 장치에 시리얼 포트가 나와 있지 않다면(대부분의 경우에 해당함), adb(android debug bridge)를 사용하여 커널 로그를 확인할 수 있다. 단, 이 방법의 문제는 adb(adbd)가 살아 있을 경우에만 커널 로그 확인이 가능하므로, 부팅 초반부의 커널 로그 확인이 불가능하다는 점이다.

```
$ adb shell
```

```
$ dmesg
```

- 커널 로그 한번 출력

```
$ cat < /proc/kmsg
```

- 실시간으로 커널 로그 출력, serial cable이 없을 경우에 유용한 방법

or

```
C:\W> adb shell "cat < /proc/kmsg"
```

- 타겟 장치에 로그인하지 않은 상태에서, 실시간으로 커널 로그 출력

1.3 커널 패닉(Panic)과 오프스(Oops) 보는 법

커널이 비정상적으로 동작할 경우, 패닉(panic) 혹은 오프스(Oops) 메시지를 뿌리며 죽는 경우가 있다. 커널 패닉은 문제 발생 후 커널 실행을 멈춘다는 측면에서 커널 오프스와는 다르다. 패닉 상태에서는 커널이 더 이상 동작하지 못하지만 오프스 상태에서는 커널이 계속 수행될 수 있다. 하지만 커널 오프스가 시스템의 핵심 자료 구조체를 망가뜨려서 커널 수행이 불가능할 경우 커널 패닉이 뒤따라 나올 수 있다.

1) 커널 오프스(Oops)

유효하지 않은 메모리 위치에 값을 쓰게 되면, 오프스 상황이 발생하고, 콘솔에 로그가 남게 된다. 오프스 메시지 안에는 레지스터 덤프, 스택 덤프 및 함수 호출 트레이스 정보 등이 출력되며, "Unable to handle kernel NULL pointer dereference"로 시작되는 로그가 출력되게 된다.

<Kernel Oops Log>

```
<1>[ 188.636222] Unable to handle kernel NULL pointer dereference at virtual address 00000000
<1>[ 188.646391] pgd = e7434000
<1>[ 188.646442] [00000000] *pgd=676ea031, *pte=00000000, *ppte=00000000 <0>[ 188.654291] Internal error: Oops:
817 [#1] PREEMPT SMP
<0>[ 188.659632] last sysfs file: /sys/devices/system/cpu/cpu0/rq-stats/def_timer_ms <4>[ 188.666839] Modules linked in:
char
<4>[ 188.670402] CPU: 0      Not tainted (2.6.35.7-perf+ #2)
<4>[ 188.675537] PC is at device_read+0x34/0x7c [char]
<4>[ 188.679965] LR is at release_console_sem+0x1c4/0x220
<4>[ 188.684989] pc : [<bf0000d4>]      lr : [<c00d3930>]      psr: 60000013 ....
```

웁스가 발생한 정확한 위치를 파악하기 위해서는 addr2line과 objdump 등의 명령어를 활용할 수 있는데, 먼저 objdump를 이용한 방법을 소개하면 다음과 같다.

<objdump로 문제 위치 알아내기>

```
# arm-eabi-objdump -d ./vmlinux > objdump.txt
# vi objdump.txt
```

위의 웁스 메시지 중 "PC is at WWWWW + 0xxxxx/0xyyyy" 부분을 주목하기 바란다. 위의 arm-eabi-objdump 결과로 얻은 파일(objdump.txt)에서 WWWWW 함수를 찾고, 거기에서부터 0xxxxx offset 위치의 코드가 문제의 코드임을 알 수 있다. 0xyyyy는 WWWWW 함수의 크기를 나타낸다. 참고로, 커널 디버깅을 위해서는 반드시 vmlinux가 필요하며, 이는 아래 위치에서 구할 수 있다.

```
$(YOUR_ANDROID)/out/target/product/$YOUR_TARGET_DEVICE/obj/KERNEL_OBJ/vmlinux
```

2) 커널 패닉(Panic)

패닉의 경우는 웁스 이외의 상황으로 볼 수 있으며, 아래와 같이 패닉 메시지 및 백 트레이스 정보를 출력하고 더 이상의 실행을 멈추게 된다. 만일 이 상태에서 커널에 watchdog이 동작하도록 설정되어 있다면, 일정 시간 경과 후, 시스템이 자동으로 리셋되므로, 주의하기 바란다.

<Kernel Panic 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)
```

이미 커널 패닉이 되면서 백 트레이스 정보를 출력하였으므로, 아래 명령(addr2line)을 사용하는 것이 별 의미는 없으나, PC(program counter) 값 만을 알고 있는 경우, 커널 패닉이 발생한 위치를 찾는 방법을 소개하는 차원에서 내용을 정리해 보았으니, 참고하기 바란다.

<addr2line로 문제 위치 알아내기>

```
$ 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 죽었음을 의미한다.

1.4 기타 참고 사항

개발을 하다 보면, ioctl(), proce filesystem, sysfs filesystem 등을 활용하여 자신만의 디버그 메시지를 추가하기가 쉬운데, 이 보다는 DebugFS를 사용할 것을 권장한다.

1) DEBUG_FS를 위한 kernel config 조정

```
Kernel hacking -> Debug Filesystem
```

2) 타겟 보드에서 mount하여 사용

```
mount -t debugfs /debug /sys/kernel/debug
```

- /sys/kernel/debug 아래에서 각종 디버그 정보 확인 가능함.

또한 printk() 함수를 직접 사용하기 보다는 pr_*() 및 dev_*() 함수를 사용하는 편이 효과적이라고 말할 수 있다.

1) pr_*() 함수(include/linux/printk.h)

```
pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warning(), pr_notice(), pr_info(), pr_cont(), pr_debug()
```

2) dev_*() 함수(include/linux/device.h)

```
dev_emerg(), dev_alert(), dev_crit(), dev_err(), dev_warning(), dev_notice(), dev_info(), dev_dbg()
```

이 밖에도 커널의 상태를 보다 쉽게 확인하기 위해서는 busybox 등을 설치하는 것이 문제 해결에 도움이 된다.

2. Kernel Debug Message 추출 방법

이번 절에서는 커널 크래쉬(crash) 상황 발생 시, 어떻게든 dmesg 결과를 추출할 수 있는 방법에 관하여 고민해 보고자 한다.

2.1 kdump/kexec

커널에 장애가 발생한 경우 커널 디버그 메시지를 확보하는 일은 문제의 원인을 쉽게 풀기 위해 반드시 거쳐야 하는 매우 중요한 과정이라고 볼 수 있다. Kdump/kexec 기법은 이를 위한 효과적인 방법으로써 아래와 같은 절차를 따른다.

0) kdump/kexec가 동작하기 위해서는 두 개의 kernel(standard, debug kernel)이 준비되어야 한다.

1) Standard kernel은 crash kernel argument를 cmdline에 추가한 상태로 부팅하게 되며, debug kernel을 위한 메모리 공간을 확보한 후, debug kernel을 이 곳에 복사한다.

2) Standard kernel이 동작하던 중, panic이 발생하면, kexec가 debug kernel을 구동(warm reboot) 시키게 된다.

3) Debug kernel의 부팅이 완료되면, 패닉이 발생한 커널(standard kernel)의 메모리 내용을 저장 공간(sdcard 등)에 백업(kdump 이용)을 한다. 이것이 가능한 이유는 warm reboot으로는 memory의 내용이 지워지지 않기 때문이다.

4) 시스템은 다시 standard kernel을 사용하여 재 부팅(이번에는 cold reboot임)하게 되고, 3단계에서 백업해 둔 내용을 사용하여 디버깅을 시작한다.

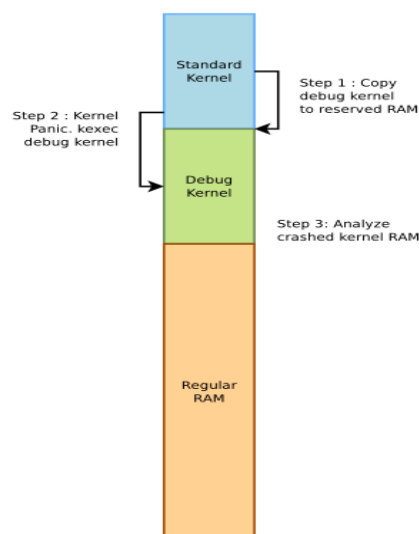


그림 7-2 Kexec와 Kdump의 개념도 [출처 - 참고 문헌 5]

Kdump/kexec 기법은 (필자가 아는 한) 안드로이드에서는 실제로 사용되지는 않고 있으며, Ram Console 이라는 비슷한 개념이 이를 대신하고 있다.

2.2 Android Ram Console

Ram Console이란 커널 디버그 메시지를 RAM buffer에 저장해 두었다가, 커널 패닉 발생 시, 다음 번 부팅 시점에서 /proc/last_kmsg를 통해 이전 커널 상태를 확인할 수 있도록 해주는 기능을 말한다. 이는 앞 절에서 소개한 kdump/kexec와도 매우 유사한 개념으로 볼 수 있으며, 실제로 안드로이드에서 동작하도록 Google(Robert Love가 만듦)에서 고안한 기법이다. 관련 파일로는 아래의 파일이 있다.

- *arch/arm/mach-*/board_*.c*("ram console"로 검색)
- *drivers/staging/android/ram_console.c, persistent_ram.c* 등

1) 초기 부팅 시, 커널은 디버그 메시지를 위해 일정량의 메모리 공간을 추가로 확보해 두고, 커널 로그를 이 곳에도 동일하게 저장해 나간다.

2) 커널 패닉이 발생하면, watchdog이 동작하여 시스템이 warm boot하게 된다.

3) 시스템이 다시 정상 부팅한 후, "cat /proc/last_kmsg" 명령을 실행해 보면, 이전 부팅 상태의 kernel message를 확인할 수 있게 된다. 따라서 이전에 kernel panic이 발생했다면 그 이유를 어느 정도는 짐작이 가능할 것이다.

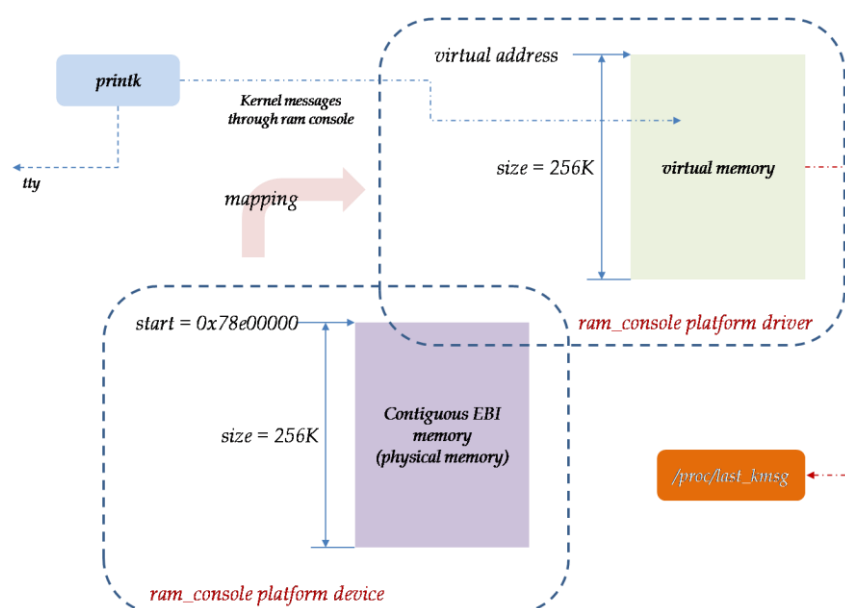


그림 7-3 Android Ram Console 개념(1)

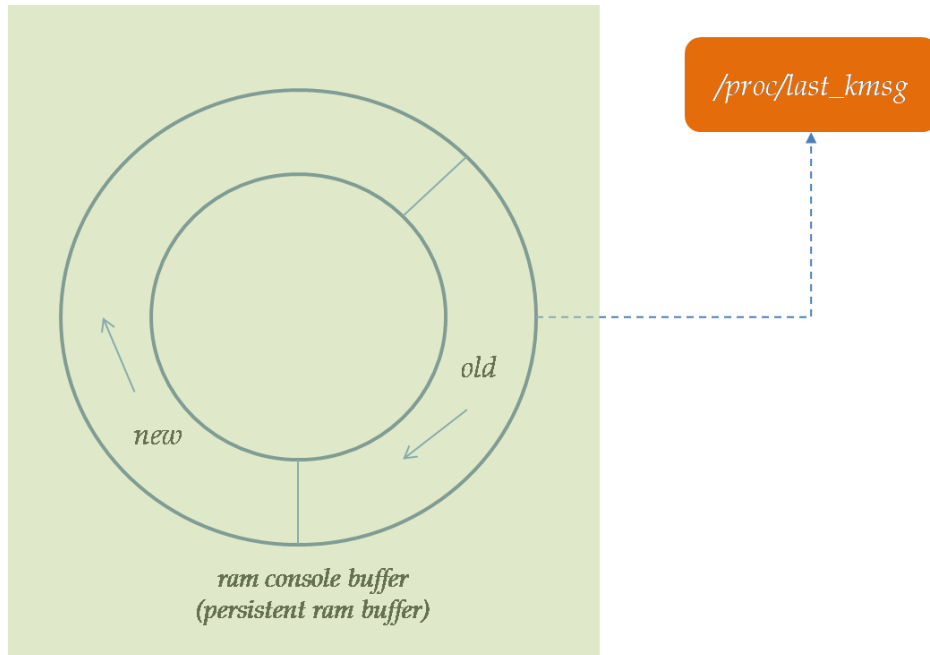


그림 7-4 Android Ram Console 개념(2)

2.3 Ram Dump

Ram Dump는 앞서 소개한 Ram Console과 매우 유사한 기능으로, 보다 폭 넓은 디버깅 정보 (dmesg, task 정보, work queue 정보 등)를 효과적으로 추출할 수 있는 방법이다. 실제로 이를 위한 코드가 인터넷에 오픈되어 있는 것은 아니지만, 충분히 구현하여 활용이 가능한 방법이기여 여기에서 그 개념만을 간략히 소개해 보고자 한다.

