임베디드 Linux 시스템 기반 프로세스 동시 디버깅을 지원하는 원격 디버거 설계 및 구현

심 현 철[†] · 강 용 혁[†] · 엄 영 익[††]

요 약

임베디드 Linux 환경에서 gdb와 gdbserver를 사용하여 현재 디버깅 중인 프로세스로부터 fork 시스템 콜에 의해 생성된 새로운 프로세스를 원격으로 동시에 디버깅할 수는 있으나 이를 위해서 개발자는 부가적인 코딩뿐만 아니라 새로운 프로세스가 생성될 때마다 원격 디버깅을 위해 별도의 gdb 및 gdbserver를 구동 시켜야 하는 등 불필요한 작업을 해야 하므로 효율적인 디버깅 작업을 진행하기 어렵다. 본 논문에서는 Linux 커널의 변경 없이 라이브러리 래핑 방법을 이용하여 원격 시스템에서 동작하는 다중 프로세스들을 동시에 디버깅할 수 있도록 지원하기 위한 mgdb 라이브러리와 mgdbserver를 제안한다. 또한, 파이프를 통하여 데이터를 주고받는 부모-자식 관계의 프로세스들을 원격으로 동시에 디버깅하는 실험을 통하여 본 논문에서 제안한 방법이 기존의 방법보다 더 효율적임을 보인다.

Design and Implementation of a Remote Debugger for Concurrent Debugging of Multiple Processes based on Embeded Linux System

Hyun Chul Sim[†] · Yong Hyeog Kang[†] · Young Ik Eom[††]

ABSTRACT

In the embedded Linux environments, developers can concurrently debug multiple processes that have parent-child relationships using multiple gdbs and gdbservers. But, it needs additional coding efforts and messy works of activating another gdbs and gdbservers for each created process, and so, it may be inefficient in the viewpoint of developers. In this paper, we propose a mgdb library and mgdbserver that supports concurrent debugging of multiple processes in the embedded Linux systems by using the library wrapping mechanism without modifying the kernel. Also, through the experimentation of concurrent debugging for multiple processes that communicate by an unnamed pipe, we show that our proposed debugging mechanism is more efficient than the preexisting mechanisms.

키워드 : 디버깅(Debugging), 다중 프로세스 디버깅(Multi-process debugging), 임베디드 리눅스(Embedded Linux)

1. 서 론

최근 Linux 기반의 임베디드 시스템에서는 원격 디버깅 도구로 gdb를 많이 사용하고 있다[1, 2]. gdb는 소스가 공개되어 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[3-5].

Linux 환경에서 gdb와 gdbserver를 사용하여 원격 디버깅 중인 프로세스가 fork 시스템 콜을 호출하여 생성한 프로세스들을 원격으로 동시에 디버깅하기 위해서는 미리 개발자가 직접 새로이 생성될 자식 프로세스의 코드부분에 sleep 함수 코드를 삽입하고 자식 프로세스가 생성된 후 sleep 코드를 실행하여 정지 상태가 되면 타겟 시스템에서 새로운 gdbserver를 실행하여 새로이 생성된 프로세스와 연결해야 한다. 또한 호스트 시스템에서는 새로운 gdb를 실행하여 타겟 시스템의 새로운 gdbserver와 연결해야만 하는 추가의 작업이 필요하다[3]. 이러한 방법으로 다중 프로세스들을 원격으로 동시에 디버깅하는 경우에는 디버깅을 하려는 프로세스의 수만큼의 호스트 시스템에는 gdb, 타겟 시스템에는 gdbserver가 필요하게 된다. 그러므로 개발자는 다중 프로세스들을 원격으로 동시에 디버깅하기 위해 호스트 및 타겟 시스템에서 여러 개의 터미널을 다루어야 하므로 간편한 디버깅 작업을 할 수 없게 된다.

본 논문에서는 현재 디버깅 중인 프로세스는 물론 현재 디버깅 중인 프로세스가 fork 시스템 콜을 호출하여 새로이 생성한 자식 프로세스들을 개발자가 원격지에서 동시에 디버깅할 수 있도록 지원하는 mgdb 라이브러리와 mgdbserver를 설계하고 구현한다. 제안된 mgdb 라이브러리와 mgdbserver

† 준 회 원 : 성균관대학교 대학원 정보통신공학부
†† 종신회원 : 성균관대학교 정보통신공학부 교수
논문접수 : 2002년 8월 16일, 심사완료 : 2003년 7월 9일

를 사용하여 개발자는 소켓(socket), 파이프(pipe)등과 같은 프로세스간 통신(interprocess communication)방법을 통해 데이터를 주고받는 프로세스들을 원격지에서 동시에 디버깅할 수 있게 된다. 또한 개발자는 호스트 시스템에서 하나의 gdb를 통하여 타겟 시스템에서 실행 중인 다중 프로세스들을 디버깅할 수 있어 효율적인 디버깅 작업을 할 수 있다. 본 논문은 다음과 같이 구성되어 있다. 2장에서는 gdb와 gdbserver를 사용하여 디버깅 중인 프로세스가 fork 시스템 콜을 사용하여 생성한 새로운 프로세스를 원격으로 디버깅하는 경우에 발생하는 문제점에 대해서 분석한다. 3장에서는 mgdb 라이브러리와 mgdbserver를 제안하며 4장에서는 mgdb 라이브러리와 mgdbserver를 사용하여 원격으로 동시에 다중 프로세스들을 디버깅하는 실험 결과를 설명한다. 또한 ETNUS사의 TotalView 프로그램과 본 논문에서 제안한 방법과의 성능 비교를 하며 마지막으로 5장에서는 결론 및 향후 연구 방향에 대해서 기술한다.

