리눅스 환경에서의 다중 프로세스 응용에 대한 원격 디버깅 기법

(A Remote Debugging Scheme for Multi-process Applications in Linux Environments)

심 현 철 † 강 용 혁 † 엄 영 익 ††
(Hyun Chul Sim) (Yong Hyeog Kang) (Young Ik Eom)

요 약 리눅스 기반의 임베디드 시스템에서 동작하는 응용 프로그램의 디버깅은 타겟 시스템의 제한된 자원으로 인하여 대부분 원격으로 이루어진다. 리눅스 기반의 시스템에서 동작하는 gdb는 원격에서 또는 로컬에서 디버깅 중인 프로세스가 fork 시스템 콜 호출 시 새로이 생성된 프로세스를 디버깅할 수 있도록 지원하지 않는다. 이에 따라, 리눅스 환경에서 동작하는 gdb와 gdbserver를 사용해서 단일 프로세스 구조를 갖는 응용 프로그램을 원격 또는 로컬 디버깅할 수는 있으나 다중 프로세스 구조의 응용 프로그램을 원격으로 디버깅할 수는 없다. 또한 로컬에서 gdb를 사용하여 다중 프로세스 구조의 응용 프로그램을 디버깅할 수는 있지만 이를 위해서는 개발자의 부가적인 코딩이 필요하다. 본 논문에서는 리눅스 커널의 변경 없이 라이브러리 래핑(wrapping) 방법을 이용하여 원격 시스템에서 동작하는 다중 프로세스 구조의 응용 프로그램을 gdb와 gdbserver를 사용하여 디버깅하는 방법을 제시한다.

키워드 : 디버깅, 다중 프로세스 디버깅, 임베디드 리눅스

Abstract Debugging for application programs running in embedded Linux systems has mostly been done remotely due to the limited resources of the target systems. The gdb, which is one of the most famous debugger in Linux systems, does not support the debugging of the child processes which is created by the fork system call in local and remote environments. Therefore, by using gdb, developers can debug the application programs that have single-process structure in local and remote environments, but they cannot debug the application programs that have multi-process structures by using gdb in remote environments. Also, although developers can debug the application programs that have multi-process structures by using gdb in local environments, it needs additional and unnecessary codings. In this paper, we presents the remote debugging scheme that can be used for debugging multi-process structured applications. The proposed scheme is implemented by using the library wrapping scheme, and also uses the conventional system components such as gdb and gdbserver.

Key words : debugging, Multi-process debugging, Embedded Linux

1. 서 론

최근 리눅스 기반의 임베디드 시스템에서는 원격 디버깅 도구로 gdb를 많이 사용하고 있다. gdb는 소스가 오픈 되어 있으며 다양한 플랫폼에 이식되어 있다. gdb는 호스트 시스템에서 gdb를 실행시키고 타겟 시스템에서 gdbserver를 실행함으로써 상대적으로 호스트 시스템보다 부족한 자원을 갖는 타겟 시스템에서 동작하는 프로세스를 원격으로 디버깅할 수 있도록 지원한다[1][2].

리눅스 환경에서 gdb를 사용하여 로컬 시스템에서 디버깅 중인 프로세스가 생성한 프로세스를 디버깅하기 위해서는 개발자가 직접 sleep 함수 코드를 삽입하고 새로운 gdb를 새로이 생성된 프로세스와 연결해야만 하는 작업이 필요하다[1]. 더구나 이 방법은 로컬 시스템에서 gdb를 이용하는 경우에만 가능하며 원격 디버깅 시 이 방법을 사용해서는 새로 생성된 프로세스를 디버

† 비 회 원 : 성균관대학교 정보통신공학부
jlmaj@ece.skku.ac.kr
yhkang1@ece.skku.ac.kr
†† 종신회원 : 성균관대학교 정보통신공학부 교수
yieom@ece.skku.ac.kr
논문접수 : 2002년 2월 28일
심사완료 : 2002년 8월 5일

깅할 수 없다.

본 논문에서는 gdb와 gdbserver를 사용한 원격 디버깅 환경에서 개발자가 디버깅 중인 프로세스로부터 fork 시스템 콜에 의해 생성된 프로세스를 디버깅하는 방법을 제시한다. 이것은 원격으로 gdbserver에 의해 디버깅 중인 프로세스가 fork 시스템 콜로 새로운 자식 프로세스를 생성할 때 개발자가 gdbserver로 부모-자식간의 관계가 아닌 새로 생성된 프로세스를 디버깅할 수 있도록 fork 시스템 콜이 호출되는 시점을 가로채어 개발자가 gdbserver를 사용하여 프로세스를 디버깅 할 수 있도록 지원하는 것이다. 본 논문은 다음과 같이 구성되어 있다. 2장에서는 gdb와 gdbserver를 사용하여 디버깅 중인 프로세스가 생성한 새로운 프로세스를 원격으로 디버깅하지 못하는 문제점에 대해서 분석한다. 3장에서는 본 논문에서 제안하는 SPY_Library와 SPY_Library 라이브러리를 사용하여 문제점을 해결하는 방법에 관하여 설명한다. 4장에서는 본 논문에서 제안하는 방법을 통하여 원격으로 다중 프로세스를 디버깅하는 실험 결과를 설명한다. 또한 ETNUS사의 TotalView 프로그램과 본 논문에서 제안한 방법과의 성능 비교를 하며 마지막으로 5장에서는 결론 및 문제점들을 지적하고 향후 연구방향에 대해서 기술한다.