<Ram Dump의 원리>

- 1) 커널 crash 상황(kernel panic or oops)이 발생한다.
- 2) 일정 시간 경과 후, watchdog이 동작하여 시스템이 자동으로 reset(warm reboot)된다.
- 3-4) 부트로더가 구동되면서, 부팅 상황을 보고 정상 부팅일 경우는 커널을 구동시키는 루틴으로 분기하고, 비정상 부팅(watchdog에 의한 부팅)일 경우에는 부팅을 멈춘다.
 - (숙제 1) 이 경우 정상 부팅인지 아닌지를 판단하는 코드가 부트로더에 추가되어야 함.
- 5) 호스트 PC와 타겟 보드가 USB로 연결되어 있을 경우, 전용 프로그램을 통해 타겟 보드(phone)의 RAM 내용을 dump하여, 디버깅에 활용한다.
 - (숙제 2) 타겟 보드의 RAM을 dump하는 전용 프로그램이 준비되어야 함.
 - 구현이 용이하지 않을 경우, 부트로더에서 제공하는 memory dump(예: u-boot md 명령)를 활용하여도 어느 정도 추출이 가능하다.

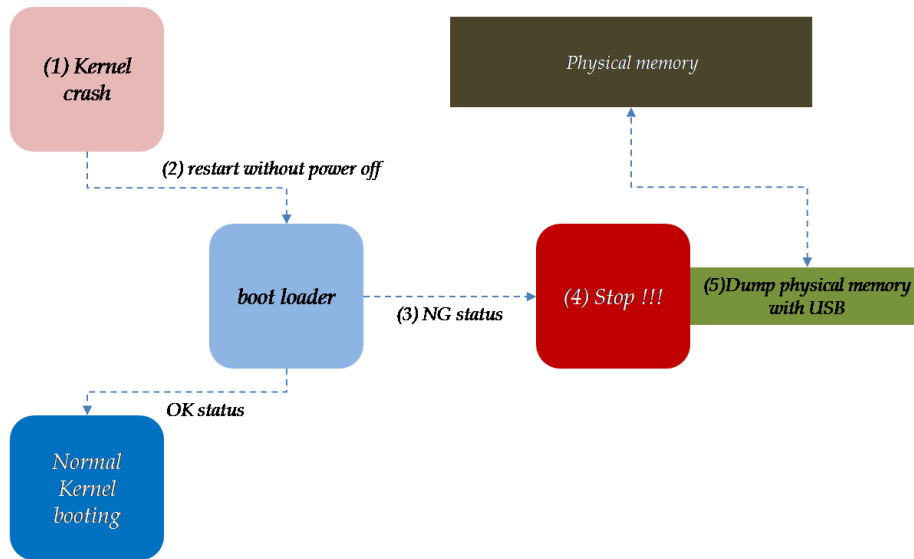


그림 7-5 Ram Dump의 개념

이상의 과정을 통해 문제가 되는 커널의 RAM 내용을 dump하였으면, 이를 토대로 원하는 정보 즉, kernel debug message(그림 7-6), task 목록 정보(그림 7-7), work queue 상태 정보(그림 7-8)를 추출할 수 있게 된다.

그림 7-6은 RAM dump 파일(physical memory에 대한 dump 결과 파일)로부터 kernel debug message를 추출하는 절차를 요약한 그림이다.

1) kernel/print.k 파일에 있는 “__log_buf”는 kernel debug message가 저장되는 시작 번지에 해당하는 부분으로 이 값과 버퍼의 크기 즉, “sizeof(__log_buf)”를 vmlinux(ELF format)에서 검색하여 관련 심볼에 해당하는 주소(여기서는 가상 주소임)를 얻어 낸다.

2) Virtual 주소를 Physical 주소로 바꾼다.

3) 2에서 얻은 Physical 주소를 RAM dump한 부분에서 찾아 그 크기 값만큼을 추출해 내면, 이것이 곧 kernel debug message가 된다. 따라서 이를 별도의 text 파일로 저장한다.

4) 이후, 이 text 파일을 분석하여 kernel crash의 원인을 분석한다.

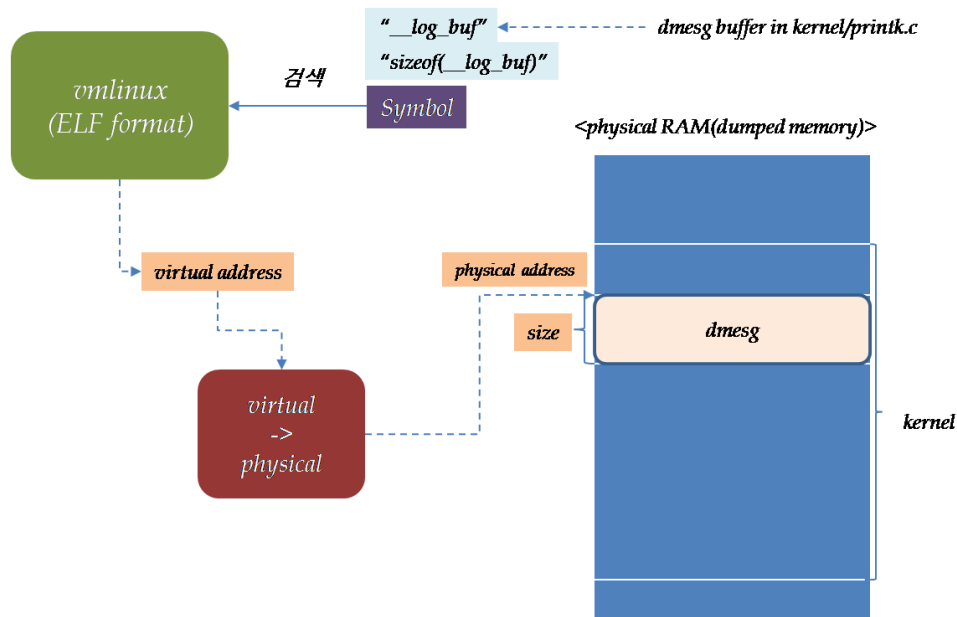


그림 7-6 Ram Dump 후, kernel debug message 추출 과정

<커널 debug message 추출 예>

```
<3>[03-06 18:00:50.348] msm_release_ion_client Calling ion_client_destroy
<3>[03-06 18:00:50.348] ion_iommu_delayed_unmap: Virtual memory address leak in domain 1, partition 2
<3>[03-06 18:00:50.348] ion_iommu_delayed_unmap: Virtual memory address leak in domain 1, partition 2
...
<3>[03-06 18:00:50.348] ion_iommu_delayed_unmap: Virtual memory address leak in domain 1, partition 2
<3>[03-06 18:00:50.348] ion_iommu_delayed_unmap: Virtual memory address leak in domain 1, partition 2
<3>[03-06 18:00:50.348] ion_iommu_delayed_unmap: Virtual memory address leak in domain 1, partition 2
<3>[03-06 18:00:50.348] ion_iommu_delayed_unmap: Virtual memory address leak in domain 1, partition 2
[03-06 18:01:03.571] select 26141 (obile.backup.lg), adj 647, size 7750, to kill
[03-06 18:01:03.571] send sigkill to 26141 (obile.backup.lg), adj 647, size 7750
[03-06 18:01:03.581] timed_output vibrator: lge_isa1200_hw_vib_on_off(0), level = 84
[03-06 18:01:03.601] timed_output vibrator: lge_isa1200_hw_vib_on_off(1), level = 84
[03-06 18:01:03.621] msm_camera_v4l2_streamoff: hw failed to stop streaming
[03-06 18:01:03.621] Unable to handle kernel NULL pointer dereference at virtual address 00000019
[03-06 18:01:03.621] pgd = d3218000
[03-06 18:01:03.621] [00000019] *pgd=00000000
[03-06 18:01:03.621] Internal error: Oops: 5 [#1] PREEMPT SMP ARM
[03-06 18:01:03.621] Modules linked in: vpnclient(O) bthid
[03-06 18:01:03.621] CPU: 0 Tainted: G W O (3.4.0-perf-gd324e58-00004-ge53ac7f #1)
[03-06 18:01:03.621] PC is at mutex_spin_on_owner+0x3c/0x5c
[03-06 18:01:03.621] LR is at mutex_spin_on_owner+0x14/0x5c

[03-06 18:01:03.621] [<c00ee0b0>] (mutex_spin_on_owner+0x3c/0x5c) from [<c07c1490>]
(__mutex_lock_slowpath+0x40/0x2a4)
[03-06 18:01:03.621] [<c07c1490>] (__mutex_lock_slowpath+0x40/0x2a4) from [<c07c1714>] (mutex_lock+0x20/0x40)
[03-06 18:01:03.621] [<c07c1714>] (mutex_lock+0x20/0x40) from [<c03bc4e4>] (ion_unmap_iommu+0x1c/0xf0)
[03-06 18:01:03.621] [<c03bc4e4>] (ion_unmap_iommu+0x1c/0xf0) from [<c0536c6c>]
(videobuf2_pmem_contig_user_put+0x2c/0x50)
[03-06 18:01:03.621] [<c0536c6c>] (videobuf2_pmem_contig_user_put+0x2c/0x50) from [<c054a12c>]
```

```

(msm_vb2_ops_buf_cleanup+0x384/0x3bc)
[03-06 18:01:03.621] [<c054a12c>] (msm_vb2_ops_buf_cleanup+0x384/0x3bc) from [<c0533254>]
(__vb2_queue_free+0x4c/0x158)
[03-06 18:01:03.621] [<c0533254>] (__vb2_queue_free+0x4c/0x158) from [<c0533c1c>] (vb2_reqbufs+0x21c/0x38c)
[03-06 18:01:03.621] [<c0533c1c>] (vb2_reqbufs+0x21c/0x38c) from [<c05412b0>]
(msm_camera_v4l2_reqbufs+0x6c/0x23c)
[03-06 18:01:03.621] [<c05412b0>] (msm_camera_v4l2_reqbufs+0x6c/0x23c) from [<c0528e44>]
(__video_do_ioctl+0x1ad0/0x4f80)
[03-06 18:01:03.621] [<c0528e44>] (__video_do_ioctl+0x1ad0/0x4f80) from [<c052712c>] (video_usercopy+0x354/0x4b0)
[03-06 18:01:03.621] [<c052712c>] (video_usercopy+0x354/0x4b0) from [<c0526408>] (v4l2_ioctl+0xe0/0x114)
[03-06 18:01:03.621] [<c0526408>] (v4l2_ioctl+0xe0/0x114) from [<c0198290>] (do_vfs_ioctl+0x55c/0x5d0)
[03-06 18:01:03.621] [<c0198290>] (do_vfs_ioctl+0x55c/0x5d0) from [<c0198338>] (sys_ioctl+0x34/0x54)
[03-06 18:01:03.621] [<c0198338>] (sys_ioctl+0x34/0x54) from [<c000dec0>] (ret_fast_syscall+0x0/0x30)
[03-06 18:01:03.621] Code: 1a000005 e5942010 e1520005 1a000002 (e5952018)
[03-06 18:01:03.691] ---[ end trace 0da3e76b87a358c7 ]---
[03-06 18:01:03.691] Kernel panic - not syncing: Fatal exception
[03-06 18:01:03.691] subsys_q6_crash_shutdown
[03-06 18:01:03.691] subsystem-fatal-8x60: Q6 NMI was sent.
[03-06 18:01:03.691] Rebooting in 5 seconds..set_dload_mode : 0
[03-06 18:01:03.691] restart time : 2013-03-06 09:01:09.805921556 UTC
[03-06 18:01:03.691] Going down for restart now

```

비슷한 방법으로 kernel crash 시점에 동작 중이던 task 목록 정보를 추출할 수가 있게 된다. 이때 task 목록 정보를 추출하는데 사용하는 심볼은 arch/arm/kernel/init_task.c 파일에 있는 "init_task" 스트링이다.

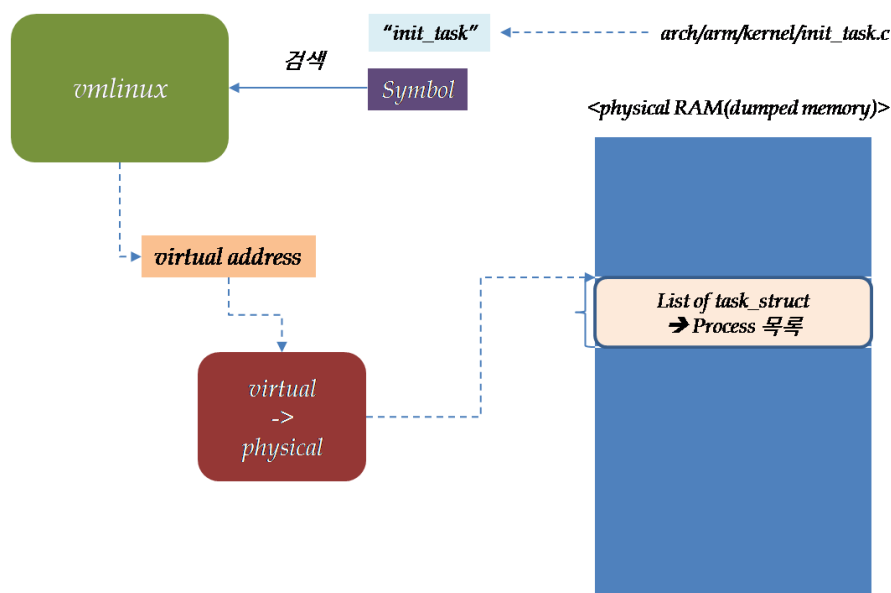


그림 7-7 Ram Dump 후, task 목록 정보 추출 과정

Kernel(혹은 device driver) 2.6 ~ 3.x에서는 work queue를 매우 자주 사용하는 것이 특징이다. 따라서 work queue 관련 상태 정보를 확인함으로써 문제의 원인을 해결할 가능성이 커지게 된다.