2. 관련 연구

Linux 환경에서 다중 프로세스 디버깅을 지원하는 상용 도구로는 ETNUS사의 TotalView라는 것이 있다[6]. 개발자가 이 프로그램을 사용하여 fork 시스템 콜을 통한 다중 프로세스 구조의 프로그램을 디버깅하기 위해서는 libdbfork.a라는 정적 라이브러리를 링크(link)하여 프로그램 이미지를 생성해야 하므로 프로그램 이미지의 크기가 커지게 된다. 또한 이 프로그램은 원격 디버깅 환경이 아닌 로컬 디버깅 환경에서의 다중 프로세스 디버깅 만을 지원하기 때문에 비교적 부족한 자원을 갖는 임베디드 시스템을 위한 원격 디버깅 도구로는 부적절하다.

본 장에서는 Linux 환경에서 호스트 시스템의 gdb와 타겟 시스템의 gdbserver를 사용하여 다중 프로세스 구조의 응용프로그램을 원격 디버깅하는 방법과 문제점에 대해서 알아본다.

2.1 기존의 gdb와 gdbserver를 이용한 다중 프로세스의 원격 디버깅 방법

(그림 1)은 호스트 시스템의 gdb와 타겟 시스템의 gdbserver를 사용하여 다중 프로세스를 디버깅하기 위해 개발자가 해야 하는 필요한 작업을 나타낸다. (그림 1)에서 #ifdef와 #endif로 되어 있는 사이 부분이 다중 프로세스를 디버깅하기 위해 개발자가 추가해야 하는 코드에 해당한다.

```
if ( ( pid = fork() ) == 0 )
{
    /* child */
    #ifdef DEBUG_FORKS    // gdb와 gdbserver를 사용하여
    sleep () ;            // 다중 프로세스를 디버깅하기
                          // 위해 개발자가 추가해야 하는
    #endif                // 코드
    DoChildThing () ;
}
else
{
    /* parent */
    DoParentThing () ;
}
```

(그림 1) 원격에서 gdb 및 gdbserver를 이용한 다중 프로세스의 디버깅 방법

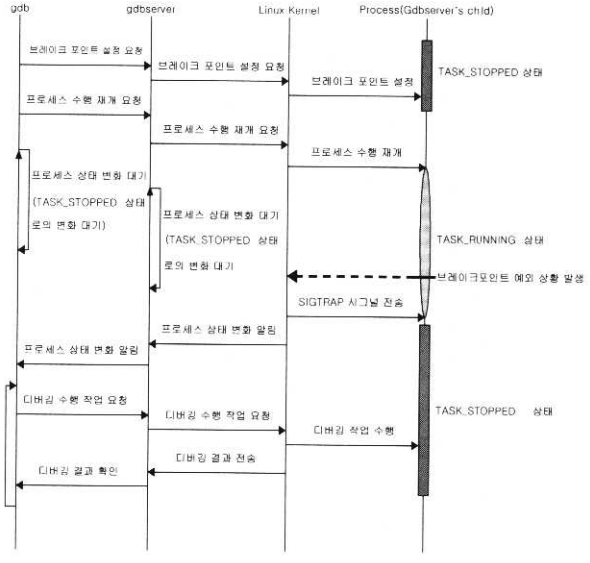
이 방법은 프로그램 컴파일 시간에 해당 프로그램을 디버깅할 것인지를 결정해야 한다는 단점이 있고 그로 인하여 해당 프로그램의 소스코드를 계속 갖고 있어야 한다. 또한 새로이 생성된 자식 프로세스를 디버깅하기 위해서 개발자는 타겟 시스템에서 새로운 gdbserver를 실행해야 하며 호스트 시스템에서도 새로운 gdb를 실행해야 한다. 디버깅하려는 프로세스의 수가 많아질수록 개발자가 관리해야 하는 터미널의 수가 많아지기 때문에 개발자는 호스트에서 하나의 gdb를 통해서가 아닌 여러 개의 gdb를 통해서 다중 프로세스 디버깅을 진행하게 되므로 쉽고 효율적인 디버깅을 할 수 없다.

2.2 gdb와 gdbserver를 사용한 다중 프로세스 원격 디버깅의 문제점

인텔의 x86 기반의 프로세서는 op-code가 "cc"인 명령어(INT3)를 수행하려 하면 정지점 예외 상황을 발생시킨다[7]. 정지점 예외 상황이 발생하면 Linux 커널은 커널 모드 스택에 대부분의 레지스터 값을 저장하고 예외 상황 처리 함수를 호출한다. 예외 상황 처리 함수에서는 예외 상황이 발생한 프로세스에게 SIGTRAP 시그널을 전송한다[8]. Linux 커널은 SIGTRAP 시그널을 전달받은 프로세스가 gdbserver의 자식 프로세스인 경우에는 해당 프로세스를 정지시키고 gdbserver에게 해당 프로세스의 제어를 넘긴다. 이후 개발자는 호스트시스템의 gdb와 타겟 시스템의 gdbserver를 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 2)에서는 gdbserver와 Linux 커널 그리고 디버깅 프로세스 사이의 일반적인 프로세스 디버깅 과정을 보인다.

