□ 기술해설 □

Unix 커널 디버거

중앙대학교 박상서* · 김성조**

목 차

1. 서 론

에러는 프로그래머가 작성한 프로그램과 실제로 프로그램이 실행되는 과정에서 발생하는 차이이고 버그(bug)는 에러를 발생시키는 원인이다[13]. 일반적으로, 에러는 프로그램 개발자가 자신의 프로그램을 테스트하는 과정 또는 일반 사용자가 프로그램을 사용하는 과정에서 발견되며 디버깅에 의하여 수정된다. 즉, 디버깅은 에러가 완전히 제거될 때까지 계속 반복하면서 에러의 위치를 찾고 그 원인을 분석하여 수정하는 과정이다[13,16,36,44].

대부분의 프로그래머는 프로그램 개발시 많은 시간을 디버깅에 소모한다. 즉, 에러가 발생하거나 프로그래머가 원하는 결과가 출력되지 않으면 이의 원인 파악에 많은 시간을 투입하게 된다. 간단한 프로그램일 경우에는 입력 데이타 또는 에러로 인한 수행 결과(errornous output)만으로도 디버깅이 가능하지만, 커널과 같이 복잡한 프로그램의 경우에는 이것만으로는 정확한 디버깅이 어렵다. 일반적으로, 운영 체제 개발시 에러를 해결할 수 있는 디버깅 능력을 보유하게 되어야 비로소 운영 체제의 개발이 완료되었다고 할 수 있으므로 커널 디버거는 운영 체제의 개발과 밀접한 관계를 갖고 있다.

응용 프로그램 개발을 위해 주로 이용되는 사용자 레벨 디버거는 adb[11,12], sdb[53], dbx[10] 및 gdb[26]와 같이 여러 가지 종류의 디버거가 이용 가능한 반면, Unix 커널의 개발에 반드시 필요한 커널 디버거는 종류가 다양하지 않다. 뿐만 아니라, 커널의 디버깅에 이용할 수 있는 디버깅 방법에는 정적 분석 기법[4,5,8,24,37,44], 매크로를 이용하는 방법[9,13], 사용자 레벨 디버거를 이용하는 방법[9,14,28,33,42,43], 그리고 crash 유틸리티를 이용하는 방법[39,47] 등이 있으나 이들은 모두 커널 디버깅에는 적합하지 않다[1,2]. 정적 분석 기법은 시스템에 독립적일 뿐 아니라 원시 코드 수준에서의 교착상태, livelock 및 무한 대기와 같은 에러를 쉽게 발견할 수 있다. 그러나 분석해야 할 데이타의

*준회원
**종신회원

양이 많고, 최악의 경우에는 모든 가능한 제어의 흐름을 고려해야 한다. 또한, 포인터나 첨자와 같은 동적 오브젝트, 프로세스의 생성과 종료 또는 프로세스 간의 통신과 같은 동적 특성을 분석할 수 없다. 매크로 기법을 이용하여 디버깅 코드를 삽입하면 프로그래머가 지정한 특정 에러를 편리하게 발견할 수 있다. 그러나 원시 프로그램이 수정될 때마다 디버깅 코드가 추가되거나 수정되어야 하고, 많은 관련 모듈들이 재컴파일되어야 하는 문제가 있다. 사용자 레벨 디버거는 single step, breakpoint, 변수 조사 기능과 다중프로세스를 추적할 수 있는 기능도 제공한다. 그러나 이들 디버거를 이용할 경우 동작중인 커널을 정지시키기 어려울 뿐더러, 디버깅이 완료된 뒤 정지해 있던 커널 프로세스와 사용자 프로세스를 다시 시작시킬 수 있는 방법도 없다. 4.3BSD의 개발에서는 커널이 더 이상 진행될 수 없는 상황이 발생했을 때 디스크에 저장했던 메모리의 복사본을 참조할 수 있도록 지원하는 crash 유틸리티가 사용되었다. Crash 유틸리티는 커널의 디버깅에 사용되기는 하지만 디스크에 저장했던 메모리 내용만을 참조할 수 있을 뿐이며, 일반적으로 디버거가 제공해야 할 breakpoint나 single step 등의 기능을 지원하지 않는다.

더구나, 주전산기나 SVR4.2/MP와 같은 다중프로세서 운영 체제(multiprocessor operating system) 커널은 병행 프로그램이기 때문에 기존의 단일 프로세서 커널용 디버거를 사용할 수 없을 뿐만 아니라[46] 비결정성(non-determinacy)[16,23,24], 탐사 효과(probe effect)[24,28,29,30,44], 비재생산성(non-reproducibility)[24] 등의 문제를 내포하고 있어 디버깅이 매우 어렵다.

이와 같은 문제를 종합적으로 해결할 수 있는 방법은 커널 디버깅의 특성을 최대한 반영한 전용 디버거를 사용하는 것이다. 본 고에서는 Unix 운영 체제의 개발 및 안정화에 효율적으로 이용될 수 있는 Unix 커널 디버거에 대하여 소개한다. 먼저, 2장에서는 커널 디버거가 갖추어야 할 요건을 설명하고, 3장에서는 커널의 디버깅에 이용될 수 있는 디버거 모델에 대하여 기술한다. 4장에서는 커널 디버거가 갖추어야 할 기능에 대하여 살펴보고, 5장에서는 커널 디버거의 예로서 SVR4의 kdb와 주전산기II의 커널 디버거에 대하여 간략히 소개한다. 마지막으로, 6장에서는 본 고의 결론을 기술한다.

2. 커널 디버거의 요건

커널 디버거가 갖추어야 할 기본적인 요건을 요약하면 다음과 같다.