앞서의 방법들과 마찬가지로 그림 7-8의 내용처럼 여러 심볼을 활용하여 work queue 정보 추출이 가능하다.

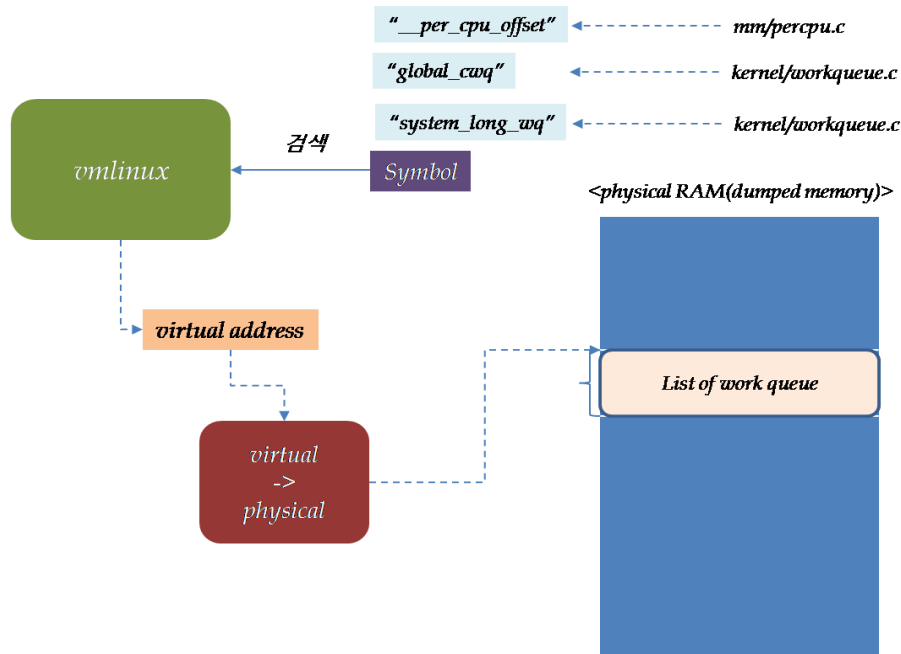


그림 7-8 Ram Dump 후, workqueue 정보 추출 과정

3. KGDB - S/W 방식 디버깅 기법

이번 절에서는 kernel crash 상황이 재현 가능할 경우, S/W 기법만을 사용하여 이를 찾아내는 방법에 관하여 고민해 보고자 한다. 이번 절에서 소개하는 방법은 앞서 2절의 방법을 동원하여 dmesg를 추출한 경우나, 그렇지 못한 경우 모두에 적용이 가능한 방법으로 드라이버를 일부 수정해야만 한다.

3.1 kgdb & USB debugging

KGDB는 printk와 더불어 kernel debugging을 위한 가장 필수적인 도구라고 볼 수 있다. KGDB를 사용하려면, 호스트 PC 상에 GDB를 실행하고, target 장치의 KGDB와 연결해야 한다. KGDB에 관한 기본 정보는 <http://kgdb.linsyssoft.com>에서 확인이 가능한데, KGDB는 크게 두 부분 즉, kgdb core(kernel/kgdb.c)와 연결 인터페이스(tty 혹은 Ethernet)로 이루어져 있다. KGDB는 x86, ARM 등 대부분의 아키텍처에서 지원이 되고 있지만, Android에서는 효과적으로 지원되고 있지 않은 실정이다. Android에서 KGDB를 사용하는 것이 어려운 이유는 KGDB가 요구하는 serial port나 ethernet port가 android 단말에는 없기 때문이다(serial port의 경우는 개발 초기에는 있으나, 대개 양산 버전에서는 빠지게 됨). 따라서, 이를 대체할 수 있는 인터페이스인 USB 포트를 사용하여

KGDB가 동작할 수 있도록 하는 code 추가 작업이 필요하다고 볼 수 있다.

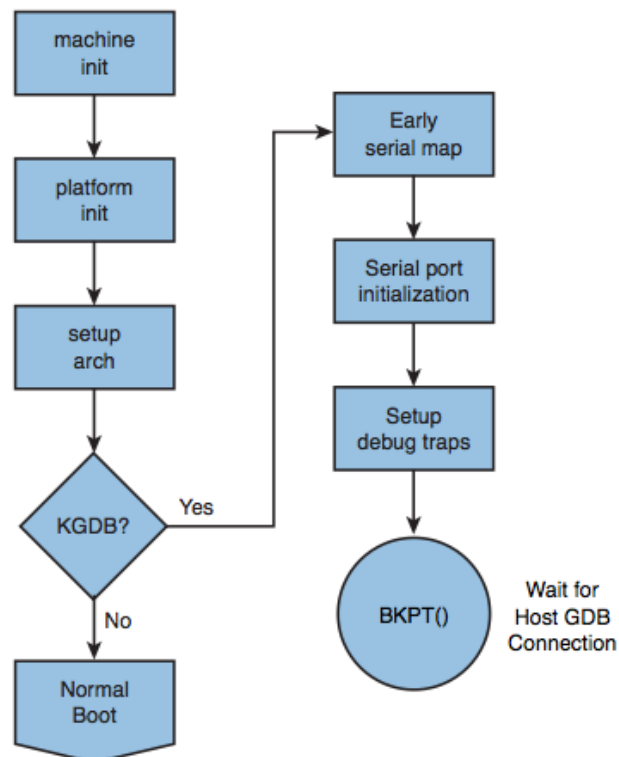


그림 7-9 KGDB 코드 흐름 [출처 - 참고 문헌 2]

아래에 소개하는 두 가지 방법은 USB 인터페이스를 사용하여 KGDB를 사용하는 내용을 정리한 것이다.

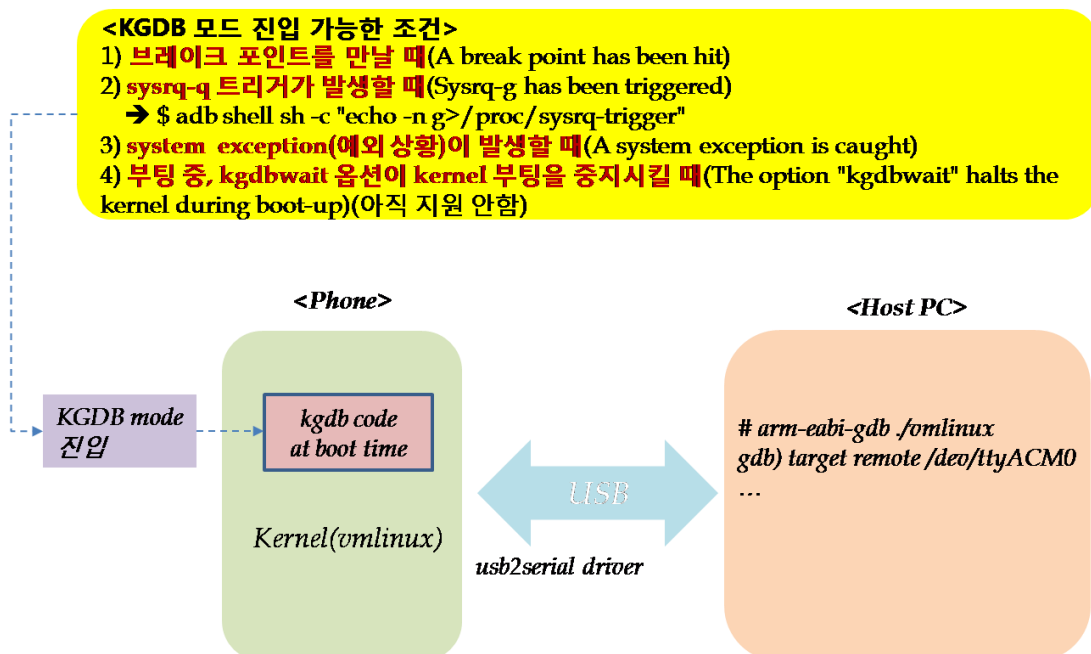


그림 7-10 KGDB를 이용한 kernel debugging 방법 개요

3.2 USB/Serial 기반 KGDB 사용 방법(참고 문헌 [6])

소개

USB/Serial 기반의 KGDB 사용 방법을 정리하면 아래와 같다.

- 1) *kgdb* <-> *serial* 장치(*tty*) <-> *USB*를 활용한 방법이다.
- 2) *serial driver*를 사용한 방법이라 속도가 느릴 수 있다.
- 3) *Qualcomm MSM* 칩에서만 테스트하였으나, 다른 플랫폼에 반영하는 것이 어렵지는 않다.
- 4) *adb*와 동시 사용 불가하다.

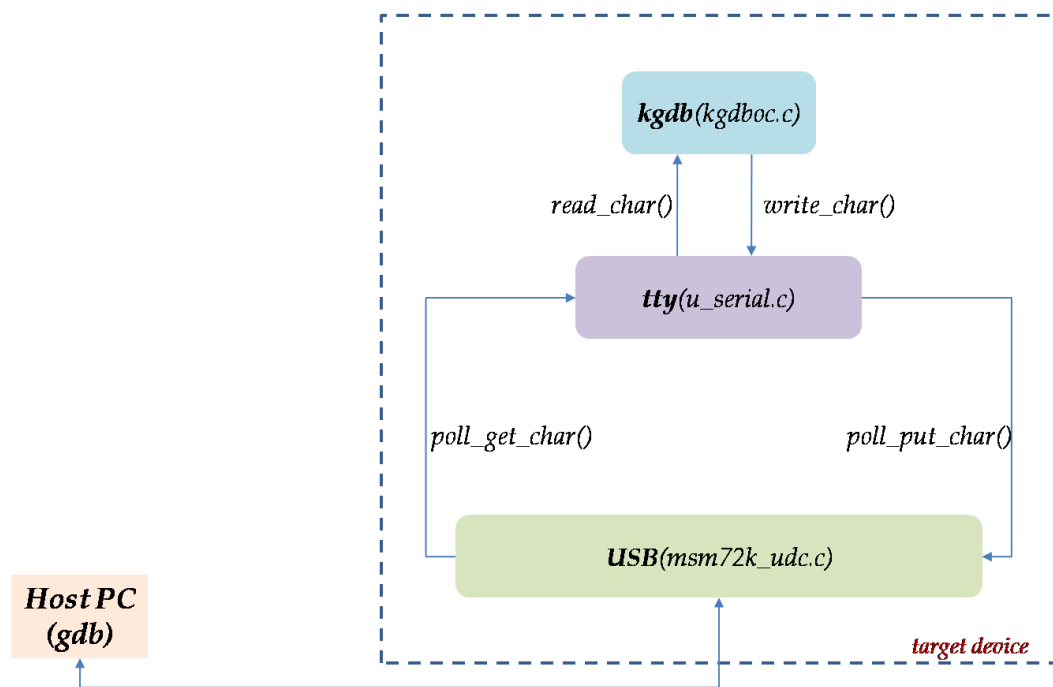


그림 7-11 USB/Serial 기반 KGDB 방법의 내부 구조

USB/Serial 기반의 KGDB 사용 방법과 관련한 소스 코드는 아래 사이트에서 확인이 가능하며, 이를 기초로 하여 USB/Serial 기반의 KGDB 사용 방법을 아래에 정리하였다.

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

<Android에서 KGDB를 사용하기 위한 드라이버 작업 요약>

1) Kernel patch 후, kernel build하기(다음의 configuration을 켜)

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 보드 초기화 파일을 수정할 필요가 있음.

```
#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,
};
```

3) USB serial & gadget 드라이버 수정 작업

a) usb serial 가젯 드라이버(driver/usb/gadget/u_serial.c)가 호스트 PC의 GDB client와 통신하기 위해서는 poll_init(), poll_get_char(), 및 poll_put_char() 등의 함수가 새롭게 필요하였다.

```
static const struct tty_operations tty_ops = {
    [... normal ops like open/close/put_char ...]
    .poll_init = gs_poll_init,
    .poll_get_char = gs_poll_get_char,
    .poll_put_char = gs_poll_put_char,
}
```

b) drivers/gadget/msm72k_udc.c 수정 - USB loop poll code 구현(실제로 USB 장치로 data를 내

보내거나, USB 장치에서 data를 읽어 들이는 코드임)

```
int usb_loop_poll_hw(struct usb_ep *_ept, int is_rx)
{
    struct msm_endpoint *act_ept, *ept = to_msm_endpoint(_ept);
    struct usb_info *ui = ept->ui;
    int done = 0;
    u32 n;

    /* Normally there is a read request in the endpoint, wait for new data */
    for (;;) {
        n = readl(USB_USBSTS);
        writel(n, USB_USBSTS);
        if (n & STS_UI) /* finished transaction */
            break;
    }

    /* USB Transaction is complete */
    if (n & STS_UI) {
        n = readl(USB_ENDPTSETUPSTAT);
        if (n & EPT_RX(0))
            handle_setup(ui);

        n = readl(USB_ENDPTCOMPLETE);
        writel(n, USB_ENDPTCOMPLETE);

        while (n) {
            unsigned bit = __ffs(n);
            act_ept = ui->ept + bit;
            if (ept == act_ept) {
                pr_debug("%s: recv'd right tx %d\n", __func__, bit);
                done = 1;
            }
            else {
                pr_debug("%s: recv'd extra tx from ept %d (exp %d)\n",
                        __func__, bit, ept->bit);
            }
        }
        /* always call the handler for KGDB and other usb functions.
         * this is to avoid hardware timeout, but can leave a bit
```

```

        * kernel code running when kgdb is invoked to stopped the
        * kernel. this works quite well with adb but might not
        * support usb mass storage devices very well.
        */
        handle_endpoint(ui, bit);
        n = n & ~(1 << bit);
    }
}

return done ? 0 : -EAGAIN;
}

```

4) Kernel command line 수정하기

a) kgdboc 장치로 ttyGS0를 지정함. Android full build 시에도 동작이 가능하도록 하기 위해서는 BoardConfig.mk 파일에 kernel command line을 지정하는 부분이 있으니, 이를 수정하면 됨.

b) 두 번째 옵션인 "kgdbretry=4"은 drivers/serial/kgdboc.c 파일에 새롭게 추가된 파라미터로, kgdb가 부팅 초반 부에서 ttyGS0 장치를 발견 못하게 되면, 지정된 시간(kgdbretry)이 지난 후, 다시 시도하라는 의미를 내포하고 있다. 이는 부팅 시, USB 장치가 빨리 초기화되지 않기 때문에 추가된 실험 코드라고 볼 수 있다.

```
kgdboc=ttyGS0 kgdbretry=4
```

<kgdb 사용하기>

1) KGDB 모드에 진입하기(Sysrq-g 트리거 사용)

```
$ adb shell sh -c "echo -n g>/proc/sysrq-trigger"
```

2) gdb로 연결하기

```
$ 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_setWn"
```

```
(gdb)
```

3.3 단독 USB 기반 KGDB 사용 방법(참고 문헌 [7])

소개

앞서 3.2절에서 소개한 USB/Serial 기반의 KGDB 방법에 비해, 이번 절에서 소개하는 단독 USB 기반의 KGDB 사용 방법은 아래와 같은 특징을 갖는다.