타겟 시스템의 gdbserver는 Linux 커널이 제공하는 ptrace 시스템 콜을 사용하여 개발자에게 프로세스 디버깅 기능을 제공한다. gdbserver가 ptrace 시스템 콜을 사용하기 위해서는 gdbserver와 디버깅하려는 프로세스 사이에 부모-자식간의 관계가 성립되어야 한다[9]. 만일 gdbserver를 통해 디버깅 중인 프로세스가 새로운 프로세스를 생성하면 gdbserver

와 새로이 생성된 프로세스 사이는 부모-자식 관계가 성립되지 않으므로 개발자는 gdbserver를 사용하여 디버깅 중인 프로세스에 의해 생성된 새로운 프로세스를 디버깅할 수 없게 된다.



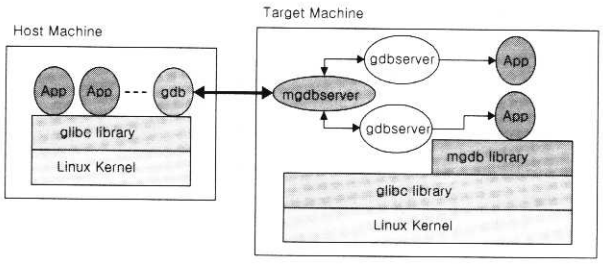
(그림 2) gdb와 gdbserver를 사용한 일반적인 프로세스 디버깅 과정

개발자가 gdb와 gdbserver를 사용하여 현재 디버깅 중인 프로세스가 fork 시스템 콜에 의해 생성한 새로운 자식 프로세스를 디버깅하기 위해서는 (그림 1)에서 보인 것처럼 자식 프로세스에 부가적인 코드를 삽입하여 새로이 생성된 프로세스가 스케줄링 되어 수행을 시작하려 할 때 커널에 의해 종료되지 않도록 해야 한다. 또한 타겟 시스템에서 새로운 gdbserver를 실행하여 새로 생성되어 멈추어 있는 프로세스를 gdbserver의 자식 프로세스로 연결하고 이후 진행될 디버깅을 위해서 멈추어 있는 프로세스의 태스크 구조체 안의 ptrace 변수에 PT_PTRACED 값을 설정한다. 또한 개발자는 호스트 시스템에서 gdb를 실행하여 타겟 시스템의 gdbserver와 연결한다. 모든 연결이 정상적으로 이루어지면 개발자는 호스트 시스템에서 gdb를 사용하여 타겟 시스템에서 새로이 생성된 프로세스를 디버깅할 수 있다. 하지만 이 방법은 2.1절에서 보인 바와 같이 개발자에게 쉽고 효율적인 디버깅 환경을 제공하지 않는다.

3. 다중 프로세스들의 동시 디버깅을 지원하는 원격 디버거의 설계 및 구현

3.1 원격 디버거 시스템 구조

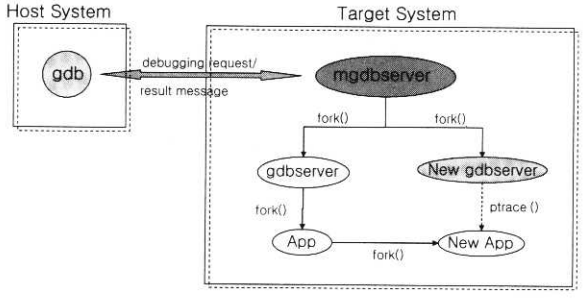
본 논문에서는 (그림 3)에서와 같은 mgdb 라이브러리와 mgdbserver를 사용하여 다중 프로세스들의 동시 디버깅을 지원하는 원격 디버거 시스템 구조를 제안한다.



(그림 3) 다중 프로세스 디버깅을 위한 구조

(그림 3)에서 타겟 시스템의 mgdbserver는 호스트 시스템의 gdb와 통신을 하며 gdb의 디버깅 요청을 적절한 gdb server에게 전달하고 디버깅 결과를 다시 gdb에게로 전달하는 교환기(switch)의 역할을 한다. mgdbserver를 사용함으로써 개발자는 원하는 순간에 원하는 프로세스들을 선택하면서 다중 프로세스들을 동시에 디버깅할 수 있다. 또한 mgdbserver는 현재 디버깅 중인 프로세스가 fork 시스템 콜을 호출하여 새로운 자식 프로세스를 생성한 경우 새로이 생성된 자식 프로세스의 디버깅을 지원하기 위해 타겟 시스템에 새로운 gdbserver를 생성하고 새로 생성된 gdbserver와 새로이 생성된 프로세스를 자동으로 연결시킨다. 이와 같은 구조의 장점으로는 기존의 gdb와 gdbserver를 사용한 다중 프로세스 원격 디버깅시 나타났던 문제점인 개발자가 디버깅하려는 프로세스의 수만큼 호스트 시스템의 터미널에서 gdb를 실행할 필요가 없다는 것이다. 개발자는 호스트 시스템에서 하나의 gdb만을 사용하여 타겟 시스템의 다중 프로세스들을 선택해 가면서 동시에 디버깅을 진행 할 수 있다.

(그림 4)에서는 개발자가 호스트 시스템의 gdb와 타겟 시스템의 mgdbserver로 다중 프로세스 구조의 응용 프로그램을 원격 디버깅하는 경우 현재 디버깅 중인 프로세스가 fork 시스템 콜을 통하여 새로운 자식 프로세스를 생성할 때 새로 생성된 프로세스를 디버깅하기 위한 프로세스들간의 관계를 나타낸다.