2. 관련 연구

리눅스 환경에서 다중 프로세스 디버깅을 지원하는 상용 도구로는 ETNUS사의 Totalview라는 것이 있다[3]. 개발자가 이 프로그램을 사용하여 fork 시스템 콜을 통한 다중 프로세스 구조의 프로그램을 디버깅하기 위해서는 libdbfork.a라는 정적 라이브러리를 링크(link)하여 프로그램 이미지를 생성하므로 프로그램 이미지의 크기가 커지게 된다. 또한 이 프로그램은 원격 디버깅 환경이 아닌 로컬 디버깅 환경에서의 다중 프로세스 디버깅만을 지원하기 때문에 비교적 부족한 자원을 갖는 임베디드 시스템을 위한 원격 디버깅 도구로는 부적절하다.

gdb를 사용하여 프로세스를 디버깅하는 경우 정지 상태인 프로세스의 코드 부분에 있는 특정 명령어의 op-code를 변경하여 정지점(breakpoint)을 설정하고 프로세스의 수행을 재개한다. 프로세스의 수행 중 프로세서로부터 정지점 예외 상황이 발생하면 gdb는 해당 프로세스의 제어권을 얻게 되어 개발자가 프로세스를 디버깅 할 수 있도록 지원한다. 본 장에서는 로컬의 리눅스 환경에서 gdb를 사용하여 다중 프로세스를 디버깅하는 방법과 문제점에 대해서 알아본다. 또한 개발자가 원격지의 리눅스 환경에서 동작하는 gdbserver를 사용하여 fork 시스템 콜을 호출한 프로세스를 디버깅할 때 발생하는 문제점을 분석한다.

2.1 로컬에서 기존의 gdb를 이용한 다중 프로세스 디버깅 방법

그림 1은 로컬 시스템에서 gdb를 사용하여 다중 프로세스를 디버깅하기 위해 개발자가 해야 하는 필요한 작업을 나타낸다. 그림 1에서 #ifdef와 #endif로 되어 있는 사이 부분이 로컬에서 gdb를 사용하여 다중 프로세스를 디버깅하기 위해 개발자가 추가해야 하는 코드에 해당한다.

```
if ((pid = fork()) == 0) {
        /* child */
      #ifdef DEBUG_FORKS// 로컬에서 gdb를 사용하여
        int waiting=1;      // 다중 프로세스 디버깅을 하기
        while(waiting) {}   // 위해 개발자가 추가해야 하는
      #endif                // 코드
        DoChildThing();
      } else {
        /* parent */
        DoParentThing()
}
```

그림 1 로컬에서 gdb를 이용한 다중 프로세스의 디버깅 방법

그림 1에서 새로 생성된 자식 프로세스는 불필요한 무한 루프를 수행하고 있다. 개발자는 새로 생성된 자식 프로세스를 디버깅하기 위해 새로운 gdb를 실행하여 gdb와 새로 생성된 자식 프로세스를 연결한다. 개발자는 gdb를 사용하여 자식 프로세스의 waiting 변수 값을 0으로 설정하여 자식 프로세스가 무한 루프에서 빠져나오도록 하고 디버깅을 진행한다. 이 방법은 프로그램 컴파일 시간에 해당 프로그램을 디버깅할 것인지를 결정해야 한다는 단점이 있고 그로 인하여 해당 프로그램의 소스코드를 계속 갖고 있어야 한다. 또한 gdbserver는 현재 독립적으로 수행중인 프로세스와 연결될 수 없도록 구현되어 있어 이 방법은 gdbserver를 사용하는 원격 디버깅 환경에서 사용될 수 없다[4]. 다음절에서는 gdbserver를 사용하여 원격으로 디버깅 중인 프로세스가 생성한 새로운 프로세스를 디버깅할 수 없는 문제점에 대해서 알아본다.

2.2 원격지 리눅스 환경의 gdbserver가 지원하는 다중 프로세스 디버깅의 제한

인텔의 x86기반의 프로세서는 op-code가 "cc"인 명령어(INT3)를 수행하려 하면 정지점 예외 상황을 발생시킨다[5]. 정지점 예외 상황이 발생하면 리눅스 커널은

커널 모드 스택에 대부분의 레지스터 값을 저장하고 예외 상황 처리 함수를 호출한다. 예외 상황 처리 함수에서는 예외 상황이 발생한 프로세스에게 SIGTRAP 시그널을 전송한다[6]. 리눅스 커널은 SIGTRAP 시그널을 전달받은 프로세스가 gdbserver의 자식 프로세스인 경우에는 해당 프로세스를 정지시키고 gdbserver에게 해당 프로세스의 제어를 넘긴다. 이후 개발자는 gdb server를 사용하여 해당 프로세스로부터 레지스터 및 메모리 정보 등을 읽거나 쓸 수 있고 프로세스의 수행 흐름을 제어할 수 있다. 하지만 SIGTRAP 시그널을 전달받은 프로세스가 gdbserver의 자식 프로세스가 아닌 경우에는 해당 프로세스는 커널에 의해서 종료된다. 그림 2에서는 gdbserver와 리눅스 커널 그리고 디버깅 프로세스 사이의 일반적인 프로세스 디버깅 과정을 보인다.