1) 속도 향상을 위해 *tty(serial)* 드라이버를 활용하지 않고, *USB* 드라이버만을 사용하여 속도의 향상을 꾀하였다.

- 별도의 KGDB *usb gadget* 드라이버를 만들(그림 7-14 참조).

2) *adb*와 동시에 사용(응용 계층과 *kernel*의 동시 *debugging*이 가능해짐)이 가능하며, 커널 로그를 볼 수 있는 *console* 기능을 제공하고 있다(그림 7-13 참조).

- 이를 위해 공개 소프트웨어인 *agent-proxy*를 수정하여 *android-agent-proxy*를 만들.
- 커널 *debugging*과 커널 메시지를 분리시킴.

3) 여러 플랫폼 즉, *Qualcomm*, *TI*, *Samsung* 등에서 동작 테스트를 하였다.

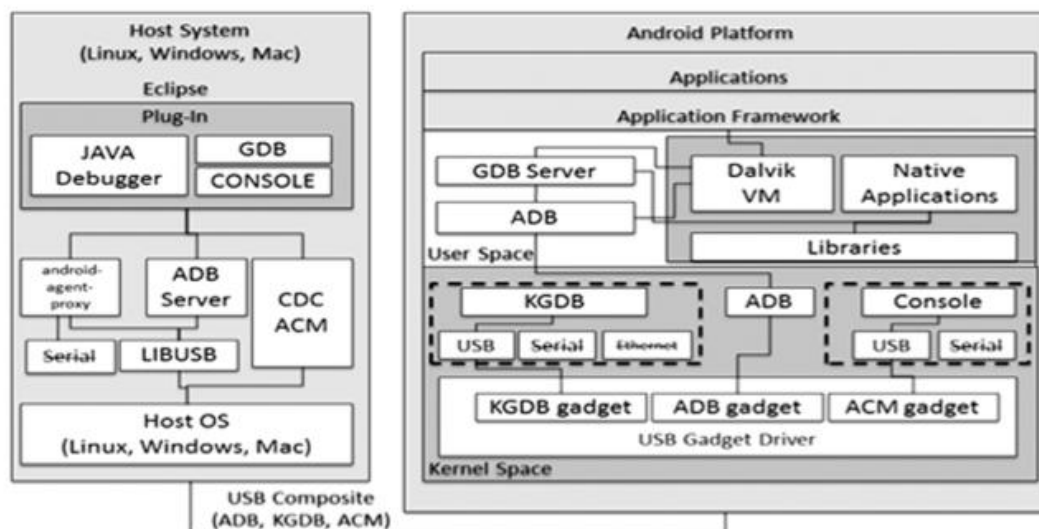


그림 7-12 USB 기반의 KGDB 사용 방법의 전체 구조 [출처 - 참고 문헌 7]

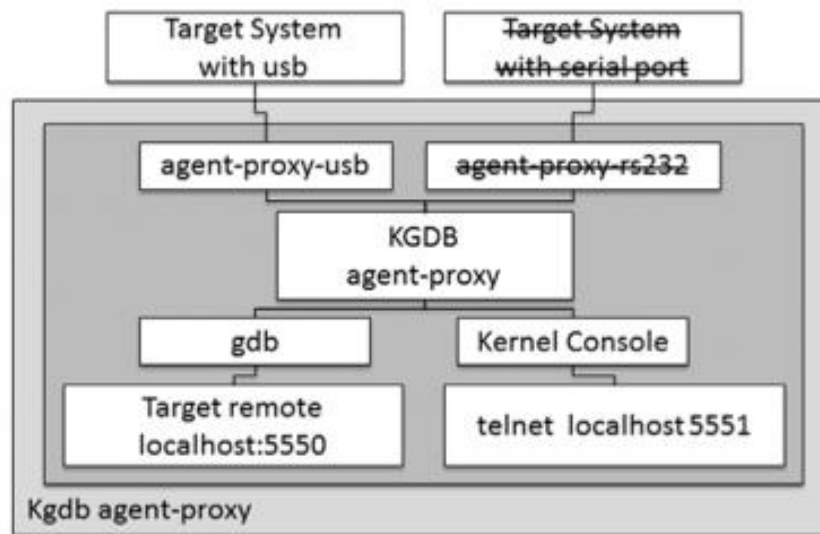


그림 7-13 KGDB agent-proxy(gdb와 kernel console로 분기) [출처 - 참고 문헌 7]

그림 7-14는 drivers/usb/gadget/f_kgdb.c, kgdb_io_usb.c와 msm72k_udc.c 코드와 관련된 부분으로, kgdb gadget 드라이버(아래 그림에서 Linux KGDB 부분)에서 kgdb io usb 루틴을 통해 kgdb write 혹은 read operation(polling 방식임 - 그림 7-15 참조)을 수행하고, platform usb driver(msm72k_udc.c 내에 있음)를 사용하여 interrupt handler(usb_interrupt() 함수)를 호출하는 형태로 구성되어 있다.

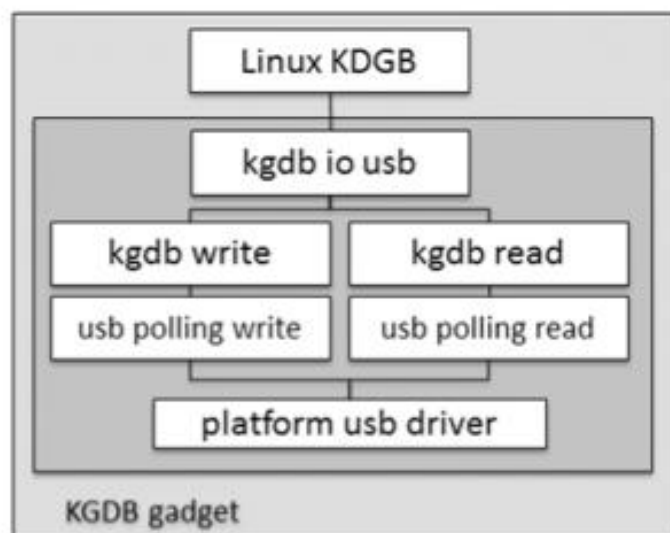


그림 7-14 KGDB USB gadget 드라이버 구조 [출처 - 참고 문헌 7]

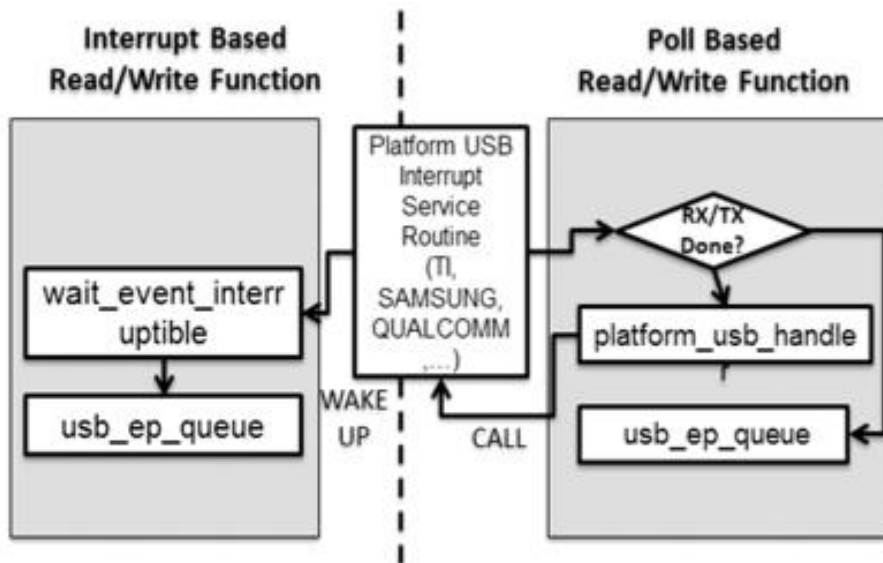


그림 7-15 Polling 방식의 USB gadget 드라이버 내부 구조 [출처 - 참고 문헌 7]

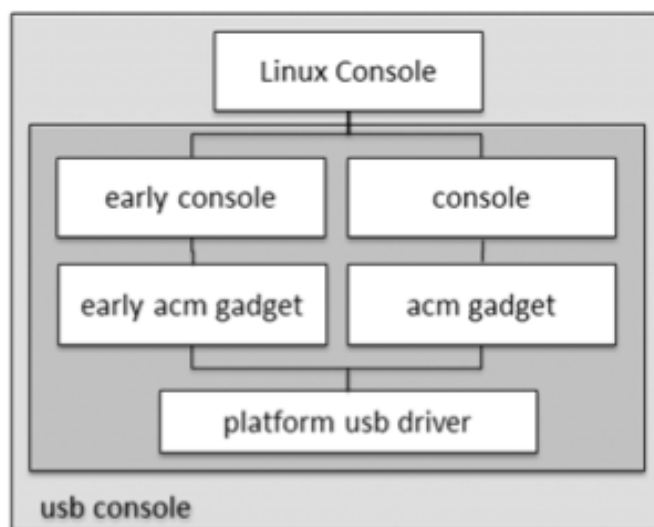


그림 7-16 USB Console 드라이버 [출처 - 참고 문헌 7]

android-agent-proxy와 arm-eabi-gdb를 사용하여 debugging하는 방법을 정리하면 다음과 같다.

```
$ sudo ./android-agent-proxy 5550^5551 0 v
```

```
$ arm-eabi-gdb vmlinux
```

```
...
```

```
(gdb)
```

```
...
```

gdb를 사용하는 것이 가능하므로 ddd 역시 사용이 가능하다. 아래 그림을 참조하기 바란다.

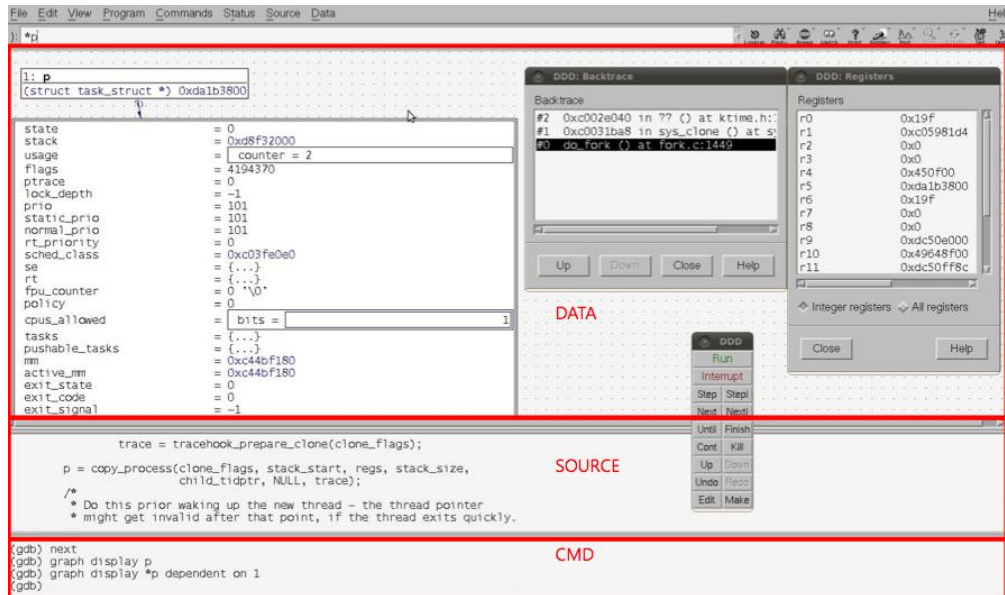


그림 7-17 ddd를 이용한 디버깅 화면 [출처 - 참고 문헌 8]

지금까지 설명한 USB 기반의 KGDB 사용 방법 관련 코드는 아래 사이트에서 확인이 가능하다.

