제27회 한국정보처리학회 춘계학술발표대회 논문집 제14권 제1호 (2007. 5)

ELF에 기반한 Remote Debugger의 call stack watch view 구현에 관한 연구

주상민° 김홍규 문승진
수원대학교 IT대학 컴퓨터학과
{lolcat, exxxfx, sjmoon}@suwon.ac.kr

A Study of Remote Debugger for call stack watch view based on the ELF

Sang-min Joo° Hong-Kyu Kim Seung-jun Moon
Dept. of Computer Science, The University of Suwon

요 약

멀티미디어 디지털 기기의 발전은 Mobile Phone의 사용량 증가로 대두되고 있으며 그에 따른 문제를 해결하는 일도 시급한 상황이다. Mobile Phone 사용 중 작동이 멈춘다거나 하는 문제는 그 사용자 수에 비례하여 발생 빈도나 위험성이 높아질 수밖에 없다. 이런 문제가 생기지 않도록 사전에 철저한 테스트를 해보고 시행착오를 겪어봐야 완벽한 제품으로 시중에 내놓을 수 있게 되는 것이다. 기존의 하드웨어와 연결해서 사용 하던 디버거와는 다르게 Remote Debugger는 이러한 문제점을 툴을 통해서 소프트웨어적으로 찾아내어 고치는 역할을 하게 될 것이다.

본 논문에서는 이러한 Remote Debugger를 구현하기 위한 부분 중 call stack watch view 에 대해 논의 하려고 한다.

1. 서 론

휴대용 멀티미디어 디지털 기기의 사용량의 증가로 ARM 플랫폼을 기반으로 한 기기들의 수 가 늘어나고 추세에 있으며 현재 가장 많이 사용되고 있는 상품은 단연 mobile phone이라고 할 수 있다. 이러한 mobile phone은 국민 1인당 1개씩 가지고 있을 정도로 많이 쓰이고 있으며 많은 종류의 모바일 폰이 단기간동안 생산되고 있는 실정이다. 적은 개발 시간에 여러 종류의 응용프로그램을 한정된 공간에 적재하여 운영하는 것은 양산 전 테스트 공정에서 찾을 수 없는 일반적인 에러나 운영체제가 다운되는 치명적인 에러가 발생하여 큰 문제를 야기 시킬 수 있다. 따라서 보다 정확하고 빠른 에러 검출을 위한 툴이 필요한 실정이다.

이에 본 논문에서는 보다 정확하고 빠른 에러 검출을 위하여 remote debugger를 구현하기 위한 부분 중 call stack watch view에 관한 연구에 대해 논의하려고 한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서 ARM과 ELF file에 관하여 간략하게 설명하였고, 3장에서 remote debugger의 call stack watch view에 관하여 논의하고, 4장에서 본 논문의 결론과 앞으로 나아가야 할 방향을 짚어 보려고 한다.

2. ARM & ELF FILE

2.1 ARM프로세서

ARM(Advanced Risc Machine)이 휴대용 기기에서 많이 언급되는 이유 중 하나는 연산 속도 대비 소비 전력이 적다는 것이다.

휴대용 기기의 문제점 중 하나로 꼽을 수 있는 전력

보급 문제에서 이런 장점은 더 크게 작용 하게 된다.

두 번째로 독자 생산이 아닌 라이센스 방식의 제작으로 타 반도체 업체에서 라이센스를 받아 주변 회로를 구성하여 독자적인 ARM 프로세서를 만들어 생산 및 판매 하고 있다. 그 예로 Intel, Samsung, Motorola 등을 꼽을 수 있겠다.

세 번째로 Soc Designs IP 제공을 한다. 즉, 하나의 칩 안에 하나의 시스템이 모두 들어 있음으로서 VGA, sound card 를 포함한 메인보드를 하나의 칩이라고 생각 할 수 있다.

우리가 사용하고 있는 호스트인 PC는 대부분이 인텔 X86 계열의 CPU가 탑재되어 있지만 타겟의 CPU 계열은 호스트와 다른 것이 대부분이다. 따라서 호스트에서의 컴파일 작업은 이런 CPU를 지원하는 별도의 컴파일러에 의존해야 한다. 이러한 컴파일러를 크로스 컴파일러라고 하는데 C파일을 크로스 컴파일러로 컴파일 했을 경우 파일 정보를 확인해보면 32bit 혹은 64bit ELF 포맷, ARM용으로 컴파일 된다.