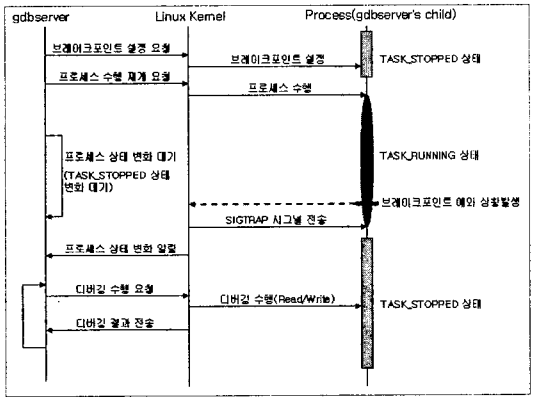


그림 2 gdbserver의 일반적인 프로세스 디버깅 과정

타겟 시스템의 gdbserver는 리눅스 커널이 제공하는 ptrace시스템 콜을 사용하여 개발자에게 프로세스 디버깅 기능을 제공한다. gdbserver가 ptrace시스템 콜을 사용하기 위해서는 gdbserver와 디버깅하려는 프로세스 사이에 부모-자식간의 관계가 성립되어야 한다[7]. 만일 gdbserver를 통해 디버깅 중인 프로세스가 새로운 프로세스를 생성하면 gdbserver와 새로이 생성된 프로세스 사이는 부모-자식 관계가 성립되지 않으므로 개발자는 gdbserver를 사용하여 디버깅 중인 프로세스에 의해 생성된 새로운 프로세스를 디버깅 할 수 없게 된다.

gdbserver와 부모-자식간의 관계를 갖는 프로세스들만을 gdbserver를 통하여 디버깅할 수 있는 이유는 다음과 같다. 리눅스 환경의 태스크 구조체에는 ptrace변수가 있다. 이 변수에 PT_PTRACED라는 값이 설정된 프로세스는 정지점 예외 상황 발생 시 커널에 의해 종료되지 않고 TASK_STOPPED상태가 되어 제어를 부모 프로세스(gdbserver)에게 넘긴다. 하지만 gdbserver와 해당 프로세스간에 부모-자식 관계가 성립되지 않는 경우에는 gdbserver가 ptrace 시스템 콜을 사용하여 해당 프로세스의 태스크 구조체 안에 있는 ptrace변수에 PT_PTRACED값을 설정할 수 없다. 따라서, 개발자는 gdbserver를 사용하여 gdbserver와 부모-자식 관계를 갖지 않는 프로세스를 디버깅 할 수 없게 되는 것이다. 다음 장에서는 gdbserver를 사용하여 부모-자식 관계를 갖지 않는 프로세스를 디버깅하는 방법을 제시한다.

3. 다중 프로세스 응용에 대한 원격 디버깅 기법

본 장에서는 2장에서 언급한 문제점들을 해결하기 위해 본 논문에서 제안하는 SPY_Library 라이브러리에 대해서 설명하고, SPY_Library 라이브러리를 이용하여 문제점들을 해결하는 방법을 제시한다.

3.1 다중 프로세스 원격 디버깅 시스템 구조

본 논문에서는 그림 3에서와 같은 SPY_Library 라이브러리를 사용한 원격 디버깅 시스템 구조를 제안한다.

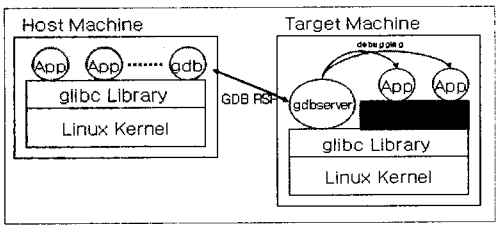


그림 3 다중 프로세스 디버깅을 위한 구조

그림 3에서 타겟 시스템의 gdbserver는 SPY_Library 라이브러리와 동적으로 링크되어 만들어진 응용 프로그램의 디버깅을 지원하기 위해 응용 프로그램을 자식 프로세스로 생성한다. 호스트 시스템의 gdb는 타겟 시스템의 gdbserver와 TCP/IP 통신을 하며 개발자로부터의 디버깅 요구에 대한 처리를 수행하고 해당 프로세스가 fork시스템 콜을 호출하는 경우 기존의 디버깅 중인 프로세스는 물론 새로 생성된 프로세스들을 디버깅하기 위한 기능을 지원한다[1][8].