- <https://sourceforge.net/projects/usbdevicesupport>

마지막으로, KGDB를 활용한 방법은 USB 장치가 인식되기 전에 시스템이 죽어 버리는 경우에는 사용할 수 없는 방법이지만, 반대의 경우에는 매우 효과적인 방법으로 볼 수 있다. 특히, 문제의 재현 경로를 확실히 알 수 있는 경우 및 양산 보드에서 효율 가치가 높은 디버깅 기법이라고 말할 수 있다.

4. JTAG H/W 기반 Debugging 기법

이번 절에서는 3절에서 설명한 KGDB 기법의 한계를 극복할 수 있는 JTAG 기반의 H/W 디버깅 기법에 관하여 소개하고자 한다. 안드로이드에서 사용 가능한 JTAG 기반의 디버깅 기법으로는 아래와 같이 크게 두 가지로 분류가 가능하다.

1) OpenOCD(Open On-Chip Debugger) 방식

- 오픈 소스 프로젝트인 OpenOCD를 지원하는 JTAG dongle이 필요함(저렴한 가격임)
- 대부분의 ARM 플랫폼을 지원하는 장점이 있음.
- GDB(Eclipse, ddd 사용 가능) 기반의 디버깅이 가능함.

2) TRACE32 방식

- 고가의 전용 하드웨어 및 소프트웨어 패키지를 구매해야 함.

- 디버깅에 매우 효과적인 UI 등 다양한 기능 지원함.
- 현존 최고의 디버깅 도구라 볼 수 있음.

JTAG은 Joint Test Action Group으로 최초 PCB나 IC 등을 검사하기 위한 목적으로 만든 IEEE 1149.1 표준이다. 지금은 반도체 설계 시 대부분 JTAG 모듈이 포함되어 기본 5개 테스트 선(TCK, TDI, TDO, TMS, TRST)을 사용해서 테스트 및 디버깅에 활용하도록 한다. JTAG으로 할 수 있는 일을 간단히 말하자면, CPU가 가진 다양한 레지스터들의 데이터를 읽고 쓸 수 있다는 것이다. 이것은 디버거를 통해 아래의 두 가지 일을 가능하게 해 준다.

1) Target 장치 Halt, Resume

- 가장 중요한 기능으로 ARM의 경우 Debug_Test, 및 ICE Breaker등의 레지스터를 통해 Target 장치를 원하는 시점에 Go/Break가 가능하도록 함.

2) Register 및 Memory Access

- ARM CPU, peripheral, memory access가 가능하며, 읽고 쓰기가 가능함.

결국 JTAG과 이를 이용한 디버거는 Target 장치에 대한 go/break가 가능하게 하며 데이터를 읽고 쓰는 테스트를 가능하게 해 주는 것이다. 그림 7-18과 7-19는 지금까지 설명한 JTAG의 내부를 그림으로 표현한 것이다.

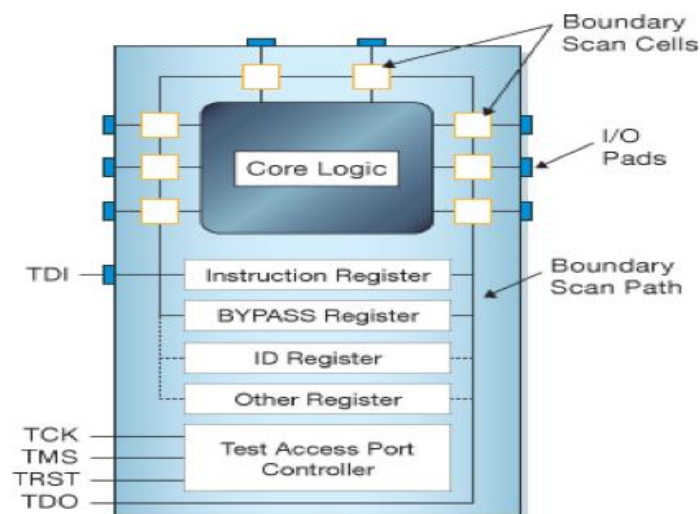


그림 7-18 JTAG(1) [출처 - 참고 문헌 11]

1. TDI (Test Data In)
2. TDO (Test Data Out)
3. TCK (Test Clock)
4. TMS (Test Mode Select)
5. TRST (Test ReSeT) optional.

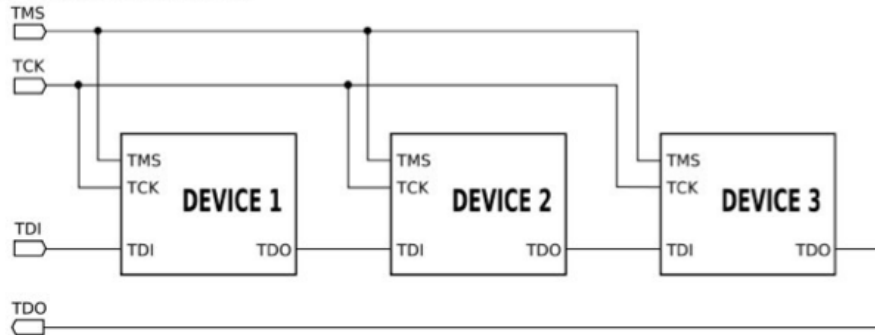


그림 7-19 JTAG(2) [출처 - 참고 문헌 11]

4.1 OpenOCD

OpenOCD(Open On-Chip Debugger)는 Dominic Rath가 박사 학위 논문을 작성하기 위해 처음 만들었으며, 그 후 전세계의 많은 소프트웨어 및 하드웨어 개발자들의 지지를 받으며 오픈 소스 프로젝트로 성장하였다(<http://openocd.berlios.de/web/>).

아래 그림 7-20은 Beagleboard에 OpenOCD를 지원하는 JTAG dongle을 연결한 그림으로, 개발자는 호스트 PC로부터 나오는 USB 케이블을 JTAG dongle과 연결(그림 우측 부분)하고 있으며, JTAG dongle과 Target 보드(왼쪽)은 리본 케이블(그림 상단 중앙)을 통해 연결되어 있음을 알 수 있다(참고로, 그림 아래 중앙 부분의 케이블은 시리얼 연결 케이블임).

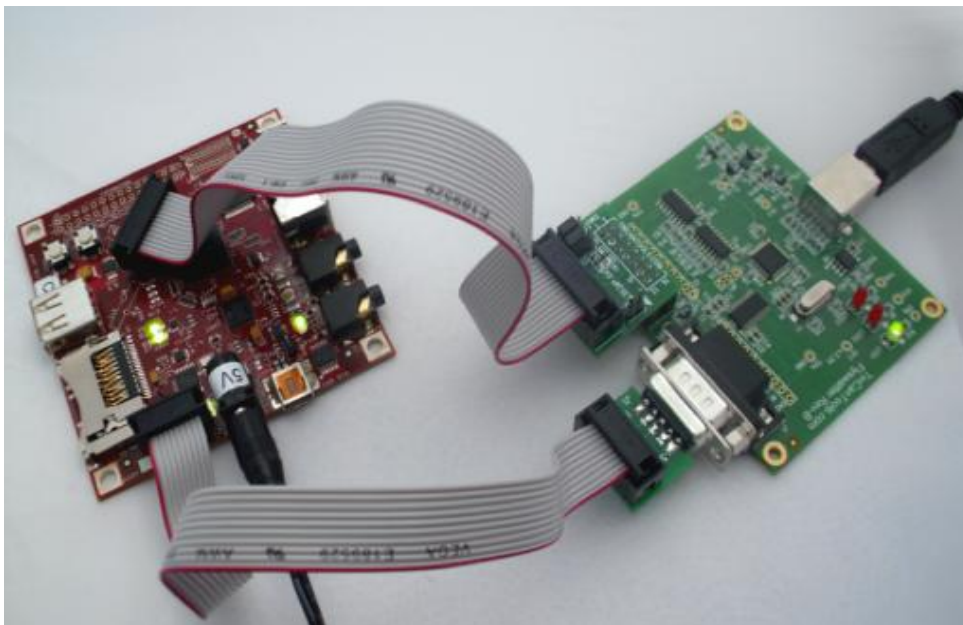


그림 7-20 Flyswatter(JTAG dongle - OpenOCD 지원)과 Beagleboard(오른쪽 보드) [출처 - 참고 문헌 11]

그림 7-21은 Target 장치를 디버깅하기 위해 사용되는 JTAG adapter dongle의 예를 보여준다.



그림 7-21 JTAG dongle 예 [출처 - 참고 문헌 11]

OpenOCD가 하는 역할은 그림 7-22에서도 확인 가능하듯이, JTAG adapter dongle을 통해서 Target Device를 하드웨어적으로 제어 혹은 모니터링하는 것과 gdb나 telnet(client)에 대한 서버 역할을 하는 것으로 이해될 수 있다. 다시 말해 gdb나 telnet은 openocd daemon process(일종의 proxy)를 통해서 Target 장치를 제어하는 것이 가능해진다는 말이다.

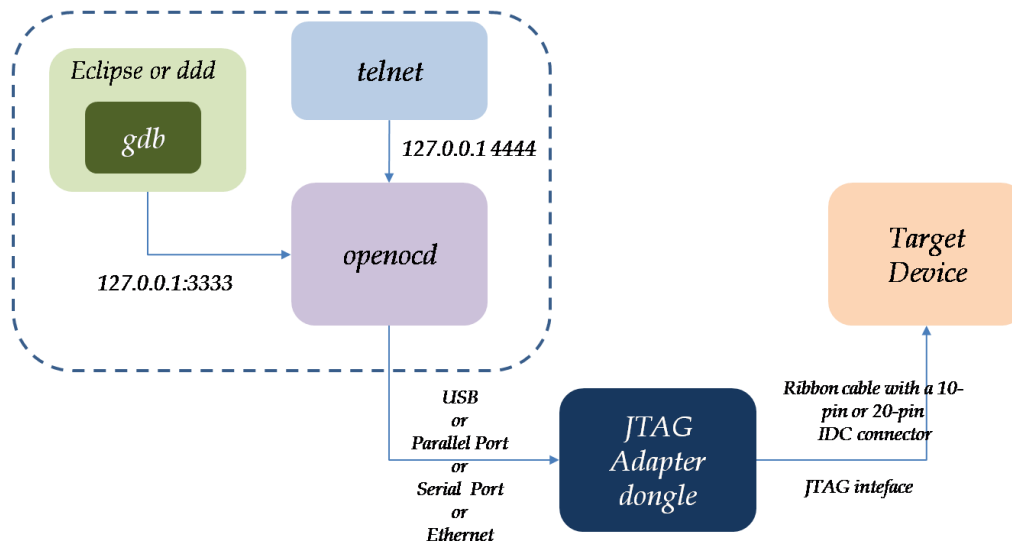


그림 7-22 gdb, openocd, JTAG 간의 관계

Openocd를 실행하는 방법은 아래와 같다.

```
$ openocd -f interface/ADAPTER.cfg -f board/MYBOARD.cfg
```

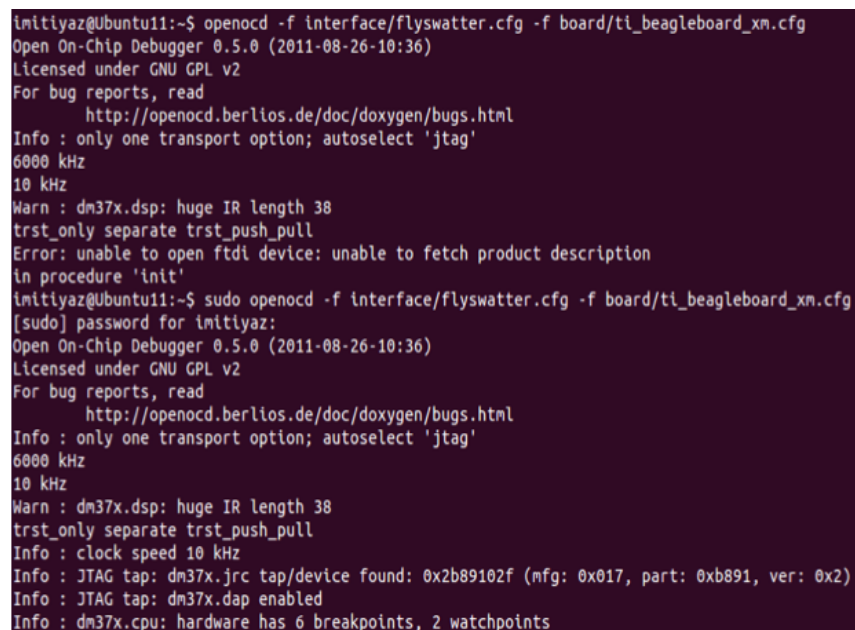
1) interface는 사용하는 debug jtag adapter의 종류를 나타내며,

```
#FLYSWATTER.CFG
interface ft2232
ft2232_device_desc "Flyswatter"
ft2232_layout "flyswatter"
ft2232_vid_pid 0x0403 0x6010
adapter_khz "6000"
```

2) board는 사용하는 타겟 보드를 의미한다.

```
# BeagleBoard xM (DM37x)
# http://beagleboard.org
set CHIPTYPE "dm37x"
source [find target/amdm37x.cfg]
# The TI-14 JTAG connector does not have srst. CPU reset is handled in hardware.
reset_config trst_only
```

그림 7-23은 호스트 PC에서 openocd 서버를 실행하는 예를 보여준다.



```
imtiyaz@Ubuntu11:~$ openocd -f interface/flyswatter.cfg -f board/ti_beagleboard_xm.cfg
Open On-Chip Debugger 0.5.0 (2011-08-26-10:36)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.berlios.de/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
6000 kHz
10 kHz
Warn : dm37x.dsp: huge IR length 38
trst_only separate trst_push_pull
Error: unable to open ftdi device: unable to fetch product description
in procedure 'init'
imtiyaz@Ubuntu11:~$ sudo openocd -f interface/flyswatter.cfg -f board/ti_beagleboard_xm.cfg
[sudo] password for imtiyaz:
Open On-Chip Debugger 0.5.0 (2011-08-26-10:36)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.berlios.de/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
6000 kHz
10 kHz
Warn : dm37x.dsp: huge IR length 38
trst_only separate trst_push_pull
Info : clock speed 10 kHz
Info : JTAG tap: dm37x.jrc tap/device found: 0x2b89102f (mfg: 0x017, part: 0xb891, ver: 0x2)
Info : JTAG tap: dm37x.dap enabled
Info : dm37x.cpu: hardware has 6 breakpoints, 2 watchpoints
```

그림 7-23 openocd 실행 예 [출처 - 참고 문헌 11]

<호스트 PC에서 수행하는 명령 예>

```
$ openocd -f interface/flyswatter.cfg -f board/ti_beagleboard_xm.cfg
$ telnet 127.0.0.1 4444
```

➔ reset, halt, load code, access registers, set break points/watch points ...

```
$ ddd -debugger arm-linux-gdb vmlinux
```

```
(gdb) target remote 127.0.0.1:3333
```

```
(gdb) monitor reset halt
```

정리하자면, OpenOCD는 JTAG dongle을 구입해야 하는 부담(단, 가격이 저렴함)은 있으나, 앞서 3절에서 설명한 USB 기반의 KGDB 방법이 해결하지 못하는 부팅 초기의 디버깅 문제까지도 해결할 수 있는 장점을 갖는다. 앞으로 소개할 상용 버전인 TRACE32에 비하면 기능이 미흡하다고도 볼 수 있으나, 가격대 성능 비를 고려하면 매우 훌륭한 도구임에 틀림없다.