2.2 ELF FILE

ELF(Executable and Linking Format)파일은 시스템 바이너리 파일로서 Unix System Lboratory에서 개발되고 발전되어 왔다. SVR4와 Solaris 2.X version의 운영체제에서는 기본적인 실행파일의 포맷으로 사용되고 있다. 실행파일의 포맷으로는 "*a.out*"과 "*COFF*" 포맷이 있지만, ELF 포맷이 보다 강력하며, 유연성을 가지고 있다. 적절한 툴과 같이 사용될 때 실행되는 과정을 제어 할 수 있다. 프로그래머들은 바이너리 파일의 이러한 인터페이스만을 중심으로 프로그램을 할 수 있는 방법을 제공 받을 수 있으며, 더불어 새로이 코드를 재 컴파일해서 기록할 필요가 없게 된다.

<표 1> 자료형의 표현

```
/* 32-bit ELF base types. */
typedef __u32        Elf32_Addr;
typedef __u16        Elf32_Half;
typedef __u32        Elf32_Off;
typedef __s32        Elf32_Sword;
typedef __u32        Elf32_Word;
/* 64-bit ELF base types. */
typedef __u64        Elf64_Addr;
typedef __u16        Elf64_Half;
typedef __s16        Elf64_SHalf;
typedef __u64        Elf64_Off;
typedef __s64        Elf64_Sword;
typedef __u64        Elf64_Word;
```

<표 2> 32bit 데이터 타입

Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

위 두 표에서 ELF 포맷이 다양한 프로세서에서 사용할 수 있으며 각각의 기본 데이터 타입의 길이가 달라질 가능성이 있기 때문에 32Bit와 64Bit 구조를 가진 CPU를 지원한다.

(a) linking view	(b) execution view
ELF Header	ELF Header
Program header table optional	Program header table
Section 1	Section 1
...	...
Section n	Section n
...	...
...	...
Section header table	Section header table

(그림 1) ELF 파일 구조

- ELF Header : 파일 처음부분에 존재하며 전체 파일의 구성을 알려준다.
- Program Header Table : 시스템이 어떻게 프로그램을 실행 시키는지 알려주며 옵션이라고 생각하면 된다.
- Section : 링킹을 할 때 필요한 명령어, 데이터, 테이블, 재배치 정보 등을 가지고 있다.
- Section Header Table : 목적파일에 들어있는 Section 들이 어떻게 구성되어 있는지에 대한 정보를 가지고 있다.

3. Remote Debugger call stack watch view

3.1 Remote Debugger의 기본 형태

remote debugger로 ELF 파일을 읽어오게 되면 그림 2와 같이 ELF 파일의 내용이 헥사 값으로 출력되고 그에 따른 주소가 나오게 된다.

<표 3> rex.h 파일의 형식

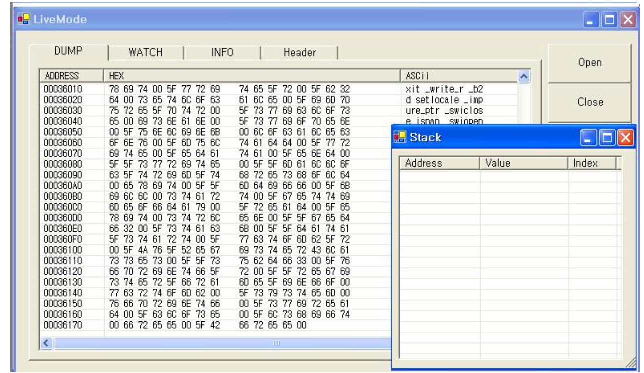
```
typedef struct rex_tcb_struct {
#ifdef FEATURE_ZREX
#error code not present
#endif
void *sp;              /* Stack pointer  */
void *stack_limit; /* Stack Limit    */
unsigned long slices;      /* Task slices     */
rex_sigs_type sigs;        /* Current signals */
rex_sigs_type wait;        /* Wait signals    */
rex_priority_type pri;     /* Task's priority */
#if defined FEATURE_REX_PROFILE
unsigned long time_samples; /*Profiling information */
unsigned long  max_intlock_time; /* Profiling Info */
#endif
#if defined TIMETEST
word     leds;           /* TIMETEST val     */
#endif
#if defined FEATURE_SOFTWARE_PROFILE

 /* 32 bits counter, ~30 usec/tick, ~35 hours */

dword numticks;
#endif
#ifdef FEATURE_REX_APC
long num_apcs;      /* Number of APCs queued */
#endif
#ifndef FEATURE_ZREX

rex_tcb_link_type link;           /* for TCB list */
#endif
rex_tcb_link_type cs_link;
/* non-NULL when waiting for a * critical section */

rex_crit_sect_type *cs_stack[REX_CRIT_SECT_MAX];
/* crit sects TCB wants/has */
rex_crit_sect_type    **cs_sp;     /* crit sects stack
pointer */
boolean     suspended;          /* whether task starts
suspended */
struct   rex_tcb_struct      *pri_rep_ptr;   /*    priority
representative */
char   task_name[REX_TASK_NAME_LEN + 1];
#if defined FEATURE_REX_EXTENDED_CONTEXT
void                      *ext;
#endif
```