3.2 SPY_Library

다중 프로세스 디버깅 지원에 필요한 기본 작업을 하기 위해서 gdb와 gdbserver는 디버깅 중인 프로세스가 fork 시스템 콜을 호출하는 시점을 알아야 한다. 본 연구에서는 임의의 프로세스가 시스템 콜을 호출하는 시점을 알기 위해서 라이브러리를 래핑 하는 방법을 사용

한다. 라이브러리 래핑 방법은 응용 프로그램에서 glibc 라이브러리에 있는 함수를 호출 시 실제로 glibc 라이브러리에 있는 함수보다 SPY_Library 라이브러리에 있는 함수를 먼저 호출되도록 하여 다중 프로세스 디버깅을 위해 필요한 작업을 진행 한 뒤에 glibc 라이브러리에 있는 실제로 응용 프로그램이 호출하려는 함수를 호출하도록 하는 방법이다. 만일 커널 레벨에서의 시스템 콜을 가로채는 방법을 사용한다면 커널 코드를 변경해야 하며 이것은 부가적으로 커널 버전에 따른 유지보수를 필요로 하게 된다. gdbserver는 LD_PRELOAD라는 링커(Linker) 환경변수를 이용하여 SPY_Library 라이브러리를 먼저 로딩(loading)함으로 응용 프로그램이 fork 시스템 콜을 호출하는 시점을 알아 낼 수 있다[9]. 이것은 glibc 라이브러리에 있는 fork 심볼(symbol)보다 SPY_Library 라이브러리에 있는 fork 심볼이 먼저 동적 바인딩(binding)을 수행하기 때문에 가능하다. 그림 4에서는 응용 프로그램이 fork시스템 콜을 호출하는 경우와 관련된 일반적인 호출 및 반환 절차를 보인다.

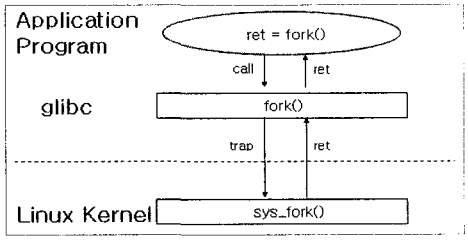


그림 4 fork() 시스템 콜에 따른 흐름도

그림 4와 같은 일반적인 시스템 콜의 수행 흐름에서 SPY_Library 라이브러리가 링커 환경 변수 LD_PRELOAD를 이용하여 응용 프로그램의 fork 시스템 콜을 가로채는 경우는 그림 5와 같다.

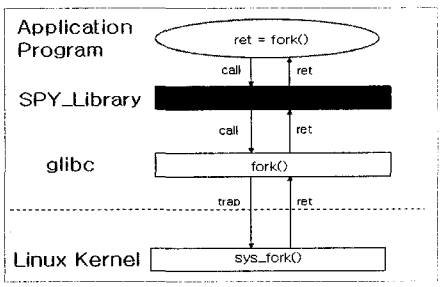


그림 5 LD_PRELOAD를 사용한 경우의 fork 시스템 콜 흐름도

LD_PRELOAD라는 링커 환경 변수를 사용함으로써 SPY_Library 라이브러리의 fork함수는 응용 프로그램에서 fork 시스템 콜을 호출했을 때 glibc 라이브러리의 fork 함수를 호출하기 전에 먼저 수행이 된다[6]. 응용 프로그램의 fork 시스템 콜은 SPY_Library 라이브러리의 fork 함수를 호출하여 필요한 작업을 수행하고 SPY_Library 라이브러리의 fork함수는 glibc 라이브러리의 fork 함수를 호출한다. 이와 같은 방법을 사용하여 디버깅 중인 응용 프로그램에서 fork 시스템 콜을 호출하는 경우 SPY_Library 라이브러리는 다중 프로세스 디버깅을 지원하기 위해 필요한 처리를 할 수 있다. 다음절에서는 SPY_Library 라이브러리를 이용하여 디버깅 중인 프로세스로부터 fork 시스템 콜에 의해 생성된 프로세스를 디버깅하는 방법을 보인다.

3.3 SPY_Library 라이브러리를 이용한 다중 프로세스 디버깅 기법

그림 6에서는 gdbserver를 통해 디버깅 중인 프로세스가 새로운 자식 프로세스를 생성하는 경우 SPY_Library 라이브러리는 새로 생성된 프로세스가 실행되기 전에 중단시키고 gdbserver에게 디버깅 중인 프로세스로부터 새로운 프로세스가 생성됨을 알리는 과정을 보인다.

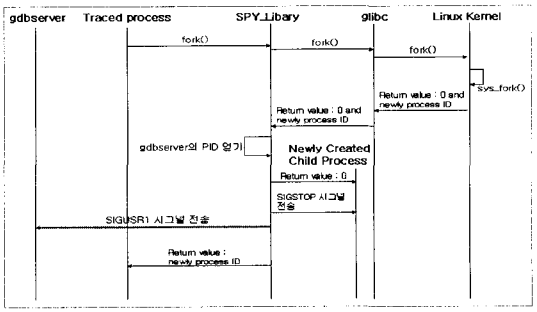


그림 6 디버깅 중인 프로세스가 새로운 프로세스 생성 시 gdbserver에게 새로운 프로세스의 생성을 알리는 과정

gdbserver에 의해 디버깅 중인 프로세스가 새로운 프로세스를 생성한 경우 SPY_Library 라이브러리는 새로이 만들어진 프로세스와 디버깅 중인 프로세스를 모두 정지시킨 후 gdbserver에게 SIGUSR1 시그널을 전달한다. 개발자는 SIGUSR1 시그널을 전달받은 gdbserver를 통해 앞으로 디버깅할 프로세스를 선택한다. 두 프로세스를 중단시킨 후 개발자는 gdbserver를 통해 디버깅하려는 프로세스를 ptrace 시스템 콜의 PTRACE_

ATTACH 인자를 사용하여 디버깅을 진행하고, 디버깅을 포기한 프로세스를 ptrace 시스템 콜의 PTRACE_DETACH 인자를 사용하여 디버깅을 중단한다. 그림 7에서는 호스트 시스템의 gdb와 타겟 시스템의 gdb server를 사용하여 원격으로 디버깅 중인 프로세스가 생성한 새로운 프로세스를 디버깅하는 과정을 보인다.