- 항상 정확한 상태 정보 제공
- 디버깅으로 인한 탐사 효과와 성능 저하의 최소화
- 다양한 하드웨어 플랫폼으로의 이식성
- 사용의 편이성
- 디버깅 정보의 효율적 출력
- 다양한 기능의 제공을 통한 에러의 신속한 발견

이와 같은 커널 디버거의 요건들(requirements)은 커널 디버거 모델의 선정 과정에서부터 설계와 구현에 이르는 전 과정에서 항상 고려되어야 할 사항이다. 본 장에서는 커널 디버거가 갖추어야 할 요건들은 일반적 요건, 구조적 요건 및 기능적 요건으로 구분하여 설명한다.

2.1 일반적 요건

2.1.1 정확성

정확성은 커널 디버거가 갖추어야 하는 가장 중요한 요건으로서 커널 디버거가 커널의 상태를 정확히 검색할 수 있어야 함을 의미한다. 즉, 커널 디버거는 사용자가 원하는 시점에서의 커널 상태 정보를 정확하게 제공할 수 있어야 한다는 것이다. 사용자는 디버거가 제공하는 정보를 근거로 개발자가 의도한 대로 커널이 정확히 실행되고 있는지 판단하게 되므로, 정확히 동작하는 커널이 비정상적으로 동작하는 것처럼 보여지거나 비정상적으로 동작하는 커널이 마치 정상적인 상태인 것처럼 보여지지 않아야 한다.

2.1.2 탐사 효과 최소화

탐사 효과는 병행 프로그램을 디버깅하거나 모니터할 때 시간지연(delay)으로 인하여 병행 프로그램의 정확성에 영향을 주는 현상이다. 즉, 디버거의 실행시 발생하는 시간지연으로 인하여 비정상적으로 동작하던 커널이 정상적으로 동작하게 되거나 그 반대의 경우가 발생하는 현상이다. 현재까지는 탐사 효과의 완벽한 제거가 매우 어려운 것으로 알려지고 있기 때문에 디버깅시 다양한 측면에서 탐사 효과를 최소화시키기 위한 고려가 수반되어야 한다[2,44].

2.1.3 성능 저하 최소화

커널이 실행되는 동안 디버거에서 필요로 하는 정보를 로그에 저장하거나 모니터하는 과정은 시스템의 전체적인 성능 저하를 일으킬 수 있다. 특히, 관련 정보를 디버깅되는 시스템의 하드디스크에 저장할 경우 관련 정보의 저장을 위한 입출력 루틴에는 집중적으로 제어가 옮겨지게 되므로 성능 저하의 주된 요인이 될 수 있다. 더구나, 이 로깅 지점에서는 탐사 효과가 발생할 수도 있어 결과적으로 정확하지 않은 정보가 수집될 수도 있다. 따라서, 정보를 로그에 저장하거나 모니터하는 동안 커널의 성능 저하를 최소화할 수 있어야 한다[35].

2.1.4 이식성

일반적으로 커널 디버거는 커널의 일부분으로 포함되기 때문에 디버거를 이식하기 위해서는 항상 커널 원시 코드의 일부가 수정되어야 하지만 개발되는 디버거의 이식성을 높이기 위해서는 커널 원시 코드의 수정을 가능한 한 최소화하여야 한다[1,2,6,9,35,46,48]. 또한, 최소한의 디버거 원시 코드 변경만으로 하드웨어 구조와 운영 체제 버젼이 유사한 다양한 커널에 이식될 수 있어야 한다. 이를 위하여 대부분의 디버거 원시 코드를 상위 레벨 언어로 구현하고 시스템에 의존적인 부분만을 어셈블리 언어로 구현하는 것이 바람직하다. 특히, 디버거 원시 코드 중 커널 자료구조를 검색하는 루틴들은 시스템 헤더 화일이 변경될 때마다 재작성되어야 하므로 헤더 화일로부터 검색 코드들을 자동으로 생성하는 코드 생성 도구와 이식 도구가 함께 개발되어야 한다.

2.1.5 디버깅 정보의 가용성

커널의 초기 개발 단계에서는 시스템의 동작 또는 시험 도중 시스템이 비정상적으로 동작할 확률이 매우 높다. 이 때 화일 시스템에 이상이 발생하여 디버거에서 수집한 디버깅 정보들을 전혀 사용할 수 없는 경우가 발생하지 않도록 디버깅 정보를 안정된(stable) 저장 장치에 저장하여야 한다[46]. 또한, 차후 통계 정보 추출 또는 재실행에 이용될 수 있도록 커널의 순간상태(snapshot)를 내력(history)별로 저장할 수도 있어야 한다[44].

2.1.6 사용 편의성

디버거의 사용 편의성을 높이기 위해서는 사용자 명령어와 사용자 인터페이스가 잘 정의되어야 한다[35]. 디버깅을 위한 사용자 명령어는 논리적이면서도 명백, 간단해야 하고 함축적이어야 한다[2,6,49]. 이를 위해서는 관련있는 기능들을 그룹화하여 대표 명령어로 참조할 수 있도록 하고, 옵션을 통하여 세부 기능들이 실행되도록 명령어가 계층 구조로 정의되어야 한다. 또한, 명령어의 융통성과 확장성을 제공하기 위하여 디폴트를 제공하여야 하며 가능한 한 특수기호의 사용을 줄여야 한다. 이와 함께, 윈도우 시스템을 이용한 고기능 사용자 인터페이스와 디버거 각 기능의 사용법에 대한 도움말을 제공하여야 한다.

2.2 구조적 요건

2.2.1 커널의 정지 및 재개

Adb와 같은 일반적인 사용자 레벨 디버거는 디버깅 대상 프로그램을 자신의 서브프로세스로 실행시키기 때문에 자유자재로 대상 프로그램을 정지시키거나 재개(resume)시킬 수 있다. 마찬가지로 커널 디버거도 커널을 임의의 위치에서 정지시킬 수 있어야 하며 디버깅이 완료된 후 정지된 위치에서 커널의 실행을 재개시킬 수 있어야 한다. 따라서, 커널 디버거는 기본적으로 동작 중인 커널을 임의로 정지시키거나 재개시킬 수 있는 구조를 가져야 한다[9,46]. 특히, 앞 절에서 기술한 정확성 요건을 만족시키기 위해서는

디버깅의 시작과 종료시 데이타의 손실이 없어야 하고, 커널이 정지되는 시점에서 모든 프로세스와 클럭도 정지되어야 한다. 또한, 디버깅이 완료되어 실행 제어가 커널로 옮겨지면 그 지점부터 이상없이 실행이 재개될 수 있어야 한다[6, 46].