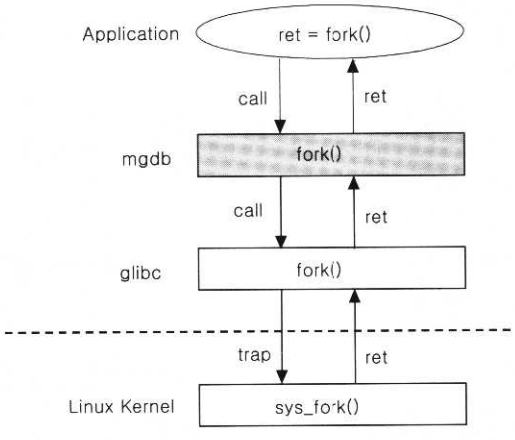
(그림 4) 현재 디버깅 중인 응용 프로그램이 새로운 자식 프로세스를 생성한 경우의 프로세스간 관계

3.2 mgdb 라이브러리의 설계

다중 프로세스 디버깅 지원에 필요한 기본 작업을 하기

위해서 mgdbserver는 디버깅 중인 프로세스가 fork 시스템 콜을 호출하는 시점을 알아야 한다. 본 연구에서는 임의의 프로세스가 시스템 콜을 호출하는 시점을 알기 위해서 glibc 라이브러리를 래핑(wrapping)하는 방법을 사용한다. 라이브러리 래핑 방법은 응용 프로그램에서 glibc 라이브러리에 있는 함수를 호출 시 실제로 glibc 라이브러리에 있는 함수보다 mgdb 라이브러리에 있는 함수를 먼저 호출되도록 하여 다중 프로세스 디버깅을 위해 필요한 작업을 진행한 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. mgdbserver는 Linux 동적 링커(linker)의 대치(interposition)기능을 사용하여 mgdb 라이브러리를 먼저 로딩(loading)함으로 mgdbserver의 자식 프로세스인 gdbserver와 부모-자식 관계를 갖는 응용 프로그램이 fork 시스템 콜을 호출하는 시점을 알아 낼 수 있다[10]. 이것은 glibc 라이브러리에 있는 fork 심볼(symbol)보다 mgdb 라이브러리에 있는 fork 심볼이 먼저 동적 바인딩(binding)을 수행하기 때문에 가능하다.

(그림 5)는 Linux 동적 링커의 대치기능으로 인해 mgdb 라이브러리에 있는 fork 함수가 glibc 라이브러리에 있는 fork 함수보다 응용 프로그램에 있는 fork 함수에 먼저 바인딩(binding)되는 것을 보여주고 있다.

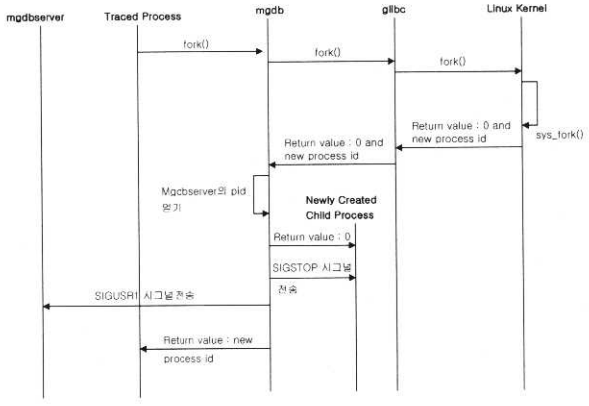


(그림 5) 동적 링커의 대치기능을 사용한 경우의 fork 시스템 콜 흐름도

동적 링커의 대치기능을 이용하여 디버깅 중인 응용 프로세스에서 fork 시스템 콜을 호출하는 경우 mgdb 라이브러리는 다중 프로세스 디버깅을 지원하기 위해 필요한 처리를 할 수 있다.