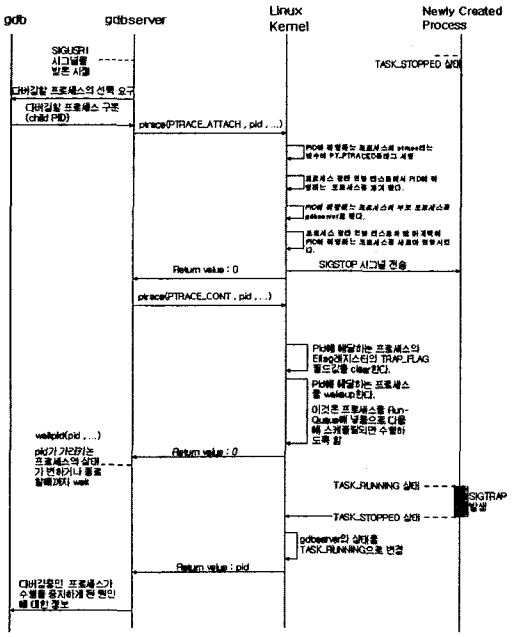


그림 7 SIGUSR1을 받은 후에 새로이 생성된 프로세스를 디버깅하는 과정

그림 7에서 gdbserver는 ptrace 시스템 콜의 PTRACE_ATTACH 인자를 사용하여 해당 프로세스의 태스크 구조체에 있는 ptrace변수에 PT_PTRACED값을 설정하고 gdbserver가 새로이 생성된 프로세스의 부모 프로세스가 되도록 프로세스간의 관계를 변경한다. 이후 개발자는 gdbserver를 사용하여 그림 2에서 설명한 일반적인 디버깅 절차를 통하여 새로 생성된 프로세스를 디버깅할 수 있다.

4. 실험 및 평가

본 논문에서는 실험을 위해 원격으로 디버깅 중인 프로세스가 fork 시스템 콜을 통하여 새로운 프로세스를 생성하면 개발자는 호스트 시스템의 gdb와 타겟 시스템의 gdbserver를 사용하여 새로이 생성되는 프로세스를 원격으로 디버깅하도록 하였다. 실험 환경은 다음과 같다. 타겟 시스템은 국내 Hybus라는 회사가 개발한 Hyper104 평가보드이다. 이 보드에는 32비트 RISC (Reduced Instruction Set Computers)프로세서인 SA1110(StrongArm, 206MHz)가 동작하고 있으며 32메가바이트 SDRAM(Synchronous DRAM)과 16메가바이트 플래쉬 메모리가 설치되어 있다. 타겟 시스템에서 동작하는 운영체제는 ARM(Advanced Risc Machine)프로세서에서 동작하도록 패치된 리눅스 커널 2.4.5-rmk7-np2이다. 호스트 시스템은 한컴 리눅스 2.2가 설치된 데스크탑 컴퓨터이다. 사용한 GNU/gdb는 릴리즈 버전인 5.0이다. 타겟 시스템에서의 프로그램 실행 결과와 리눅스 커널의 부팅과정에서의 메시지들을 호스트 시스템에서 확인할 수 있도록 하기 위하여 호스트 시스템에서는 minicom이라는 시리얼 통신 프로그램을 사용하고 이를 위해 타겟 시스템의 시리얼 포트와 호스트 시스템의 시리얼 포트를 연결하였다. 또한 개발자는 호스트 시스템에서 동작하는 minicom프로그램을 통하여 타겟 시스템의 gdbserver를 실행하게 되고 디버깅에 필요한 입력 데이타 역시 minicom프로그램을 통하여 입력하게 된다.

본 논문은 fork 시스템 콜에 의한 다중 프로세스 구조의 응용 프로그램을 원격으로 디버깅하는 기법에 초점을 맞추고 있으므로 그림 8의 새로운 자식 프로세스를 생성하는 간단한 프로그램을 테스트 프로그램으로 사용하였다. 이 테스트 프로그램은 타겟 시스템에서 실행되며 개발자는 호스트 시스템의 gdb와 타겟 시스템의 gdbserver를 사용하여 테스트 프로그램을 디버깅할 수 있다. 디버깅 및 실행 결과는 호스트 시스템에서 실행하는 minicom 프로그램을 통하여 호스트 시스템에서 확인할 수 있다.

```
int main()
  {
    int pid;
    printf("Test Program for Multi-process debugging...\n");
    pid = fork();

    if(pid > 0)
    {
        printf("This is Parent Process...[pid = %d]\n" ,
        getpid());
    }
    else if(pid == 0)
    {
        printf("This is Child Process...[pid = %d]\n" ,
        getpid());
    }
    return 0;
  }
```