2.2.2 대상 시스템의 하드웨어적 특성 반영

커널은 하드웨어와 매우 밀접한 연관을 가지기 때문에 커널을 정확하게 디버깅하기 위해서는 커널 디버거의 구조에 대상 시스템의 하드웨어적 특성이 반영되어야 한다[6]. 즉, 펌웨어의 이용 가능 여부, 프로세서의 갯수, 메모리의 크기, 그리고 하드웨어를 구성하는 시스템 요소들의 연결 방법(예: bus) 등에 따라 디버거의 구조가 정의되어야 한다. 또한, 레지스터의 종류와 포맷, 커널 디버깅에 사용할 수 있는 어셈블리 명령어, 예외상태(exception)의 종류와 발생 방법 등의 특성을 고려하여 디버거의 하드웨어 인터페이스를 설계하여야 한다.

2.2.3 대상 시스템의 소프트웨어적 특성 반영

디버거 구조에는 커널의 소프트웨어적 특성도 반영되어야 한다[6]. 즉, 시스템 콜과 트랩의 종류, 주요 자료 구조 및 프로세스 사이의 통신 방법 등을 디버거 구조에 반영하고 디버거에서 이용할 수 있는 윈도우 시스템을 고려하여 사용자 인터페이스를 설계하여야 한다.

2.3 기능적 요건

2.3.1 디버깅 정보 표현

디버깅 정보는 기본적으로 심볼화된(symbolic) 형태로 출력되어야 하며[35,46,49]. 텍스트뿐만 아니라 도표, 애니메이션, 다중 윈도우 등 다양한 형태로 출력되어야 한다[44]. 그리고 다중프로세서에 관련된 정보들은 병행성이 잘 표현되어야 한다. 또한, 출력되는 디버깅 정보에는 메모리의 주소화(addressing) 방법 또는 프로세서의 종류에 따라 범용 레지스터뿐만 아니라 수퍼 유저(super user) 모드의 레지스터, 파이프라인과 명령어 선입수 큐(prefetch queue) 등이 추가되어야 한다.

2.3.2 다양한 디버깅 훅 제공

커널 디버거는 특정 명령어가 실행될 때(instruction breakpoint)와 매번 명령어가 실행될 때(single step), 그리고 그밖의 여러 가지 상황이 발생하는 경우에 대해서도 디버거로 트랩(trap)할 수 있도록 다양한 형태의 디버깅 훅(hook)을 제공하여야 한다. 즉, 커널의 실행 도중 특정 조건이 만족될 때(conditional breakpoint), 특정 데이타가 사용(read/write)될 때(data breakpoint), 입출력이 발생할 때(I/O breakpoint), 특정 메모리 영역이 참조될 때 (range breakpoint), 특정 서브루틴이 호출될 때(subroutine breakpoint), 또는 분기가 발생할 때(branch trace)에도 각각 디버거로 트랩할 수 있어야 한다. 또한, 디버거로 트랩할 때 사용자가 작성한 프로그램을 동적으로 링크시켜 실행(scan-point)시킬 수 있도록 함으로써 보다 융통성있는 디버깅 훅을 제공하여야 한다[2,14,15].

2.3.3 시스템 제어

커널 디버거는 시스템을 전체적으로 제어할 수 있어야 한다[6,24]. 즉, 클럭을 포함하여 특정 또는 모든 프로세서와 프로세스의 실행을 정지시키거나 재개시킬 수 있어야 하며 그 당시의 상태를 조사할 수 있어야 한다. 또한, 프로세스의 강제 종료와 우선순위 변경도 가능해야 한다. 뿐만 아니라 프로세서의 특성에 따라 명령어, 데이타 또는 주소 번역 캐쉬의 캐싱을 가능하게 하거나 불가능하게 할 수도 있어야 한다.

2.3.4 상위 레벨 기능 지원

내력별로 저장된 디버깅 정보가 정확하다면 재실행(replay)을 통하여 이전의 실행 결과와 동일한 결과를 얻을 수 있다[32.40,44]. 따라서 디버거는 상위 레벨 기능으로 병행 프로그램의 재실행을 지원함으로써 비재생산성을 해결할 수 있어야 한다. 또한, 이 정보들을 바탕으로 관련있는 정보들의 변이(transition) 과정을 추적하거나 실행 과정을 시각화할 수 있어야 하며[38.41, 51,52], 순간 상태 정보들로부터 통계 정보를

생성할 수도 있어야 한다. 마지막으로, 시스템의 정상적인 동작 여부뿐만 아니라 시스템의 성능을 감시하거나[41,45,49] 구현된 알고리즘의 효율성을 측정할 수도 있어야 한다[35].

3. 디버거 모델

디버거의 모델은 컴퓨터 시스템의 구조, 대상 프로그램과의 상호 작용 방법과 디버거가 실행되는 위치에 따라 일반적인 디버거, 대상 프로그램의 일부분으로서 실행되는 디버거, 원격 디버거, 분산 디버거, 부착 디버거 및 이들을 혼합한 혼합형 디버거로 분류할 수 있다. 본 장에서는 여러 가지 형태의 디버거 모델들을 비교·분석하고, 커널의 디버깅에 적합한 디버거 모델에 대하여 알아본다.