4.2 TRACE32(줄여서 T32라고도 부름)

TRACE32는 독일의 Lauterbach라는 회사에서 개발 판매하는 JTAG 기반의 디버깅 툴이다. TRACE32는 전세계 시장점유율 1위의 임베디드 시스템 개발용 JTAG 에뮬레이터로서 강력한 성능, 다양한 기능, 뛰어난 안정성으로 국내외 이동통신 단말기 (CDMA/GSM) 시장에서 디버거의 표준으로 자리잡아 삼성전자, LG전자 등 휴대폰 개발 업체를 비롯한 전 산업분야에서 표준 개발 장비로 사용되고 있다. TRACE32는 JTAG/BDM, OCDS 등과 같은 모든 On-Chip 디버깅 표준들을 지원한다. MPU와 DSP가 내장된 시스템의 Board Bring-Up에서 C, C++, Java 등의 고급 언어 디버깅, Flash Programming, 코드 품질 검증까지 가능한 고속의 디버깅 장비이며 ARM, PowerPC, MIPS, SH, TI DSP 등 다양한 아키텍처를 지원하도록 Module 방식으로 설계되어 있어 장기 투자에 적합한 경제적인 개발 장비이다.

그림 7-24는 JTAG 인터페이스를 통해 PandaBoard와 TRACE32 장치를 상호 연결한 그림으로, 그림 7-25에 이들의 관계를 다시 표현해 보았다. 또한 그림 7-26은 TRACE32의 Viewer(GUI) 화면을 보여주고 있다.



그림 7-24 TRACE32 장치 연결 [출처 - 참고 문헌 12]

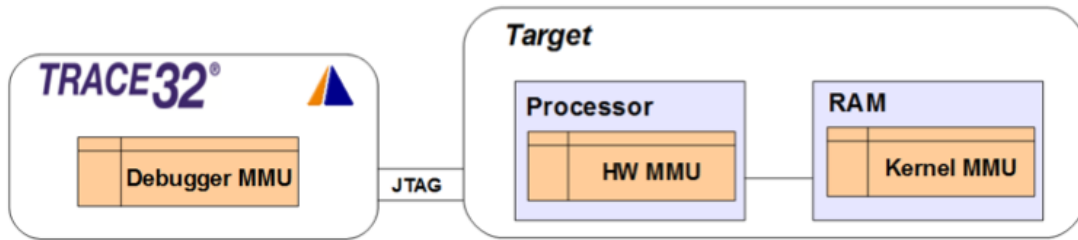


그림 7-25 TRACE32 장치와 JTAG [출처 - 참고 문헌 12]

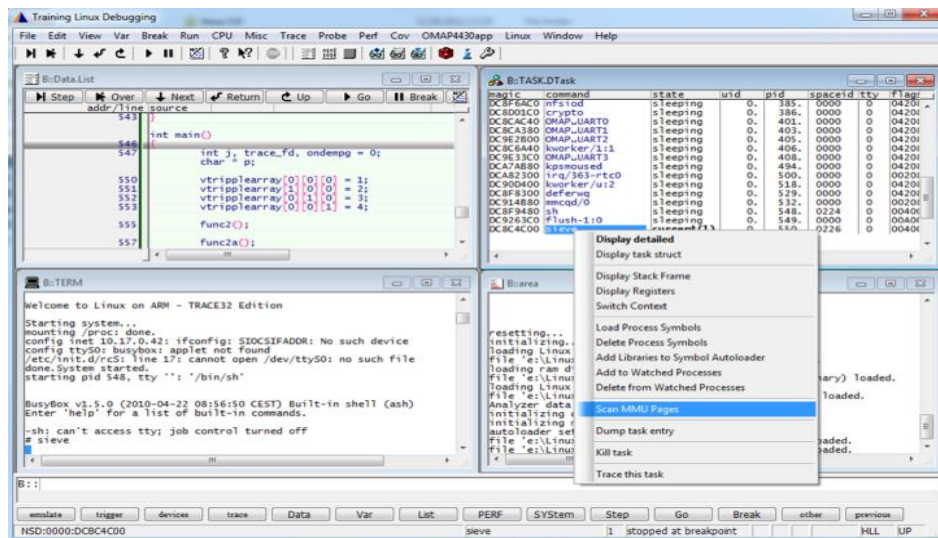


그림 7-26 TRACE32 Debugging GUI 화면 [출처 - 참고 문헌 12]

TRACE32로 타겟 장치를 디버깅하기 위해서는 다소 복잡한 절차(CMM 스크립트로 작성)가 필요한데, 이를 그림으로 표현하면 다음과 같다.

- 1) 디버거 리셋(Reset)
- 2) Chip 관련 디버거 설정
- 3) 타겟 설정(필요한 경우에 한함)
- 4) 커널을 다운로드하고 커널 시작 파라미터 조정(필요한 경우에 한함)
- 5) 파일 시스템을 다운로드(필요한 경우에 한함)
- 6) 커널 심볼을 로딩하기
- 7) 디버거 MMU 전환(translation) 설정
- 8) 리눅스 인식(awareness) 관련 설정
- 9) 심볼 자동 로더 설정

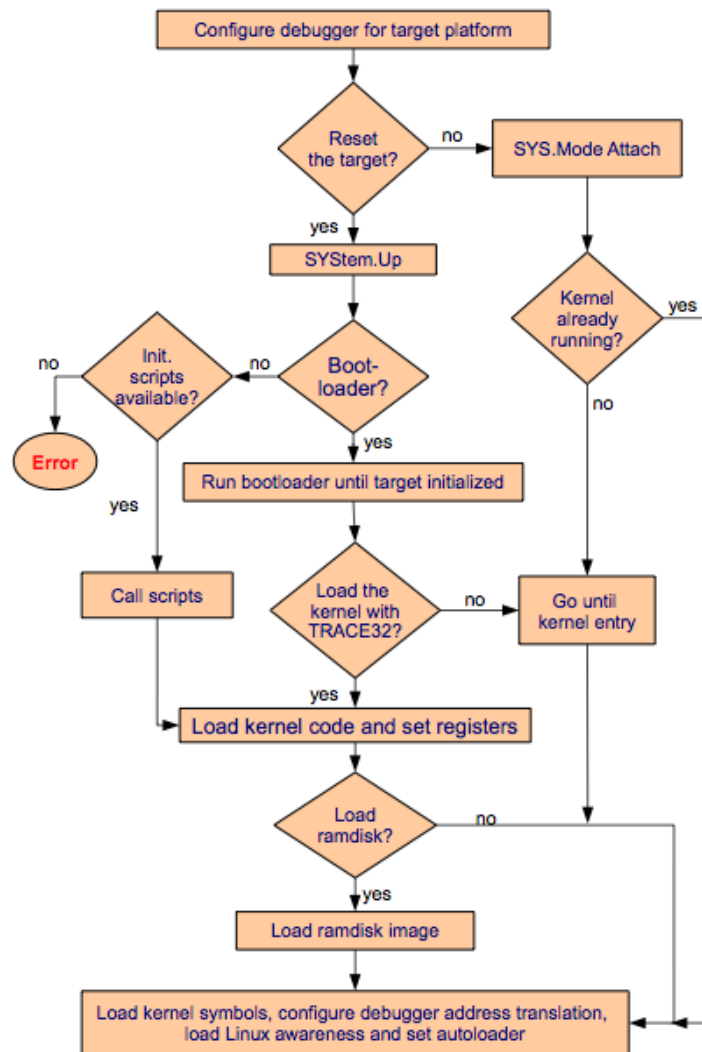


그림 7-27 리눅스 디버깅 환경 셋업 단계 [출처 - 참고 문헌 12]

그림 7-28은 PandaBoard(Android가 올라감)를 디버깅하기 위한 셋업 스크립트의 한 예를 보여주고 있다.


```

RESet

; setup of ICD
PRINT "initializing..."
SYStem.CPU OMAP4430
SYStem.JtagClock CTCK 30MHz
SYStem.Option DACR ON           ; give Debugger global write permissions
TrOnchip.Set DABORT OFF        ; used by Linux for page miss!
TrOnchip.Set PABORT OFF        ; used by Linux for page miss!
TrOnchip.Set UNDEF OFF         ; may be used by Linux for FPU detection
SYStem.Option MMUSPACES ON     ; enable space ids to virtual addresses
SYStem.Up

; Open a serial terminal window
DO ~/demo/etc/terminal/term.cmm COM1 115200.

SETUP.IMASKASM ON              ; lock interrupts while single stepping

; Let the boot monitor setup the board
Go
PRINT "target setup..."
WAIT 1.s
Break

; Load the Linux kernel
Data.LOAD.Elf vmlinux 0x80008000-0xc0008000 /GNU /NoSymbol /NoREG

Register.RESet

; Set PC on physical start address of the kernel
Register.Set PC 0x80008000

; Set machine type in R1; see arch/arm/tools/mach-types
Register.Set R1 2791. ; omap4_panda

DO atag_list.cmm ; call script to set R2 and init. the atag structure

; Loading RAM disk
Data.LOAD.Binary ramdisk.image.gz 0x81600000 /NoClear /NoSymbol

; Load the Linux kernel symbols into the debugger
Data.LOAD.Elf vmlinux /GNU /NoCODE /STRIPPART 3.

; Open a Code Window -- we like to see something
WINPOS 0. 0. 75. 20.
Data.List
SCREEN

PRINT "initializing debugger MMU..."
MMU.FORMAT LINUX swapper_pg_dir 0xc0000000--0xdfffffff 0x80000000
TRANSLATION.COMMON 0xbf000000--0xffffffff
TRANSLATION.TableWalk ON
TRANSLation.ON

; Initialize Linux Awareness
PRINT "initializing multi task support..."
TASK.CONFIG ~/demo/arm/kernel/linux/linux ; loads Linux awareness
MENU.ReProgram ~/demo/arm/kernel/linux/linux ; loads Linux menu

sYmbol.AutoLoad.CHECKLINUX "do ~/demo/arm/kernel/linux/autoload.cmm "

Go
WAIT 5.s
Break

ENDDO

```

그림 7-28 Pandaboard 디버깅 용 셋업 스크립트(CMM 파일) 예 [출처 - 참고 문헌 12]

그림 7-29는 CPU 선택 화면을 보여준다.

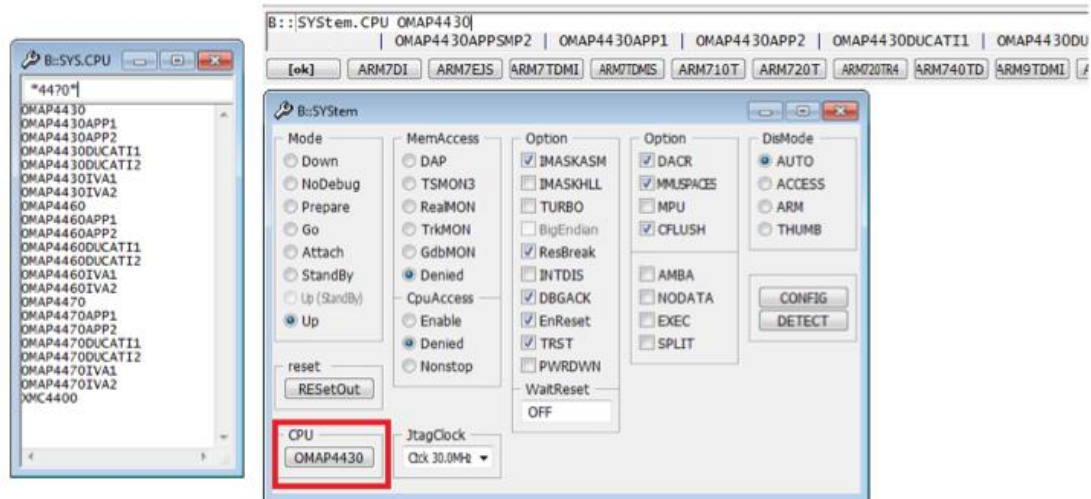


그림 7-29 CPU 선택 화면 [출처 - 참고 문헌 12]

마지막으로 그림 7-30은 커널 부팅 후, 특정 위치에 브레이크 포인트를 지정한 후, 디버깅하는 화면의 한 예를 보여주고 있다.