그림 8 테스트 프로그램

그림 9에서는 타겟 시스템에서 gdbserver를 실행시키는 방법과 개발자가 호스트 시스템의 gdb와 타겟 시스템의 gdbserver를 사용하여 본 논문에서 사용한 테스트 프로그램을 디버깅 한 결과 중 타겟 시스템에서의 출력 결과를 보인다. gdbserver와 디버깅 프로그램은 타겟 시스템에서 수행되며 이들 프로그램의 실행 과정에서의 출력 결과물은 타겟 시스템에서 호스트 시스템으로 시리얼 케이블을 통하여 호스트 시스템의 minicom 프로그램에게 전달되어 지고 개발자는 호스트 시스템에서 minicom을 사용하여 확인 할 수 있게 된다. 개발자는 타겟 시스템의 IP주소와 포트 번호 그리고 디버깅하려는 프로그램의 실행 파일을 인자로 하여 gdbserver를 실행시킨다. gdbserver는 내부적으로 자식 프로세스를 생성하고 자식 프로세스는 ptrace 시스템 콜을 통하여 부모 프로세스인 gdbserver의 제어를 받을 수 있도록 자신의 태스크 구조체 안의 ptrace변수에 PT_PTRACED값을 설정한다. 그리고 자식 프로세스는 exec시스템 콜을 사용하여 디버깅하려는 프로그램의 이미지로 실행하게 된다. 이후 gdbserver는 호스트 시스템에서 동작하는 gdb의 접속을 기다리게 된다. 호스트 시스템의 gdb로부터 접속이 정상적으로 이루어지면 타겟 시스템의 gdbserver는 개발자가 gdb를 통하여 입력한 디버깅 작업을 수행하게 된다.

그림 9에서 현재 디버깅 중인 프로세스가 새로운 프로세스를 fork 시스템 콜을 통하여 생성하게 되면 현재 디버깅 중인 프로세스로부터 새로운 프로세스가 생성됨을 SPY_Library 라이브러리가 gdbserver에게 알려주고 gdbserver는 개발자에게 앞으로 디버깅하려는 프로세스의 선택을 요구한다. 개발자는 "Which process do you want to debug? (pid)"이라고 gdbserver가 물어오면 디버깅하려는 프로세스의 pid를 호스트 시스템에서 실행중인 minicom이라는 통신 프로그램을 사용하여 입력하면 된다. 본 논문에서는 gdb와 gdbserver의 구조를 가급적 변경하지 않는 범위에서 다중 프로세스 디버깅 방법에 초점을 맞추고 있으므로 개발자는 디버깅하려는 프로세스를 호스트 시스템의 gdb를 사용해서 선택하는 것이 아니라 타겟 시스템의 gdbserver를 사용하여 선택하도록 하였다. 이렇게 한 이유는 앞에서 언급했지만 실험을 위해 가급적 gdb와 gdbserver의 코드를 변경하지 않도록 했기 때문이다. 하지만 gdb와 gdbserver 간의 메시지 프로토콜에 적합하도록 새로운 메시지 타입을 설정하고 gdb와 gdbserver를 수정한다면 개발자는 호스트 시스템의 gdb를 사용하여 앞으로 디버깅하려는 프로세스를 선택할 수 있다. 만일 이러한 부분을 구현한다면 그림 10의 "Continuing."이라는 문자열이 출력된 다음 라인에서 개발자에게 디버깅을 진행 할 프로세스의 선택을 요구하게 하면 된다.

본 실험에서는 디버깅 중인 프로세스가 새로운 프로세스를 생성한 경우 새로이 생성된 프로세스의 디버깅 가능 여부에 초점을 맞추고 있으므로 디버깅 진행 프로세스의 선택요구에 새로이 생성된 프로세스의 pid값을 입력하여 디버깅을 진행하였다. "This is Child Process...[pid = 185]"라는 결과를 통해서 알 수 있듯이 디버깅을 진행한 대상 프로세스가 기존의 디버깅 중이었던 프로세스가 아니라 새로이 생성된 프로세스임을 확인 할 수 있다. 결국 gdbserver를 통한 디버깅이 새로 생성된 자식 프로세스로 진행됨을 알 수 있다. 마지막으로 개발자는 디버깅 진행과정에서의 출력을 호스트 시스템의 minicom프로그램을 통해서 확인할 수 있게 된다.

```
[root@openES sim]$./gdbserver 203.252.53.135:33333 ./test
Process ./test created; pid = 184
Remote debugging using 203.252.53.135:33333
Test Program for Multi-process debugging...
Which process do you want to debug? (pid) 185
This is Child Process...[pid = 185]

Child exited with retcode = 0

Child exited with status 19
GDBserver exiting
 [root@openES sim]$
```