3.1 일반적인 디버거

Agora[25], CBUG[28], Multibug[21], adb[11,12], cdbg[34], dbxtool[10], gdb[26], mt-dbx[31], 또는 pdbx[50]와 같은 일반적인 사용자 레벨 디버거는 그림 1과 같이 동일한 메모리 내에서 디버깅되는 대상 프로그램을 자신의 서브프로세스로 실행시키면서 디버깅한다.

이 디버거들은 single step, breakpoint, 변수 조사 등의 기본 기능 이외에도 그래픽을 이용한 이벤트 추적, 프로세스의 재실행, 이벤트 도표화 또는 순서화(ordering), 다중프로세스 추적 등 다양하고 강력한 디버깅 기능들을 제공한다. 뿐만 아니라 대부분이 그래픽을 기반으로 하는 사용자 인터페이스를 제공하므로 사용이 편리하며 사용자가 이해하기 쉬운 형태로 디버깅 정보를 표현한다.

그러나 이 모델을 커널의 디버깅에 적용할 경우 동작 중인 커널의 정지 또는 정지해 있던 커널의 재개가 매우 어려워 2.2.1절에서 기술한 커널의 정지 및 재개 요건을 만족하지 못한다. 더구나 커널을 자신의 서브프로세스로 실행시킬 경우 두 개의 커널이 동시에 존재하게 되어 시스템이 정상적으로 동작할 수도 없다.

3.2 대상 프로그램의 일부분으로 실행되는 디버거

일반적인 디버거 모델을 사용할 수 없는 경우 그림 2와 같이 디버거를 디버깅되는 대상 프로그램의 일부분으로 포함시키기도 한다. 이 디버거 모델의 가장 대표적인 예는 커널의 디버깅에 사용되는 특수 목적 디버거인 kdb[9,27,48]이다. 일반적인 디버거 모델에서는 디버깅되는 프로그램이 디버거의 서브프로세스로 실행되다가 breakpoint에 걸리거나(hit) single step이 끝나게 되면 실행 제어를 디버거로 반환한다. 그 반면 kdb는 커널의 서브루틴으로서 실행된다. 즉, 커널이 실행되는 도중 커널 내에 설정된 breakpoint에 걸리게 되면 트랩 루틴을 거쳐 커널의 서브루틴으로 실행된다. 이 때 트랩 루틴은 클럭과 모든 프로세스를 중지시킨 뒤 kdb를 호출하므로 데이타의 손실없이 커널이 정지되었다가 재개될 수 있다.

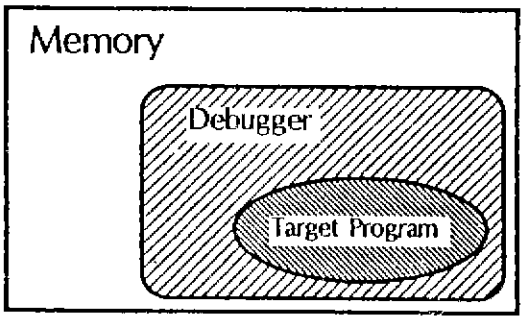

그림 1 일반적인 디버거 모델

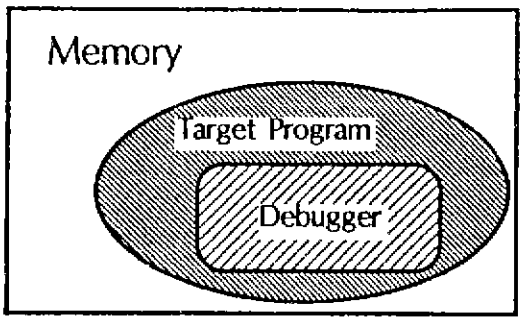

그림 2 대상 프로그램의 일부분으로 실행되는 디버거 모델

이 모델을 적용하기 위해서는 반드시 커널 원시 코드가 수정되어야 하며, 커널 헤더 화일 또는 트랩 루틴 등이 변경될 때마다 디버거 원시 코드가 수정되어야 한다는 단점이 있다. 또한, 가능한 한 커널의 서비스를 받지 않아야 하고 커널 중 breakpoint를 걸 수 없는 지점(예: 콘솔 입출력 루틴)이 존재한다는 제약 조건이 있다. 뿐만 아니라 그래픽 사용자 인터페이스를 기반으로 하는 상위 레벨 기능을 제공하기 어렵고, 디버깅 정보를 로그에 저장하거나 실행 과정을 모니터할 때 탐사 효과가 발생할 확률이 높다.

그 반면, 커널 프로세스와 사용자 프로세스를 자유자재로 정지시키거나 재개시킬 수 있고, 다양한 커널 자료 구조, 시스템 메모리, 수퍼 유저 모드 레지스터의 내용 등을 임의로 참조하거나 변경할 수 있다. 또한, 대상 시스템의 하드웨어 및 소프트웨어적인 특성을 잘 반영할 수 있으며 시스템 전반에 대한 제어 능력이 뛰어나, 대부분의 단일프로세서 커널 디버거는 이 모델을 채택하고 있다.

3.3 원격 디버거

앞에서 기술한 두 가지 디버거 모델은 동일한 시스템 내에서 대상 프로그램과 메모리를 공유하며 실행되지만 pdb[43], EBBA[18] 또는 ctkdb[27]와 같은 원격(remote) 디버거는 대상 프로그램과 물리적으로 분리된 위치에서 즉, 원격으로 실행된다. 원격 디버거는 디버거의 상주 위치에 따라 지역(local) 시스템의 ROM에서 실행되는 ROM 디버거(그림 3-a 참조)와 원격 시스템의 메모리에서 실행되는 원격 디버거(그림 3-b 참조)로 분류할 수 있다. 이들 디버거는 각각 시스템 버스 또는 네트워크를 통하여 디버깅 명령을 대상 프로그램에 전달하거나 디버깅 결과를 반환받는다.