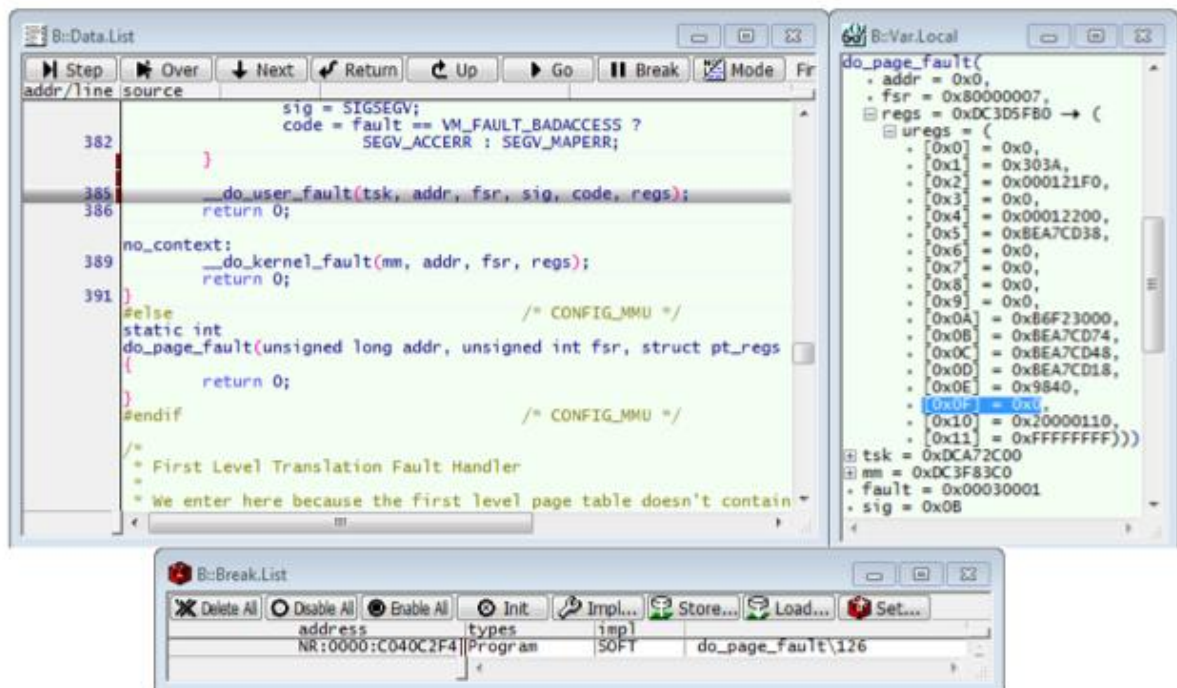


그림 7-30 커널 Segmentation Violation 추적 화면 예 [출처 - 참고 문헌 12]

지금까지 TRACE32의 특징에 관하여 매우 원론적인 부분만을 소개해 보았다. TRACE32는 Run Mode, Stop Mode 등의 디버깅 모드가 존재하며, Ram Dump 이미지를 이용하여 시뮬레이션을 하는 듯 다양한 기능을 제공한다. TRACE32 관련 보다 자세한 사항은 참고 문서 [12-14]를 참고하기 바란다.

5. Kernel Trace 기법

본 절에서는 커널 코드의 중간 중간에 특정 debugging code를 삽입한 후, 동작 시키는 과정을 통해 문제의 원인을 찾아가는 kernel trace 방법들에 관하여 소개하고자 한다. 앞서 2-4 절에서 소개한 기법에 비하면 상대적으로 효용성이 떨어지는 방법으로 볼 수도 있겠으나, 경우에 따라서는 매우 효과적인 방법으로 이용될 수도 있음을 밝힌다.

5.1 kprobe/jprobe/kretprobe

Kprobe, jprobe 및 kretprobe 관련 주요 특징을 정리하면 아래와 같다.

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

물론, kernel code에 printk 문을 집어 넣어 직접 debugging하는 방법도 있겠으나, Kernel code를 수정하지 않으면서도, run-time에 특정 함수를 debugging할 수 있는 효과적인 방법이다. debugging하고자 하는 code가 복잡하고 난해하여, 함수의 흐름을 이해하기 어려운 경우에도 매우 유용하다. kprobe/jprobe 관련 자세한 사항은 doc/Documentation/kprobes.txt 파일 참조하기 바라며, kernel/samples/kprobes 아래에 관련 sample code가 마련되어 있다.

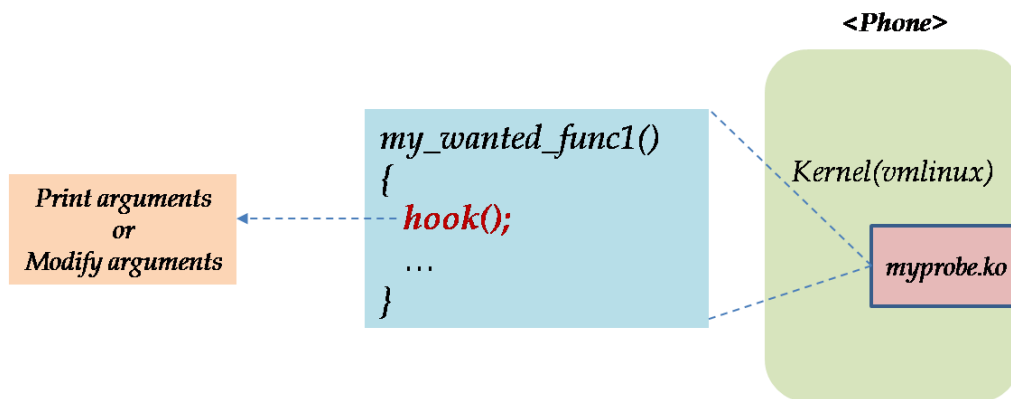


그림 7-31 Kprobe의 개념

<Kprobe 예제>

아래 코드를 build하여 생성한 모듈을 단말에 insmod하게 되면, do_fork() 함수가 호출되기 직전 및 직후에, 원하는 결과를 얻을 수 있게 된다. 참고로 fault_handler는 kprobe가 실행되는 도중 exception이 발생한 경우에 호출된다.

```
#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";    /* debugging을 원하는 함수명으로 교체 */
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%pWn", 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, quittingWn");
        return -1;
    }

    printk(KERN_INFO "Hello: module loaded at 0x%pWn", my_init);
    return 0;
}

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

module_init(my_init);
module_exit(my_exit);

```

<Jprobe 예제 >

아래 코드를 build하여 생성한 모듈을 단말에 insmod하게 되면, timer_inst() 함수가 호출될 때마다, 원하는 결과를 얻을 수 있게 된다. 예를 들어, kernel의 특정 함수에서 죽는 문제가 있는데, 누

가 문제를 발생시키는지 모를 경우, probe 함수 내에 dump_stack() 함수를 추가하면 debugging에 많은 도움이 될 것이다.

```
#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();      /* mod_timer 함수가 불리울 때마다, stack trace dump를 시도한다. */
    jprobe_return();
}

static struct jprobe jp = {
    .kp.addr = (kprobe_opcode_t *) mod_timer,    /* debugging을 원하는 함수명으로 교체 */
    .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);
```

5.2 KGTP

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 방법이다. kgtp 관련 자세한 사항은 <http://code.google.com/p/kgtp> 참조하기 바란다.

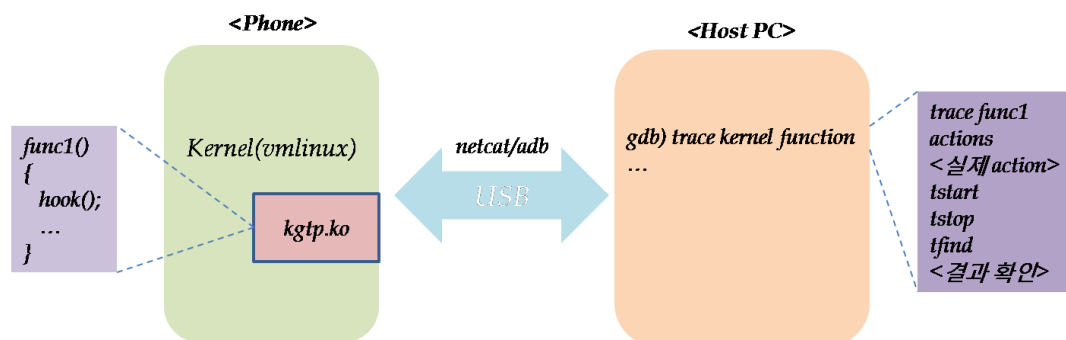


그림 7-32 KGTP 개념도

1) KGTP를 사용하기 위해 kernel config 조정 후, 빌드 하기

```
[*] Enable loadable module support --->
```

```
General setup --->
```

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

```
    [*] Kprobes
```

```
Kernel hacking --->
```

```
    [*] Debug Filesystem
```

```
    [*] Compile the kernel with debug info
```

<PC>

```
# cd android
```

```
# source build/envsetup.sh
```

```
# choosecombo
```

```
    [Device] 1
```

```
    [release] 1
```

```
    [13. YOUR_DEVICE] 14
```

```
    [eng] 3
```

```
# cd kernel
```

```
# make ARCH=arm CROSS_COMPILE=arm-eabi- YOUR_DEVICE_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
```

2) kgtp.ko 빌드 후, 설치하기

2.1) 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
```

2.2) kgtp.ko build하기

<PC>

```
# make AUTO=0
```

- gtp.ko가 정상적으로 생성됨.
- AUTO=0을 안줄 경우, insmod시 에러 발생함(Exec format error).

2.3) 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
```

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

```
(nc -l -p 1234 < /sys/kernel/debug/gtp > /sys/kernel/debug/gtp      ← netcat0| old version일 경우)
```

- 종료하지 않고, 대기 상태로 있게 됨.

3) gdb-release download 받기

- kgtp를 사용하기 위해서는 gdb-release version을 사용하여야 함.

Ubuntu 10.04 이후 버전을 사용하는 경우라면, 아래 명령을 수행해 주어야 한다.

<PC>

```
# sudo add-apt-repository ppa:teawater/gdb-$(lsb_release -rs)
```

```
# sudo apt-get update
```

```
# sudo apt-get install gdb-release
```

4) gdb-release 실행하기

<PC>

```
# adb forward tcp:1234 tcp:1234
```

- netcat이 열어둔 1234번 port의 내용을 localhost 1234번으로 forwarding

```
# cd android/out/target/product/qsd8250_surf/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.

<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
    /home/pz1944/pz1944_EF14L_GB_5040_UI_3/A-QSD8X50GB_5040/LINUX/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 /home/pz1944/pz1944_EF14L_GB_5040_UI_3/A-
QSD8X50GB_5040/LINUX/android/kernel/fs/readdir.c:93
#2  0x000006ee in slip_exit ()
at /home/pz1944/pz1944_EF14L_GB_5040_UI_3/A-
QSD8X50GB_5040/LINUX/android/kernel/drivers/net/slip.c:1365
#3  0x000006f4 in slip_exit ()
at /home/pz1944/pz1944_EF14L_GB_5040_UI_3/A-
QSD8X50GB_5040/LINUX/android/kernel/drivers/net/slip.c:1369
#4  0x000006f4 in slip_exit ()
at /home/pz1944/pz1944_EF14L_GB_5040_UI_3/A-
QSD8X50GB_5040/LINUX/android/kernel/drivers/net/slip.c:1369
Backtrace stopped: previous frame identical to this frame (corrupt stack?)

```

이 밖에도 커널 Trace를 이용한 기법으로는 LTTng 등 다양한 방법이 존재하나 본 서의 범위를 넘어서는 듯하여 추가로 정리하지는 못하였음을 밝힌다.

6. 사용자 영역 디버깅 기법

이번 장에서는 커널 디버깅 기법과는 무관한 사용자 영역의 디버깅 기법에 관하여 정리해 보고자 한다.

6.1 기본 디버깅 도구 – adb

adb(android debug bridge)는 안드로이드 디버깅 시 없어서는 안될 중요한 도구이다. adb를 이용한 주요 디버깅 명령을 정리해 보면 다음과 같다.

```

# 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

```
# 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 : thread별 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) ...