그림 9 타겟 시스템에서 gdbserver의 수행결과

그림 10에서는 호스트 시스템에서 gdb를 실행시키는 방법과 개발자가 호스트 시스템의 gdb와 타겟 시스템의 gdbserver를 사용하여 본 논문에서 사용한 테스트 프로그램을 디버깅 한 결과 중 호스트 시스템에서의 결과를 보인다. 개발자는 타겟 시스템에서 gdbserver를 실행시킬 때 사용한 인자들을 사용하여 호스트 시스템의 gdb가 타겟 시스템의 gdbserver에 TCP 소켓연결이 되도록 한다. 이후 개발자는 디버깅하려는 프로그램 실행 파일로부터 심볼 정보를 읽어들이고 원하는 경우엔 심볼 정보들을 바탕으로 하여 정지점을 설정할 수 있다. 본 실험에서는 정지점을 설정하지는 않고 디버깅을 진행하였다. 디버깅을 진행하는 도중 타겟 시스템에서 gdbserver가 개발자에게 앞으로 디버깅을 진행할 프로세스의 선택을 요구하게 되면 그림 10에서 gdb는 "Continuing."이라는 문자열을 출력하고 개발자가

```
[sim@grapevine gdb]$./gdb
GNU gdb 5.0
Copyright 2000 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 "i686-pc-linux-gnu".
(gdb) target remote 203.252.53.135:33333
Remote debugging using 203.252.53.135:33333
0x00000000 in ?? ()
(gdb) symbol-file ./test
Reading symbols from ./test...done.
(gdb) list
warning: Source file is more recent than executable.

1       #include <stdio.h>
2
3       int main()
4       {
5               int pid;
6
7               printf("Test Program for Multi-process debugging...\n");
8
9               pid = fork();
10
(gdb)
11              if(pid > 0)
12              {
13                      printf("This is Parent Process...[pid = %d]\n" , getpid());
14              }
15              else if(pid == 0)
16              {
17                      printf("This is Child Process...[pid = %d]\n" , getpid());
18              }
19              return 0;
20      }
(gdb) continue
Continuing.     <=== 타겟 시스템의 gdbserver가 개발자의 디버깅 할 프로세스의 선택을 기다리는 경우
                <=== gdb도 이 상황에서 기다리게 됨

Program received signal SIGCONT, Continued.
0x02000278 in _init ()
(gdb) continue
Continuing.

Program exited normally

(gdb)
```

그림 10 호스트 시스템에서 gdb의 결과

gdbserver를 사용해서 디버깅하려는 프로세스를 선택할 때까지 기다리게 된다. 앞에서도 언급했지만 만일 호스트 시스템의 gdb를 사용하여 디버깅을 진행할 프로세스를 선택하도록 gdb와 gdbserver를 변경한다면 이 부분에서 개발자로 하여금 앞으로 디버깅을 진행할 프로세스를 선택하도록 하면 된다. 개발자는 디버깅을 진행하려는 프로세스를 gdbserver를 통하여 선택하고 디버깅 작업을 진행하게 된다.

아래의 표 1에서는 리눅스 환경에서 fork 시스템 콜에 의한 다중 프로세스 구조의 프로그램을 디버깅할 수 있도록 지원하는 ETNUS사의 TotalView 프로그램과 본 논문에서 제안하는 방법을 비교한다.

표 1 TotalView 프로그램과 SPY_Library 라이브러리를 이용한 방법의 비교 테이블

	ETNUS사의 TotalView	SPY_Library를 이용한 방법
프로그램의 이미지 크기	79485 bytes	21550 bytes + 6307 bytes (SPY_Library)
라이브러리 링크 방법	Static Linking	Dynamic Linking
다중 프로세스 응용의 원격 디버깅 지원 여부	지원 불가	지원 가능
다중 프로세스 응용의 로컬 디버깅 지원 여부	지원 가능	지원 가능

TotalView 버전 5.0.0-4를 사용하여 성능 비교를 하였으며 성능 비교를 위해 사용한 프로그램은 그림 8의 테스트 프로그램이다. 표 1을 통해서 알 수 있듯이 둘 사이에 3배 이상의 프로그램 이미지의 크기 차이가 나는 것을 알 수 있다. 이러한 결과가 나타나는 이유는 프로그램 컴파일 시 SPY_Library 라이브러리와 TotalView가 사용하는 라이브러리의 링크 방법에 따른 결과이다. TotalView 프로그램은 원격 디버깅을 지원하지 않고 이 프로그램을 사용하여 디버깅하려는 프로그램들은 정적 라이브러리 사용에 따른 이미지의 크기 때문에 본 논문에서 제안하는 SPY_Library를 사용한 원격 디버깅 방법이 비교적 부족한 자원을 갖는 임베디드 시스템에서 동작하는 다중 프로세스 디버깅 도구로 더 적합하다.

5. 결 과

본 연구에서는 커널의 변경 없이 라이브러리 레벨에서의 래핑 방법을 통하여 원격으로 디버깅 중인 프로세스가 새로운 프로세스를 생성한 경우 새로이 생성된 프로세스를 원격으로 디버깅할 수 있는 방법을 알아보았다. 커널 코드의 변경 없이 라이브러리를 래핑 함으로서 안정적인 커널을 사용할 수 있고 코드의 개발이 커널 모드에서 동작하는 코드를 만드는 것이 아니라 사용자 모드에서 동작하는 코드를 만드는 것이므로 상대적으로 쉽고, 수행 결과를 테스트하기에도 수월하다는 장점을 얻을 수 있다.