이 모델을 커널의 디버깅에 사용하면 커널의 실행을 모니터하기 쉽고, 커널 상태를 객관적으로 파악할 수 있다. EDL(Event Definition Language)[17]을 사용하는 경우에는 커널 이벤트를 쉽게 표현할 수도 있다. 특히, ROM 디버거는 디버깅되는 시스템에 탑재되기 때문에 동작 중인

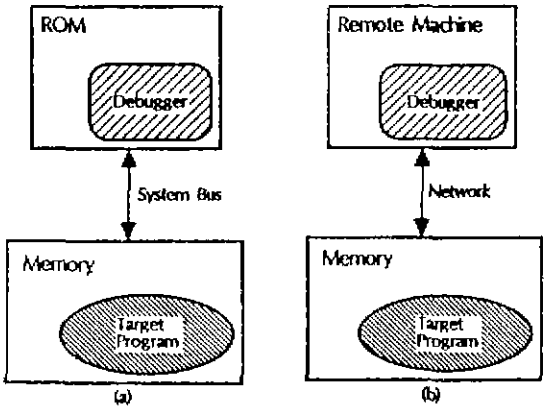

그림 3 원격 디버거 모델

커널에 대한 정보 수집과 시스템의 전체적인 제어가 매우 수월할 뿐만 아니라, 디버거 내에 대상 하드웨어의 특성을 반영하기도 용이하다.

그러나 커널과의 상호작용을 위하여 많은 양의 정보가 시스템 버스 또는 네트워크를 통해 전송되어야 하며, 시간이 오래 경과된 정보가 수집될 경우 그 정보의 가용성이 떨어질 수 있고, 커널에서 발생하는 이벤트에 대한 즉각적인 개입(intervention)이 쉽지 않다는 문제가 있다. 특히 그림 3-b의 경우, 정보를 전송하거나 수신하기 위한 프로토콜을 가지고 있어야 하며, 디버깅에 필요한 시간보다 통신 시간이 너무 길 경우에는 정확한 상태 수집이 어려울 수도 있다.

3.4 분산 디버거

Pi[19,20] 또는 Pilgrim[22]과 같은 분산 디버거는 그림 4와 같이 분산 시스템을 구성하는 각 프로세서에 동일한 능력을 갖는 지역 디버거를 설치하고 하나의 주(master) 디버거가 비동기화된 각 프로세서의 지역 디버거로부터 전송되는 디버깅 정보를 수집하여 관리한다. 원격 디버거는 디버거가 원격 시스템에만 위치하면서 디버깅 훅 또는 시스템 콜 등을 이용하여 디버깅 정보를 수집하지만, 분산 디버거는 지역 디버거를 각 프로세서에 설치하여 디버깅 정보를 수집한다[24].

이 모델은 각 지역 디버거의 자치성(autonomy)

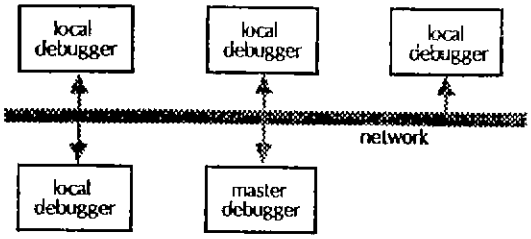

그림 4 분산 디버거 모델

이 향상될수록 보다 강력한 디버깅 능력을 발휘할 수 있으며[24], 확장성(scalability)이 뛰어나고 시스템을 모니터하기 용이하다는 장점이 있다. 또한, 주 디버거는 각 지역 디버거에서 전송하는 결과만을 취합하기 때문에 주 디버거의 작업량을 여러 지역 디버거로 분산시킬 수 있어 디버깅 비용을 최소화할 수 있고, 그룹 또는 모든 지역 디버거에 디버깅 명령을 지시할 수도 있어 디버깅 효율이 증가될 수 있다.

그러나 시스템의 크기가 증가함에 따라 통신 시간 지연과 네트워크 오버헤드 증가 문제가 발생하며, 지역 디버거의 자치성이 향상됨에 따라 주 디버거가 지역 디버거를 통제하거나 디버깅 정보를 관리하기 어려워진다. 또한, 주 디버거로 전달되는 정보가 많을수록 이 지점에서 병목현상이 발생하여 시스템의 전체적인 성능이 저하될 수도 있다.

3.5 부착 디버거

부착(attached) 디버거는 병행 프로그램의 디버깅을 위하여 제안된 새로운 모델로서 Parasight[14]에서 지원한다. 그림 5와 같은 구조를 갖는 Parasight는 디버깅을 위한 모니터 프로그램인 parasite를 대상 프로그램에 동적으로 링크시켜 실행시킴으로써 그 실행을 관찰하거나 제어한다. 이를 위하여 Parasight는 내부적으로 별도의 동적 로더를 가지고 있어서 대상 프로그램을 로드하거나 링크시킬 수 있으며, 필요시 parasite를 대상 프로그램에 링크시킬 때 사용할 심볼 테이블을 메모리에 상주시킬 수 있다.

이 모델을 사용하면 디버거가 공유 메모리를 이용하여 대상 프로그램을 적은 비용으로 투명

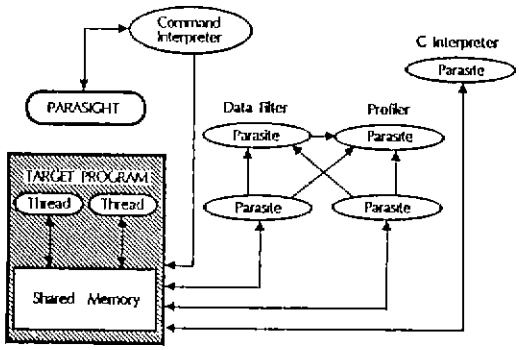

그림 5 부착 디버거 모델

(transparent)하게 관리하고 감시할 수 있지만, 이 모델을 지원하기 위한 프로그래밍 환경이 우선적으로 제공되어야 하므로 기존 환경에 직접 적용하기 어렵다.