6.2 Native Code(C/C++) 디버깅

1) strace

Strace는 debug하려는 program(or process)가 호출하는 각종 system call을 추적할 수 있는 도구이다. Strace로 debugging하고자 하는 위치를 알아내고, gdb로 breakpoint를 지정한 후, breakpoint까지 trace하는 방법을 활용하면 보다 효과적으로 debugging이 가능할 수도 있을 것이다.

```
# 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.

2) gdb & gdbserver

Gdb는 C/C++ code를 디버깅하는데 매우 유용한 도구로, 타겟 보드 내의 gdbserver 및 adb와 상호 연계하여 동작하는 구조로 되어 있다.

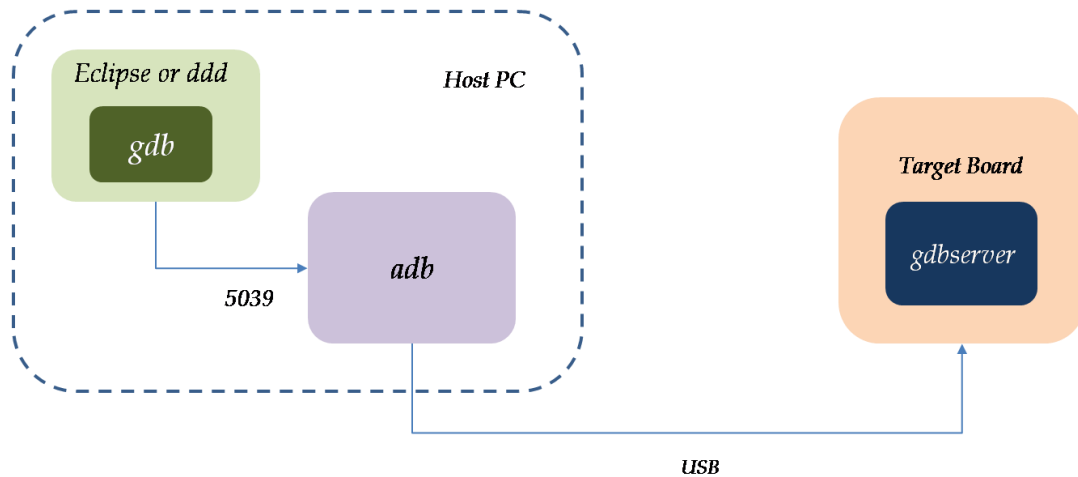


그림 7-33 gdb & gdbserver

<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/symbols/system/bin/mediaserver
```

```
(gdb) set solib-absolute-prefix ~/YOUR_PATH/symbols
```

```
(gdb) set solib-search-path ~/YOUR_PATH/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++])

```
(gdb) bt
```

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

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

3) debugged & tombstone

Debugged와 Tombstone은 Native 코드 디버깅을 위해 Google에서 추가한 기능으로, 아래와 같은 원리로 동작한다.

a) [사전 지식] Google에서는 *bionic linker* 내에 *signal handler*(SIGSEGV, SIGPIPE, SIGABRT 등이 죽을 때 처리되는 코드)를 추가하여 두었으며, 사용자 *application*은 반드시 *bionic linker*를 사용하도록 되어 있다.

b) User process가 사용 중 이상 동작하여 죽으려 한다.

c) Kernel이 이상 동작에 대해 *signal*을 발생시킨다.

■ 발생 가능한 *signal* 종류: SIGILL, SIGABRT, SIGBUS, SIGFPE, SIGSEGV, SIGSTKFLT, SIGPIPE

d) Bionic linker 내의 *signal handler*가 동작한다.

e) *signal handler*는 *debugged daemon process*와 *domain socket* 연결을 시도한 후, *pid(or tid)* 정보를 *debugged*로 전달한다.

f) *debugged*는 *ptrace*를 활용하여 죽은 *process*의 *stack* 정보를 */data/tombstones* 디렉토리 아래에 저장한다.

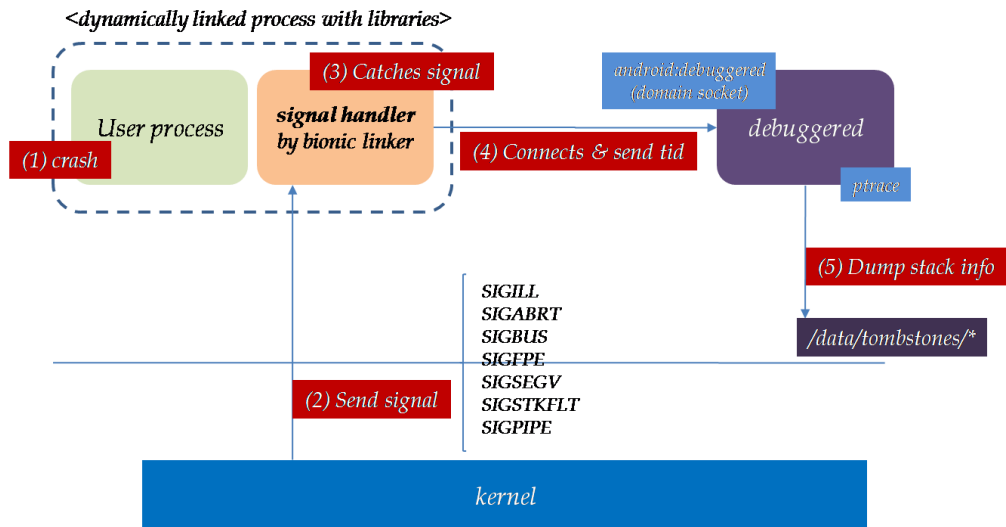


그림 7-34 Debugged와 Tombstone

4) addr2line & objdump

<logcat 내용 중, 문제가 되는 부분>

```

10-24 13:27:46.509 I/DEBUG (16058): Build fingerprint: --:user/release-keys'
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 39b8ddb0 lr 6fd191e9 pc 6ca3d444 qpsr 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
...
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

<function name을 추출한 결과 - 위의 빨간색 표시 부분을 아래에서부터 위로 trace한 결과>

```
# cd android/out/target/product/qsd8250_surf/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
```

```

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

```

arm-eabi-objdump를 사용하면 해당 library에 대한 disassemble 결과를 얻을 수 있으므로 arm-eabi-addr2line 보다 자세한 debugging이 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/YOUR_DEVICE/symbols/system/lib
# arm-eabi-objdump -S ./libdbus.so > objdump.txt ← -S는 역어셈블 옵션임
# vi objdump.txt
    ■ 13fca로 출력된 내용 검색

```

```

pz1944@mars: ~/pz1944_EF14L_GB_5040_UI_3/A-QSD8X50GB_5040/LINUX/android/out/target/product/qsd8250_surf,
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 탭(B) 도움말(H)
pz1944@mars: ~/pz1944_EF14L_GB_5040_UI_3/A-QSD8X50... ✖ pz1944@mars: ~/pz1944_EF14L_GB_5040_UI_3/A-QSD8X50... ✖

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;
1
    _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;
1
    _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
13fda: 4630      mov r0, r6
    CONNECTION_UNLOCK (connection);
    return;
1
    CONNECTION_UNLOCK (connection);
    return;
1

```

그림 7-35 arm-eabi-objdump -S ./libdbus.so 결과

Dump하려는 library가 C++로 작성되었을 경우에는 -C 옵션도 함께 사용한다.

```
# arm-eabi-objdump -S -C ./libdbus.so
```

마지막으로, arm-eabi-objdump는 android/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin 아래에 있다.

<여기서 잠깐 ! - 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.

- 1) 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 필요하다.
- 2) SIGSEGV/SEG_MAPERR는 Native code쪽에서 발생한 것이다.
- 3) 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.

6.3 Java Code 디버깅

이번 절에서는 Java code를 디버깅하는데 사용되는 DDMS와 jdb에 관하여 살펴보고자 한다.

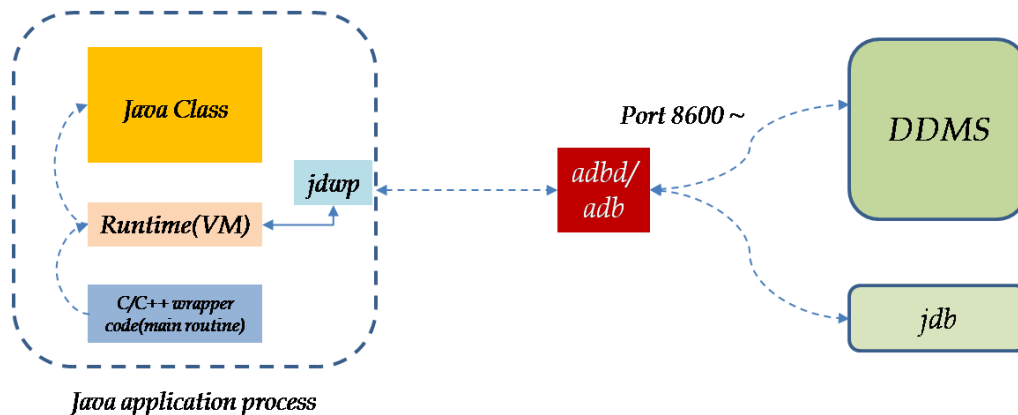


그림 7-36 DDMS와 jdb

1) DDMS

DDMS를 이용한 디버깅 절차를 정리해 보면 다음과 같다.

a) # export ANDROID_BUILD_TOP=~ /YOUR_PATH/android

- 자신의 환경에 맞게 적절히 지정

b) # ./cts/development/ide/eclipse/genclasspath.sh 실행

c) 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.

d) 새로 생성된 프로젝트의 코드 중에 디버깅 할 코드에 Breakpoint 설정

- system_server를 debugging하고자 한다면, 관련 코드 중 하나를 선택하여 breakpoint를 지정해야 함.

e) Package Explorer에서 새로 생성된 프로젝트에 대해 마우스 오른쪽 클릭 후 Debug As -> Debug configurations 클릭

f) Eclipse 버전마다 약간의 차이는 있을 수 있으나, Remote Java Application을 선택한 후, Host는 localhost, Port는 8600 또는 8700으로 입력 후 Debug 버튼 클릭

g) 에러 메시지가 나오나, Proceed를 눌러 진행

- VM에 연결할 수 없다는 popup이 뜰 경우, 다른 창에서 `sudo adb shell` 하여 단말의 `adbd`를 새로 띄워줌.

h) 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.

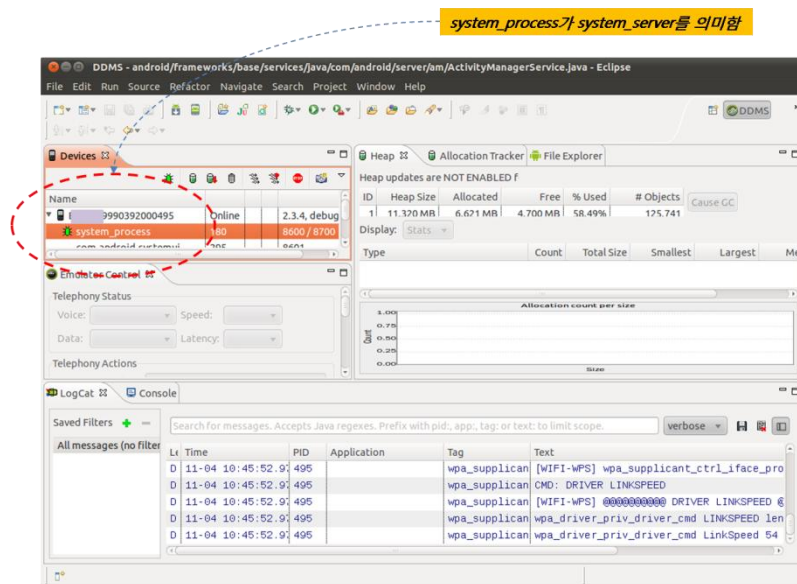


그림 7-37 DDMS를 이용한 debugging

2) jdb(Java Debugger)

DDMS가 훌륭한 도구이기는 하나, 절차가 복잡하고 느린 문제(메모리를 많이 필요로 함)가 있다. 이럴 때는 jdb를 사용하는 편이 낫을 수도 있다. 시스템이 이상한 상태(?)에서 의심 가는 process를 suspend 시킨 후, stack을 dump해서 보면 문제의 원인을 확인할 수 있는 좋은 출발점이 될 수도 있을 것이다. 아래에 jdb를 사용하는 방법을 정리해 보았다.

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

- `system_server`의 pid가 171임을 확인

(*) system_server 관련 threads

system	171	97	262516	52408	fffffff	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	6fd0bfff	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	6fd0bfff	S	DisplayEventThr
system	182	171	262516	52408	8009fa98	6fd0c748	S	SurfaceFlinger
system	186	171	262516	52408	800f21b8	6fd0c5ac	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	PackagesManager
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	6fd0c5ac	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

그림 7-38 adb shell ps -t 예

\$ adb forward tcp:8000 jdwp:171

- pid 171 process 관련 debug 정보를 tcp port 8000으로 forwarding해줌.
- 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 패키지의 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

...

```
> 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:1019)
<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)
```

그림 7-39 jdb stack dump 예

References

<Generic References>

- [1] *Essential Linux Device Drivers*, Sreekrishnan Venkateswaran, Prentice Hall.
- [2] *Embedded Linux Primer, Second Edition*, Christopher Hallinan, Prentice Hall.
- [3] *Debug Hacks*, 진명조 역, 와우북스
- [4] *Android Debug Guide*, Chunghan Yi, www.kandroid.org
- [5] *Linux Kernel and Driver Development Training*, Gregory Clement, Michael Opdenacker, Maxime Ripard, Sebastien Jan, Thomas Petazzoni, Free Electrons

<References for KGDB & USB debugging>

- [6] <http://bootloader.wikidot.com/android.kgdb>
- [7] 안드로이드 스마트폰을 위한 USB 기반 통합 디버깅 방법, 경주현 외, 정보과학회논문지, 시스템 및 이론 제 39 권 제 2 호(2012.4)
- [8] USB를 이용한 임베디드 리눅스 개발환경 개선, 경주현, ㈜세븐코아

<References for JTAG & OpenOCD>

- [9] http://www.omappedia.org/wiki/PandaBoard_JTAG_Debugging
- [10] *Using OpenOCD JTAG in Android Kernel Debugging*, Mike Anderson, The PTR Group
- [11] *Hardware Debugging using GDB, OpenOCD and JTAG*, Dheeraj Chidambaranathan(ASU ID 1205016081), Imtiyaz Hussain(ASU ID 1204032877), 12/11/2012

<References for TRACE32>

- [12] *Training Linux Debugging*, ©1989-2013 Lauterbach GmbH.
- [13] *RTOS Debugger for Linux – Stop Mode*, ©1989-2013 Lauterbach GmbH.
- [14] *RTOS Debugger for Linux – Run Mode*, ©1989-2013 Lauterbach GmbH.
- [15] *TRACE32 중급과정*, MDS Technology Technical Support Team

<References for KProbe & KGTP>

[16] *Linux Kernel GDB tracepoint module (KGTP)*, <https://code.google.com/p/kgtp/>

<References for DDMS>

[17] <http://blog.naver.com/jang2818/20078863663>