SPY_Library 라이브러리를 이용한 디버깅 방법은 디버깅 중에 새로운 프로세스가 생성되는 경우 기존의 디버깅 중인 프로세스뿐만 아니라 새로이 생성된 프로세스도 디버깅할 수 있다. 하지만 현재 gdbserver는 임의의 순간에 하나의 프로세스만 디버깅할 수 있도록 지원하기 때문에 여러 프로세스를 번갈아 가면서 디버깅할 수 없다. 이것은 현재 개발되어 있는 gdbserver의 구조적인 문제로 인하여 발생한 것이며, 만일 gdbserver를 사용하여 디버깅 중인 프로세스가 새로운 자식 프로세스를 생성하는 경우 타겟 시스템에서 새로운 gdbserver를 실행 시켜서 새로이 생성된 자식 프로세스를 ptrace 시스템 콜의 PTRACE_ATTACH 인자를 사용하여 디버깅할 수 있도록 지원한다면 임의의 순간에 여러 프로세스들을 번갈아 가면서 디버깅을 할 수 있게 될 것이다. 이러한 경우엔 로컬에는 하나의 gdb만 있으면 되고 타겟 시스템에는 디버깅하려는 프로세스의 수만큼 gdbserver가 필요하게 된다. 개발자의 손쉬운 사용을 위하여 gdb와 여러 개의 gdbserver사이를 연결해 주는 관리 프로그램이 있어야 할 것이다. 개발자는 로컬의 gdb를 사용하여 타겟 시스템의 관리 프로그램에게 디버깅 명령을 전달하면 타겟 시스템의 관리 프로그램은 임의의 순간에 개발자가 디버깅하길 원하는 프로세스와 연결된 gdbserver에게 개발자의 디버깅 명령을 전달할 것이며 타겟 시스템의 gdbserver의 디버깅 결과는 타겟 시스템의 관리 프로그램을 통하여 로컬의 gdb에게 전달되던가 또는 직접 로컬의 gdb로 전달하게 될 것이다. 이렇게 임의의 순간에 여러 프로세스들을 번갈아 가면서 디버깅할 수 없는 이유는 gdbserver의 구조적인 문제점에 해당하는 것이며 앞에서 언급한 임의의 순간에 다중 프로세스 디버깅을 하기 위한 방법에 필요한 기본적인 아이디어는 본 연구에서 제안하는 방법이다. 향후 연구 과제는 gdbserver가 임의의 순간에 다중 프로세스 디버깅을 지원할 수 있는 구조로 변경하는 것이다. 또한 새로운 프로세스의 생성 시 일관성 있는 원격 디버깅을 위해 디버깅하려는 프로세스의 선택을 타겟 시스템의 gdbserver를 통해서가 아닌 호스트 시스템의 gdb를 통해서 가능하도록 하는 것이다.

참 고 문 헌

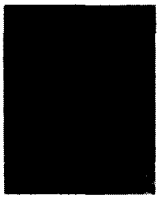
[1] Richard M. Stallman, Debugging with GDB, 4th ed., Cygnus Support, 1996.

[2] 임형택, 심현철, 손승우, 김홍남, 김채규, "Q+P Esto의 원격 개발을 지원하는 타겟에이전트", 한국정보처리학회 2001년 추계 학술대회, 제 8권, 제 2호, pp. 671-674, 2001.

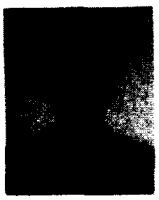
[3] Etnus, Totalview Getting Started, http://www.etnus.com/Products/TotalView/started/getting_started2.html, 2001.

[4] Nathan Field, Debugging Embedded Linux

Application, http://embedded.linuxjournal.com/magazine/issue06/4897/?sid=17, November, 2001.
[5] Intel, Intel Atchitecture Software Developer's Manual, Vol. 3, 1999.
[6] Daniel P. Bovet and Marco Cesati, Understanding the Linux Kernel, O'Reilly, 2001.
[7] Uresh Vahalia, Unix Internals, Prentice Hall, 1996.
[8] Greg Rose, Embedded Linux 101, http://www.ecnmag.com/ecnmag/issues/2001/12012001/ec1dsc100_.asp, 2001.
[9] Sun Microsystems Inc., Linker & Libraries Guide, October, 1998.



심 현 철
2000년 2월 성균관대학교 정보통신공학부 학사. 2000년 ~ 현재 성균관대학교 정보통신공학부 석사 과정. 관심분야는 Embedded Linux, 시스템 소프트웨어 운영체제



강 용 혁
1996년 2월 성균관대학교 정보공학과 학사. 1998년 2월 성균관대학교 정보공학과 컴퓨터공학전공 석사. 2000년 2월 성균관대학교 정보통신공학부 박사과정 수료. 관심분야는 분산 시스템, 이동 컴퓨팅 시스템, Mobile Ad-hoc networks

엄 영 익
1983년 2월 서울대학교 계산통계학과 학사. 1985년 2월 서울대학교 대학원 전산과학전공 석사. 1991년 8월 서울대학교 대학원 전산과학전공 박사. 2000년 9월 ~ 2001년 8월 Dept. of Info. and Comm. Science at UCI 방문교수. 현재 성균관대학교 정보통신공학부 교수. 관심분야는 분산 시스템, 이동 컴퓨팅 시스템, 이동 에이전트, 시스템 소프트웨어