3.6 혼합형 디버거

혼합형 디버거[3.46]는 앞에서 기술한 디버거 모델을 둘 이상 혼합한 모델로 주전산기를 위한 커널 디버거를 예로 들 수 있다. 그림 6에서 볼 수 있는 바와 같이 이 디버거는 대상 프로그램의 일부분으로 동작하는 디버거 모델과 ROM 디버거 모델 그리고 분산 디버거 모델을 혼합하였으며, 부착 디버거의 scan-point 기능을 사용하고 있다[1.2].

이 디버거는 공유 메모리에서 커널의 일부분

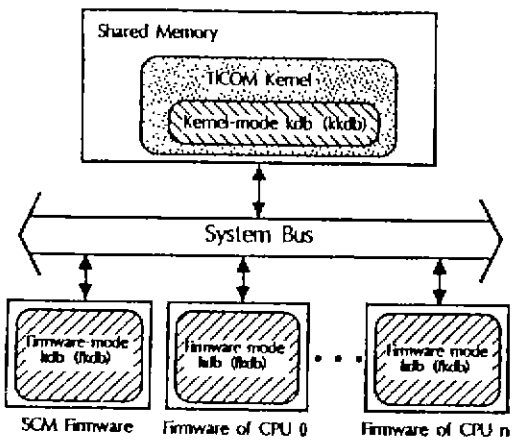

그림 6 혼합형 디버거 모델

으로 실행되는 커널모드 디버거 kkdb와, SCM 펌웨어(firmware)와 각 CPU의 펌웨어에서 실행되는 펌웨어모드 디버거 fkdb로 구성된다. Kkdb는 주로 breakpoint를 이용하여 커널의 실행을 일시적으로 정지시킨 뒤 주요 변수, 자료구조, 메모리 또는 레지스터 등을 참조하여 커널의 실행에 관련된 내부 정보를 수집한다. Fkdb는 프로세서의 정지, 재개 및 상태 조사, 프로세스의 강제 종료, 캐쉬 제어(enable/disable), 커널 모니터링 또는 프로세서 사이의 상호작용 조사 등 다중프로세서 측면에서의 디버깅을 지원한다.

이 디버거의 중요한 특징은 PC를 이용한다는 것이다. 즉, kkdb로부터 수집되는 커널의 순간 상태와 fkdb로부터 수집되는 모니터 정보를 SCM에 연결되어 있는 PC의 하드 디스크에 내력별로 저장하는 것이다. 이 PC는 커널의 실행도중 발생한 커널 이벤트에 대한 정보를 이력 형태로 저장하기 위한 안정된 저장 장치로 사용되며, 저장된 정보를 검색하는 데 이용된다.

PC에 커널 이벤트 정보를 저장함으로써 얻을 수 있는 잇점을 요약하면 다음과 같다. 첫째, 디버깅 대상 시스템의 화일 시스템이 파손(crash)되더라도 항상 디버깅 정보를 참조할 수 있다. 둘째, 저장된 정보로부터 상위 레벨 정보를 추출할 수 있다. 세째, 이력 정보를 바탕으로 특정 이벤트를 재실행시킴으로써 비재생산성 문제를 해결할 수 있다. 또한, PC에는 커널 이벤트 정보뿐만 아니라 메모리 내용 또는 레지스터 값과 같이 디버거에서 참조할 수 있는 일반적인 디버깅 정보도 화면에 출력된 형태대로 저장될 수 있다.

Fkdb[7,46]는 분산 디버거와 유사한 형태를 갖는다. 그림 7에서 볼 수 있는 바와 같이 fkdb는 각 프로세서의 펌웨어에서 동작하면서 다중프로세서의 실행을 제어하거나, 디버깅에 필요한 커널 이벤트 정보를 SCM에 연결되어 있는 PC의 하드 디스크에 저장한다.

펌웨어는 커널에 독립적으로 동작하기 때문에 fkdb는 다음과 같은 장점을 가진다. 첫째, 커널 이벤트 정보의 수집 및 저장시 커널에 대한 영향을 최소화할 수 있어 탐사 효과를 줄일 수 있다.

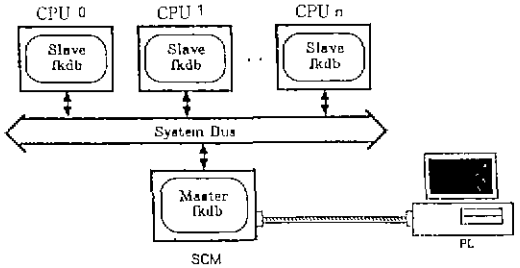

그림 7 펌웨어모드 디버거 모델

둘째, 커널이 전혀 동작하지 않는 경우에도 커널의 상태를 파악하는 데 이용될 수 있어 커널 개발 초기 단계에서부터 사용될 수 있다. 세째, 커널과 다중프로세서 시스템을 제어하기 용이하다.

이 모델의 단점은 커널 원시 코드뿐만 아니라 펌웨어 원시 코드도 수정되어야 한다는 것이다. 특히, kkdb는 대상 프로그램의 일부분으로 동작하는 디버거 모델이 가지는 단점을 모두 가진다. 또한, 펌웨어의 크기가 제한되어 있기 때문에 디버거의 크기 역시 제한될 수 밖에 없어 fkdb는 최소한의 주요 기능만을 지원할 수 있다. 더구나, 펌웨어에 디버거를 탑재하기 위한 하드웨어 장비들이 구비되어 있지 않을 경우에는 별도의 장치들을 이용하여야 한다.