3.3 mgdb 라이브러리와 mgdbserver를 이용한 다중 프로세스들의 원격 동시 디버깅

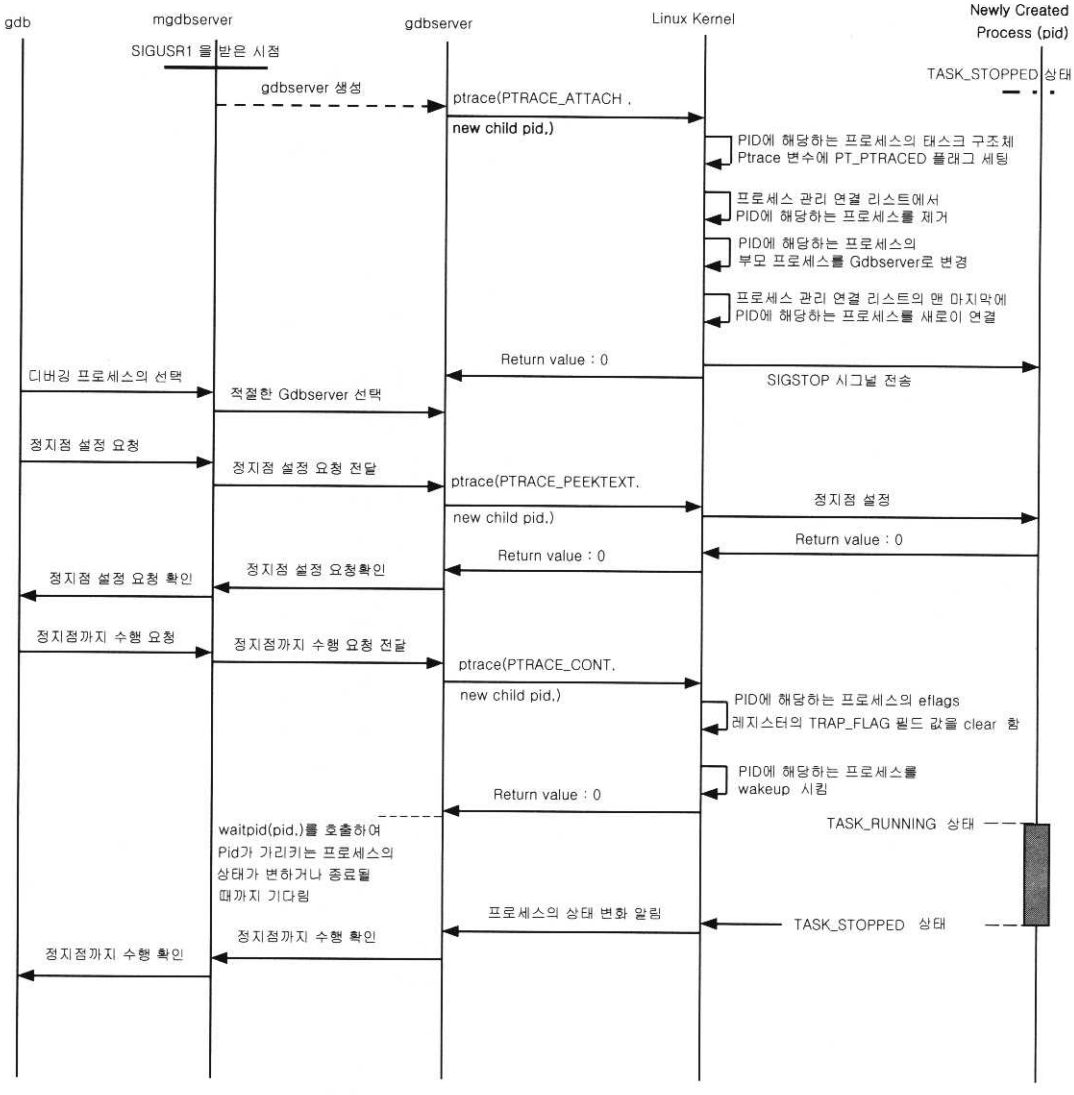
(그림 6)에서는 mgdbserver의 자식 프로세스인 gdbserver를 통해 디버깅 중인 프로세스가 새로운 자식 프로세스를 생성하는 경우 mgdb 라이브러리는 새로 생성된 프로세스가 실행되기 전에 커널에 의한 종료를 막기 위해 새로 생성된 프로세스를 중단시키고 mgdbserver에게 디버깅 중인 프로세스로부터 새로운 프로세스가 생성됨을 시그널을 통해 알리는 과정을 보인다.



(그림 6) 디버깅 중인 프로세스가 새로운 프로세스 생성시 mgdbserver에게 새로운 프로세스의 생성을 알리는 과정

mgdbserver의 자식 프로세스인 gdbserver에 의해 디버깅 중인 프로세스가 fork 시스템 콜을 통하여 새로운 프로세스를 생성한 경우 mgdb 라이브러리는 새로이 만들어진 프로세스를 정지시킨 후 mgdbserver에게 SIGUSR1 시그널을 전달한다. mgdbserver는 SIGUSR1을 전달받으면 새로운 gdbserver를 실행하고 gdbserver와 새로 생성되어 디버깅을 진행할 정지 상태의 프로세스를 연결시킨다. gdbserver와 정지 상태의 프로세스를 연결시키는 과정은 ptrace 시스템 콜의 PTRACE_ATTACH 인자를 사용하여 새로 생성되어 디버깅을 진행할 정지 상태의 프로세스를 gdbserver의 자식 프로세스로 변경하고 gdbserver와 새로 생성되어 디버깅을 진행할 프로세스간에 부모-자식간의 관계를 갖도록 하여 gdbserver가 ptrace 시스템 콜을 통하여 해당 프로세스의 디버깅을 지원할 수 있도록 하는 것을 의미한다. (그림 7)에서는 호스트 시스템의 gdb와 타겟 시스템의 mgdbserver 및 gdbserver를 사용하여 원격으로 디버깅 중인 프로세스가 fork 시스템 콜을 통해 생성한 새로운 프로세스를 디버깅하는 과정을 보인다.

(그림 7)에서 SIGUSR1 시그널을 mgdb 라이브러리로부터 전달받은 mgdbserver는 새로운 gdbserver를 실행하고 gdbserver로 하여금 ptrace 시스템 콜의 PTRACE_ATTACH 인자를 사용하여 새로이 생성된 프로세스의 태스크 구조체에 있는 ptrace 변수에 PT_PTRACED 값을 설정하고 gdbserver가 새로이 생성된 프로세스의 부모 프로세스가 되도록 프로세스간의 관계를 변경한다. 또한 개발자는 호스트 시스템의 gdb를 사용하여 앞으로 디버깅을 진행할 프로세



(그림 7) mgdbserver가 SIGUSR1 시그널을 전달받은 후에 새로 생성된 프로세스를 디버깅하는 과정

스를 선택하면 mgdbserver는 선택한 프로세스의 부모 프로세스인 gdbserver에게 gdb로부터 전달받은 디버깅 요청 메시지를 전달하게 된다. 새로운 gdbserver를 생성하고 새로이 생성된 자식 프로세스와 gdbserver를 연결하는 작업을 수행한 후 개발자는 호스트의 gdb를 이용하여 (그림 2)에서 설명한 일반적인 디버깅 절차를 통하여 새로 생성된 프로세스를 디버깅할 수 있다.