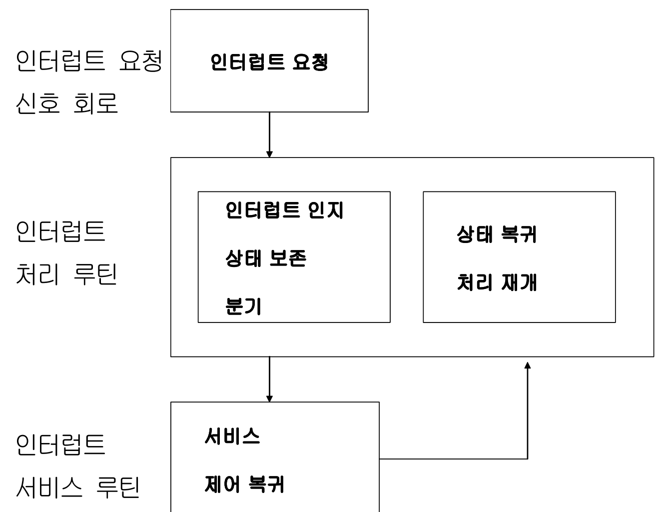


(그림 2) remote debugger

본 논문에서 다루고자 하는 부분은 ELF 파일에서 실질적으로 필요한 스택부분의 주소를 찾아내어 그 값을 고칠 수 있도록 만드는 것이다. 이러한 작업을 통해서 mobile phone에서 문제가 되는 부분을 쉽게 찾아내어 고칠 수 있게 된다.

3.2 Call Stack Watch View의 구현

본 논문에서 제안하고자 하는 call stack watch view의 원리는 먼저 프로그램에서 어떤 명령을 호출하게 되면 호출에 관한 정보가 생성된다. 따라서 생성된 정보를 스택 프레임이라 불리는 데이터 블록에 저장하고 스택 프레임은 *call stack* 메모리 영역에 대입된다. 이에 mobile phone이 작동하다가 어떠한 오류가 발생하여 작동이 멈추게 되면 그림 3과 같이 인터럽트를 인지한 후 해당 스택 프레임 중 하나를 디버거에 의해 선택되어 명령어들이 선택된 프레임을 가리키게 하여 오류 발생 부분을 수정 할 수 있게 한다.