3.7 각 디버거 모델의 비교

3.1절부터 3.6절까지 기술한 디버거 모델들을 2장에서 정의한 커널 디버거 요건에 따라 비교하면 표 1과 같다. ○로 표시된 항목은 2장에서 기술한 커널 디버거의 요건을 충분히 만족시킴을

표 1 커널 디버거 요건에 따른 디버거 모델 비교

디버거 모델 / 디버거 요건	일반 디버거	일부분으로 실행되는 디버거	원격 디버거	분산 디버거	부착 디버거	혼합형 디버거
정확성	○	○	○ △	△	○	○
탐사 효과 최소화	△	△	△ △	△	△	△
성능 저하 최소화	△	△	△ ○	○	△	△
이식성	○	△	△ △	○	○	△
디버깅 정보의 가용성	△	△	△ ○	○	△	○
사용 편의성	○	△	△ ○	○	○	△
커널 정지 및 재개		○	○			○
하드웨어 특성 반영		○	○	○		○
소프트웨어 특성 반영	△	○	△ △	△	△	○
디버깅 정보 표현	△	△	△ △	△	△	△
다양한 디버깅 수준		○	○			○
시스템 제어		○	○			○
상위 레벨 기능 지원	○	△	△ ○	○	○	○

의미하고, △로 표시된 항목은 요건 중 일부분만을 만족시킴을 의미하며, 표시되어 있지 않은 항목은 전혀 만족시키지 않음을 의미한다. 또한, 원격 디버거 항목의 좌측은 ROM 디버거의 특성을 나타내고 우측은 원격 시스템의 메모리에서 실행되는 디버거의 특성을 나타낸다.

4. 커널 디버거 기능

디버거는 사용자가 신속·정확하게 에러를 검출하고 그 원인을 자세히 분석할 수 있도록 다양한 기능들을 제공해야 한다. 커널 디버거는 디버거의 보편적인 기능 중 그 특성상 재시작(restart) 또는 역실행(backtrack)과 같은 일부 기능은 전혀 제공할 수 없지만, 특정 프로세스 또는 프로세서의 정지, 프로세스의 강제 종료, 캐쉬 통제 등과 같은 특수 기능은 지원되어야 한다. 본 장에서는 커널 디버거가 제공하여야 할 기능을 살펴본다.

4.1 실행 제어 기능

4.1.1 Breakpoint

Breakpoint는 사용자가 지정한 특정 주소의 명령어가 실행될 때 커널을 일시적으로 정지시키는 기능이다. 세부 기능으로는 breakpoint를 설정하거나 해제하는 기능, 설정되어 있는 breakpoint에 대한 정보를 검색하는 기능, 그리고 일시적으로 breakpoint를 on/off시키는 기능이 필요하다.

4.1.2 데이타 breakpoint

데이타 breakpoint는 사용자가 지정한 데이타가 사용(read/write)될 때 커널의 실행을 일시적으로 정지시키는 기능이다. Breakpoint가 주소를 이용하여 커널의 실행을 정지시키는 기능인 반면, 이 기능은 특정 메모리 또는 변수 값이 사용되면 커널의 실행을 정지시킨다. 즉, breakpoint는 커널을 명령어 또는 제어 흐름의 관점에서 파악하기 위해서 이용되지만 데이타 breakpoint는 커널을 데이타 흐름의 관점에서 파악하기 위해서 이용된다. 관련된 세부 기능으로는 데이타 breakpoint의 설정, 해제, 검색 및 일시적 on/off 기능이 필요하다.

4.1.3 조건부 breakpoint

조건부 breakpoint는 특정 메모리 또는 변수의 값이 사용자가 정의한 조건을 만족할 때 커널의 실행을 일시적으로 정지시키는 기능으로서 데이타 breakpoint를 발전시킨 것이다. 따라서 이 기능은 데이타 breakpoint와 함께 커널을 데이타 흐름의 관점에서 파악할 수 있도록 지원한다. 관련된 세부 기능으로는 조건부 breakpoint를 설정하거나 해제하는 기능과 설정되어 있는 조건부 breakpoint에 대한 정보를 검색하는 기능, 그리고 조건부 breakpoint를 일시적으로 on/off시키는 기능이 필요하다.

4.1.4 서브루틴 breakpoint

서브루틴 breakpoint는 사용자가 지정한 서브루틴이 호출될 때 커널의 실행을 일시적으로 정지시키는 기능이다. 관련된 세부 기능으로는 서브루틴 breakpoint의 설정, 해제, 검색, 및 on/off 기능이 필요하다.

4.1.5 입출력 breakpoint

입출력 breakpoint는 사용자가 지정한 위치에서 입출력이 발생할 때 커널의 실행을 일시적으로 정지시키는 기능이다. 관련된 세부 기능으로는 입출력 breakpoint의 설정, 해제, 검색 및 일시적 on/off 기능이 필요하다.

4.1.6 Single step과 branch trace

Single step은 매 명령어가 실행될 때마다 커널의 실행을 일시적으로 정지시키는 기능이고, branch trace는 커널의 실행 도중 분기가 발생할 때마다 커널의 실행을 일시적으로 정지시키는 기능이다.

4.1.7 Scan-point

Scan-point는 사용자가 지정한 주소에서 커널의 실행을 일시적으로 정지시킨 뒤 사용자 루틴을 커널에 동적으로 링크시켜 실행시키는 기능이다. 관련된 세부 기능으로는 scan-point를 설

정하거나 해제하는 기능, 설정되어 있는 scanpoint를 검색하는 기능, 그리고 사용자 프로그램을 동적으로 로드하여 커널에 링크시키는 기능이 필요하다.

4.1.8 프로세스 제어

프로세스 제어는 디버거가 특정 프로세스 또는 프로세스 그룹에 kill, abort, wait, exec 또는 hangup 등의 신호를 보냄으로써 프로세스를 제어하는 기능이다.

4.1.9 프로세서 정지 및 재개

프로세서 정지 및 재개는 다중프로세서용 커널 디버거에서 필요한 기능으로서 디버거 사용자가 지정한 하나 이상의 프로세서 또는 프로세서 그룹의 실행을 정지시키거나, 정지되었던 하나 이상의 프로세서 또는 프로세서 그룹을 재개시키는 기능이다. 이를 위해서는 프로세서를 그룹에 추가하거나 그룹에서 제거하는 등의 프로세서 그룹 관리 기능이 추가로 필요하다.