4. 실험 및 평가

본 논문에서는 실험을 위해 부모 프로세스가 자식 프로세스를 fork 시스템 콜로 생성하고 부모-자식 프로세스간의 파이프를 통하여 서로 데이터를 주고받는 프로그램을 호스트 시스템의 gdb와 타겟 시스템의 mgdbserver를 사용하여 원격으로 동시에 디버깅하였다. 실험 환경은 다음과 같다. 호스트 시스템에는 한컴 Linux 2.2버전이 설치되어 있고 타겟 시스템은 와우 Linux 7.1이 설치되어 있다. 또한 gdb와 gdbserver는 gdb-5.2를 사용하였으며 mgdbserver와의 통신을 위해서 gdb와 gdbserver에 약간의 코드를 추가하였다.

본 실험에서는 디버깅 중인 프로세스가 새로운 프로세스를 생성한 경우 새로이 생성된 프로세스와 기존의 디버깅 중인 프로세스를 개발자가 호스트 시스템에서 하나의 gdb를 통하여 원하는 순간에 선택하여 변경해 나가면서 동시에 두 프로세스를 디버깅할 수 있도록 하는 방법에 초점을 맞추고 있으므로 (그림 8)의 부모 프로세스가 새로운 자식 프로세스를 생성하고 두 프로세스간 파이프를 이용하여 데이터를 주고받는 간단한 프로그램을 테스트 프로그램으로 사용하였다.

(그림 9)에서는 타겟 시스템에서 mgdbserver를 실행시키는 방법과 개발자가 호스트 시스템의 gdb와 타겟 시스템의 mgdbserver를 사용하여 본 논문에서 사용한 테스트 프로그램을 디버깅한 결과 중 타겟 시스템에서의 출력 결과를 보

인다. mgdbserver와 디버깅 프로그램은 타겟 시스템에서 수행되며 이들 프로그램의 실행 과정에서의 출력 결과물은 타겟 시스템의 mgdbserver의 제어터미널로 출력된다. 개발자는 타겟 시스템의 IP 주소와 포트 번호 그리고 디버깅하려는 프로그램의 실행 파일을 인자로 하여 mgdbserver를 실행시킨다. mgdbserver는 내부적으로 gdbserver를 실행시키고 gdbserver가 디버깅을 진행하려는 프로세스의 부모 프로세스가 되도록 한다. 이후 mgdbserver는 호스트 시스템에서 동작하는 gdb의 접속을 기다리게 된다. 호스트 시스템의 gdb로부터 접속이 정상적으로 이루어지면 타겟 시스템의 mgdbserver는 개발자가 gdb를 통하여 입력한 디버깅 작업을 디버깅하려는 프로세스의 부모 프로세스인 gdbserver에게 전달하여 gdbserver로 하여금 디버깅을 지원할 수 있도록 한다.

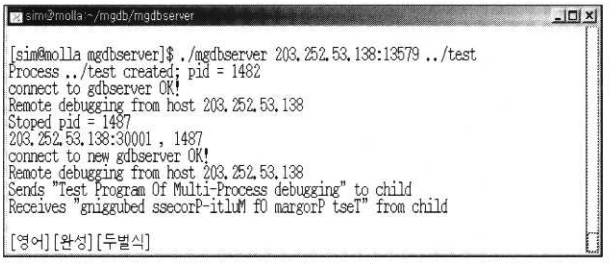
(그림 9)에서 현재 디버깅 중인 프로세스가 새로운 프로세스를 fork 시스템 콜을 통하여 생성하게 되면 현재 디버깅 중인 프로세스로부터 새로운 프로세스가 생성됨을 mgdb 라이브러리가 mgdbserver에게 알려주고 mgdbserver는 새로이 생성된 프로세스의 디버깅을 지원하기 위해서 새로운 gdbserver를 생성하고 새로이 생성된 프로세스와 연결한다. 이 작업에 대한 결과에 해당하는 부분이 (그림 9)에서 "connect to new gdbserver OK"에 해당한다. 개발자는 호스트 시스템의 "(gdb)" 명령행에서 "show-remote-debugee"를 입력하여 타겟 시스템에서 현재 디버깅 중인 프로세스가 fork 시스템 콜에 의해 생성한 모든 프로세스의 정보를 확인할 수 있다. 개발자는 마찬가지로 "(gdb)" 명령행에서 "change-remote-debugee pid 번호"를 입력하여 타겟 시스템의 특정 프로세스를 선택하여 원하는 프로세스로 디버깅을 진행할 수 있다.