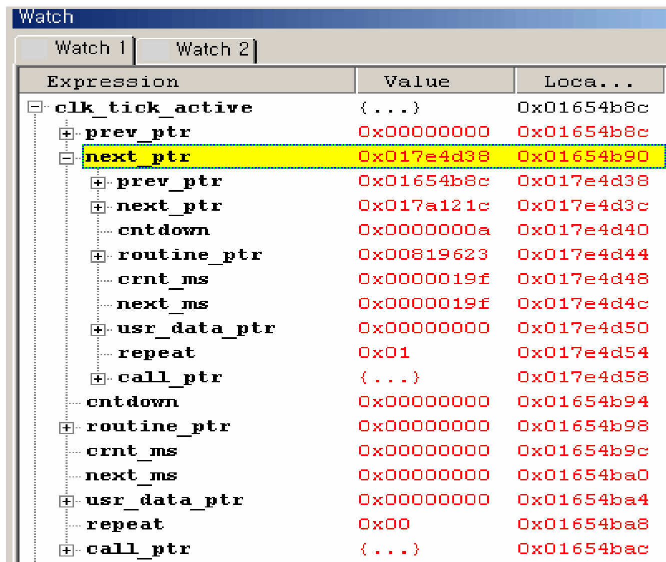


(그림 3) 인터럽트 동작 원리

그림 3의 인터럽트 동작원리를 이용하여 우선적으로 ELF 파일의 스택의 주소를 가져온 후 주소 값에 따라 값을 출력하여 준다. 필요한 부분의 최하위 주소 값과 스택 limit 주소 값의 합으로 실질적으로 필요한 부분의 값을 살펴 볼 수 있게 된다. 그 필요한 부분을 찾는 작업은 mobile phone의 각 작업 태스크를 부분적으로 나누어 표 3 과 같이 수월하게 찾을 수 있도록 한다.

각 명령어에 해당하는 함수가 호출될 때마다 새로운 프레임을 만들고 함수가 반환될 때 함수 호출을 위한 프레임을 제거 시킨다. 이러한 스택 프레임은 이와 같은 과정을 반복 하다가 문제 발생 시(인터럽트 발생) 현재 작업 중이던 스택 주소를 반환하여 어느 부분에서 문제가 발생 하였는지 알려주게 한다. 여기서 인터럽트 발생 시에 처리되는 문제는 모바일 폰의 제품출시에 상당히 중요하게 작용 한다.

본 논문에서 제안하고자 하는 기능이 체계화되어 있지 않다면 개발된 프로그램의 소스코드에서 문제 발생 부분을 찾는데 상당 시간이 걸릴 뿐 아니라 오류 부분을 찾지 못한 채로 시중에 출시되어 장래에 큰 문제를 야기 시킬 수 있게 된다.



(그림 4) 각 task의 주소 값과 value

따라서 그림 4와 같이 각 함수의 포인터에 대한 위치와 값이 출력되고 선택하였을 때 그 값을 고칠 수 있는 *call stack watch view*가 완성 될 것이다.

4. 결론 및 연구과제

본 논문에서는 remote debugger의 *call stack watch view*구현 방법에 관하여 제안하였다. 이에 ARM 프로세서와 ELF 파일 구조에 대해 많은 언급을 하였고, 실제로도 ELF 파일 구조에 대한 이해가 없다면 디버거 구현에 많은 어려움을 겪을 것이다.

따라서 본 논문에서 제안하는 방법은 ELF 파일의 스택 주소 값을 가져온 후 인터럽트 발생 시 스택 프레임에 저장된 주소를 찾아 오류를 찾아 수정하며 시스템 리소스의 최적 사용과 함께 제품의 안정성 향상을 위해 인터럽트 호출 방식을 사용하였다. 이와 같은 ELF를 기반으로 한 디버거는 ARM 기반의 임베디드 시스템의 오류 수정에 많은 도움이 될 것이다.

기존의 하드웨어적인 접근에서 탈피하여 소프트웨어적으로 접근하여 유연성이 향상되게 된다.

현재 mobile phone에 디버거를 직접 연결하여 수정 하는 방법을 연구 중에 있지만 차후에는 외부에서 원격으로 접속하여 디버거를 통해 오류를 수정할 수 있는 원격 시스템을 연구 개발 해 나가도록 할 것이다.

참고문헌

[1] “Minimalist GNU for Windows“ (http://www.mingw.org)

[2] "GDB를 위한 Debugging" (http://www.nortul.com/linux/GDB/GDB.html)

[3] "Korea Embedded System for Linux", (http://www.kesl.co.kr)

[4] "Korea Embedded Linux Project", (http://www.kelp.or.kr)

[5] “월간 임베디드 월드” (http://www.embeddedworld.co.kr)

[6] "KLDP ", (http://www.kldp.org)

[7] www.arm.com, AXD and armsd Debuggers Guide(ARM DUI 0066) & ARM Developer Suite Debug Target Guide (ARM DUI 0058D)