본 실험에서 사용한 시나리오는 다음과 같다. 부모 프로세스는 "Test Program Of Multi-Process debugging"라는 문자열을 파이프를 통하여 자식 프로세스에게 전송한다. 그리고 부모 프로세스는 Sends "Test Program Of Multi-Process debugging" to parent라는 문자열을 출력한다. 개발자는 호스트 시스템의 gdb를 사용하여 자식 프로세스로 디버깅을 진행하려는 프로세스를 변경하고(change-remote-debugee 사용) 자식 프로세스를 실행시킨다. 자식 프로세스는 부모 프로세스로부터 받은 "Test Program Of Multi-Process debugging"라는 문자열을 역순으로 변형한 뒤 다시 파이프를 통하여 부모 프로세스에게 전송한다. 개발자는 부모 프로세스가 자식 프로세스로부터 받은 문자열을 확인하도록 하기 위해서 다시 디버깅을 진행하려는 프로세스를 부모 프로세스로 변경한다. 그리고 파이프를 통해서 자식 프

```
# include <stdio.h>

void reverse_string (char *buf)
{
                    char *tmp ;
                    int i ;

                    tmp = (char*) malloc (1024) ;
                    strcpy (tmp, buf) ;

                    for (i = strlen (buf)-1 ; i >= 0 ; i--)
                              buf [strlen (buf) - i-1] = tmp [i]

                    buf [strlen (tmp)] = 0 ;

                    free (tmp) ;
                    return ;

}
int main ()
{
      int pid ;
      int pfd [2] ;
      int cfd [2] ;
      int ret ;
      char buf [1024]
      int cnt ;

      ret = pipe (pfd) ;

      if (ret < 0)
      {
            printf ("Can't make a pipe \h") ;
      }

      ret = pipe (cfd) ;

      if (ret < 0)
      {
            print ("Can't make a pipe\n") ;
      }

      pid = fork () ;

      if (pid > 0)
      {
            close (pfd [1]) ;
            close (cfd [0]) ;

            strcpy (buf, "Test Program Of Multi-Process debugging") ;

            write (cfd [1], buf, strlen (buf)) ;
            printf ("Sends \%s\ to child\n", buf) ;

            memset (buf, 0, 1024) ;
            read (pfd [0], buf, 1024) ;
            print ("Receives \%s\ from child\n", buf) ;

      }
      else if (pid == 0)
      {
            close (pfd [0]) ;
            close (cfd [1]) ;

            memset (buf, 0, 1024) ;
            read (cfd [0], buf, 1024) ;
            reverse_string (buf) ;
            write (pfd [1], buf, strlen (buf)) ;

      }
      else
            printf ("Can't fork()\n") ;
return 0 ;
```

(그림 8) 테스트 프로그램

로세스가 전송한 문자열을 읽어 들인다. 부모 프로세스는 자식 프로세스로부터 전송 받은 문자열을 출력한다. 이에 해당하는 결과가 Receives "gniggubed ssecorP-itluM fO mar gorP tseT" from child에 해당한다.

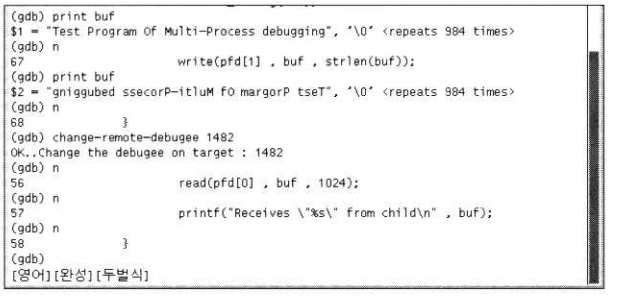


(그림 9) 타겟 시스템에서의 mgdbserver의 수행 결과

(그림 10)과 (그림 11)에서는 호스트 시스템에서 gdb를 실행시키는 방법과 개발자가 호스트 시스템의 gdb와 타겟 시스템의 mgdbserver를 사용하여 본 논문에서 사용한 테스트 프로그램을 디버깅한 결과 중 호스트 시스템에서의 결과를 보인다. 개발자는 타겟 시스템에서 mgdbserver를 실행시킬 때 사용한 인자들을 사용하여 호스트 시스템의 gdb가 타겟 시스템의 mgdbserver에 소켓 연결이 되도록 한다. 본 실험에서는 부모 프로세스가 자식 프로세스에게 문자열을 전송할 때와 자식 프로세스가 부모 프로세스에게 역순으로 변형된 문자열을 전송하려 할 때 정지점을 설정하였다. 개발자는 부모 프로세스가 파이프를 통하여 자식 프로세스에게 전송한 데이터의 확인을 위해서 "(gdb)" 명령 행에서 "change-remote-debugee 1487"를 입력하여 자식 프로세스로 디버깅 대상 프로세스를 변경한다. 개발자는 자식 프로세스의 디버깅을 진행한 후 다시 "(gdb)" 명령 행에서 "change-remote-debugee 1482"를 입력하여 부모 프로세스로 디버깅 대상 프로세스를 변경한다. 개발자는 부모 프로세스 디버깅을 진행하면서 자식 프로세스로부터 전달받은 역순서의 문자열을 확인할 수 있다. 이와 같은 방식으로 개발자는 디버깅 대상 프로세스를 변경해 가면서 다중 프로세스 디버깅을 진행할 수 있다.

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

(그림 10) 호스트 시스템에서의 gdb의 실행 결과(앞부분)



(그림 11) 호스트 시스템에서의 gdb의 실행 결과(뒷부분)

TotalView 버전 5.0.0-4를 사용하여 성능 비교를 하였으며 성능 비교를 위해 사용한 프로그램은 (그림 8)의 테스트 프로그램이다. <표 1>을 통해서 알 수 있듯이 본 논문에서 제안하는 mgdb 라이브러리와 mgdbserver를 사용한 원격

	ETNUS 사의 TotalView	mgdb와 mgdbserver를 이용한 방법
테스트 프로그램의 이미지 크기	53658Bytes	23973 Bytes(mgdb : 7173Bytes, mgdbserver : 24592Bytes)
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
파이프(pipe)를 사용하여 통신을 하는 두 프로세스간의 동시 디버깅 지원 여부	로컬에서만 가능	로컬 및 원격에서 지원 가능

〈표 1〉 TotalView 프로그램과 mgdb 라이브러리와 mgdbserver를 이용한 방법의 비교 테이블

디버깅 방법이 비교적 부족한 자원을 갖는 임베디드 시스템에서 동작하는 다중 프로세스 디버깅 도구로 더 적합하다. 또한 본 논문에서 제안하는 gdb와 mgdbserver를 로컬에서 모두 실행시키면 원격에서 뿐만 아니라 로컬에서도 다중 프로세스들을 동시에 디버깅할 수 있다.

5. 결 과

본 연구에서는 커널의 변경 없이 라이브러리 래핑 방법을 통하여 원격으로 디버깅 중인 프로세스가 새로운 프로세스를 생성한 경우 기존의 디버깅 중인 프로세스는 물론 새로이 생성된 프로세스들도 원격으로 동시에 디버깅할 수 있도록 지원하는 mgdb 라이브러리와 mgdbserver를 설계 및 구현 하였다. mgdb 라이브러리를 통해 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. 또한 mgdbserver를 통하여 원격으로 디버깅을 진행하는 개발자에게 기존의 방법보다 편리한 디버깅 환경을 지원할 수 있다.

mgdb 라이브러리와 mgdbserver를 이용한 디버깅 방법은 디버깅 중에 fork 시스템 콜에 의해 새로운 프로세스가 생성되는 경우 기존의 디버깅 중인 프로세스뿐만 아니라 새로이 생성된 프로세스도 디버깅할 수 있다. 또한 개발자가 원하는 순간에 임의의 프로세스를 번갈아 가면서 다중 프로세스들을 동시에 디버깅할 수 있다. 그리고 기존의 방법과는 달리 개발자는 호스트 시스템의 하나의 gdb만을 통하여 타겟 시스템의 모든 디버깅 진행 중인 프로세스를 접근할 수 있으므로 개발자는 쉽고 효율적인 디버깅을 진행할 수 있다.

본 연구에서 개발한 mgdb 라이브러리와 mgdbserver를 사용하여 파이프를 통해 서로 데이터를 주고받는 두 프로세스들을 동시에 원격으로 디버깅하는 실험을 실시하여 원격지에서 다중 프로세스들을 동시에 디버깅할 수 있는 것을 확인 하였다. 또한 실험을 통하여 기존의 방법과는 달리 개발자는 호스트 시스템에서 하나의 gdb만으로도 다중 프로세스들을 동시에 디버깅할 수 있어 기존의 방법보다 효율적인 디버깅을 진행할 수 있는 것을 확인 하였다.

향후 연구과제로는 개발자에게 좀더 쉬운 디버깅 환경을 제공하기 위해서 GUI(Graphic User Interface)를 개발하는 것이며. 원격 디버깅시 타겟 시스템에서 실행된 결과는 타겟 시스템에 연결된 터미널을 통해서만 확인할 수 있는데 효율적인 원격 디버깅을 위해서는 타겟 시스템에서 실행한 결과를 호스트 시스템에서 확인할 수 있도록 하는 기능이 추가되어야 할 것이다.

참 고 문 헌

[1] Daniel Jacobowitz, Remote Debugging with GDB, http://www.kegel.com/linux/gdbserver.html, 2002.

[2] Minheng Tan, A minimal GDB stub for embedded remote debugging, http://www1.cs.columbia.edu/~sedwards/classes/2002/w4995-02/tan-final.pdf, 2002.

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

[4] 임형택, 심현철, 손승우, 김홍남, 김채규, "Q+P Esto의 원격 개발을 지원하는 타겟에이전트", 한국정보처리학회 2001년 추계학술대회, 제8권 제2호, pp.671-674, 2001.

[5] Greg Rose, Embedded Linux 101, http://www.ecnmag.com/ecnmag/issues/2001/12012001/ec1dsc100_.asp, 2001.

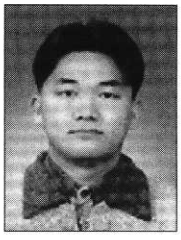
[6] Etnus, Totalview Getting Started, http://www.etnus.com/Products/TotalView/started/getting_started2.html, 2001.

[7] Intel, Intel Atchitecture Software Developer's Manual, Vol. 3, 1999.

[8] Daniel P. Bovet and Marco Cesati, Understanding the Linux Kernel, O'Reilly, 2001.

[9] Uresh Vahalia, Unix Internals, Prentice Hall, 1996.

[10] Sun Microsystems Inc., Linker & Libraries Guide, October, 1998.



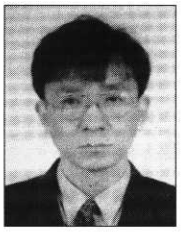
심 현 철

e-mail : jlmaj@ece.skku.ac.kr

2000년 성균관대학교 정보통신공학부 학사

2003년 성균관대학교 정보통신공학부 석사

관심분야 : Embedded Linux, 시스템 소프트웨어, 운영체제



강 용 혁

e-mail : yhkang1@ece.skku.ac.kr

1996년 성균관대학교 정보공학과 학사

1998년 성균관대학교 정보공학과 컴퓨터공학전공 석사

2003년 성균관대학교 정보통신공학부 박사

현재 극동대학 전자상거래과 교수

관심분야 : 분산 시스템, 이동 컴퓨팅 시스템, MANET

엄 영 익

e-mail : yieom@ece.skku.ac.kr

1983년 서울대학교 계산통계학과 학사

1985년 서울대학교 대학원 전산과학전공 석사

1991년 서울대학교 대학원 전산과학전공 박사

2000년~2001년 Dept. of Info. and Comm. Science at UCI 방문교수

현재 성균관대학교 정보통신공학부 교수

관심분야 : 분산 시스템, 이동 컴퓨팅 시스템, 이동 에이전트, 시스템 소프트웨어