4.1.10 캐쉬 통제

캐쉬 통제는 프로세서의 캐쉬를 on/off시킴으로써 캐싱을 가능 또는 불가능하게 하는 기능이다. 다중프로세서 시스템에서는 하나 이상의 프로세서 또는 프로세서 그룹에 대해서도 동일한 처리를 할 수 있어야 한다.

4.2 검색 기능

4.2.1 레지스터 검색 및 변경

레지스터 검색 및 변경은 모든 레지스터의 값을 검색하거나 다른 값으로 변경하는 기능이다. 이 외에도 특정 또는 모든 레지스터를 사용자가 입력한 값으로 채우는 fill 기능이 필요하며, 다중프로세서 시스템에서는 특정 프로세서의 레지스터 값을 다른 프로세서의 레지스터로 복사하는 기능, 두 프로세서의 레지스터 값을 서로 교환하는 기능, 프로세서 별로 레지스터의 내용을 비교하는 기능 등이 추가로 필요하다.

4.2.2 메모리 검색 및 변경

메모리 검색 및 변경은 메모리를 16진수와 ASCII로 검색하거나 그 내용을 1, 2 혹은 4 바이트 단위로 변경하는 기능이다. 검색과 변경 이외에도 사용자가 지정한 위치로 메모리 내용을 복사하는 기능, 메모리 영역을 사용자가 지정한 값으로 채우는 fill 기능, 사용자가 지정한 두 개의 주소에 저장된 메모리 내용을 비교하는 기능, 그리고 메모리 영역 내에 사용자가 지정한 값이 존재하는지 탐색하는 기능 등이 필요하다.

4.2.3 커널 자료 구조 검색

커널 자료 구조 검색은 struct proc, user, preg, reg, mount, filsys, ofile, file. buf, inode, vfs, vnode, var 및 nvram 등의 주요 커널 자료 구조를 검색하는 기능이다. 자료 구조의 종류에 따라 조금씩 다르기는 하지만 관련된 세부 기능으로는 슬롯(slot) 번호를 이용한 검색, 프로세스 번호를 이용한 검색, 그리고 자료 구조의 주소를 이용한 검색 기능이 필요하다.

4.2.4 시스템 버퍼 검색

시스템 버퍼 검색은 커널 프로세스가 최근에 참조한 디스크 블럭의 내용을 파악하기 위하여 사용자가 지정한 또는 모든 시스템 버퍼의 내용을 16진수와 ASCII로 검색하는 기능이다.

4.2.5 심볼 테이블 검색

심볼 테이블 검색은 커널 실행 화일에 기록되어 있는 커널 심볼들의 이름과 주소를 검색하는 기능이다. 관련된 세부 기능으로는 심볼의 이름을 이용하여 주소를 검색하는 기능, 주소를 이용하여 심볼의 이름을 검색하는 기능과 모든 심볼의 이름과 주소를 검색하는 기능이 필요하다.

4.2.6 세마포어 검색과 커널 스택 역추적

세마포어 검색은 모든 전역 세마포어의 현재 값을 검색하는 기능이고, 커널 스택 역추적은 사용자가 지정한 커널 프로세스의 커널 스택을 역추적하면서 한 프레임(frame)씩 출력하는 기능이다.

4.2.7 프로세서 상태 검색

프로세서 상태 검색은 다중프로세서 시스템에서만 필요한 것으로 모든 프로세서의 현재 상태, 각 프로세서에서 실행 중인 프로세스에 관한 정보 등을 출력하는 기능이다.

4.3 입출력 기능

4.3.1 커널의 순간상태 저장

커널의 순간상태 저장은 커널이 디버깅되는 동안 접근 가능한 모든 변수의 값, 레지스터의 값, 프로세서의 상태 등을 요구된 특정 시각에 디스크로 저장하는 기능이다.

4.3.2 스크린 덤프

스크린 덤프는 디버거의 사용 도중 현 화면에 출력되어 있는 내용이나 사용자가 정의하는 범위내의 화면을 디스크에 저장하는 기능이다.

4.3.3 입출력

입출력은 디버거에서 디버깅에 관련된 데이타를 외부 장치에 출력하거나 외부 장치로부터 읽는 기능이다. 관련된 세부 기능으로는 디스크 또는 MT로부터 데이타를 읽는 기능, 데이타를 디스크 또는 MT로 출력하는 기능, 디스크 또는 MT의 초기화와 되감기(rewind)를 수행하는 기능, 그리고 SCSI 장치로 디버깅 정보를 출력하거나 필요한 데이타를 읽는 기능이 필요하다.

4.4 그밖의 기능

4.4.1 역어셈블

역어셈블은 breakpoint를 삽입할 정확한 주소를 파악하기 위해서 필요한 기능으로 메모리에 적재되어 있는 커널 실행 화일을 어셈블리 언어로 표현하는 기능이다. 관련된 세부 기능으로는 시작 주소만을 이용하는 일반 역어셈블, 주어진 시작 주소부터 끝 주소까지를 역어셈블하는 영역별 역어셈블 그리고 함수 단위의 역어셈블 기능이 필요하다.

4.4.2 패닉시 디버깅

패닉시 디버깅은 커널이 비정상적으로 실행을 완료하게 되는 경우 온라인(on-line)으로 커널 상태를 조사하거나 원인을 파악하는 기능이다.

4.4.3 락 및 세마포어 관련 기능

락(lock) 및 세마포어 관련 기능은 락과 세마포어의 일관성(consistency)과 우선순위를 조사하고 경쟁률을 측정하며, 교착상태를 검출하는 기능이다.

5. 커널 디버거 예

본 장에서는 커널 디버거의 예로서 단일프로세서 운영 체제용 커널 디버거인 SVR4 커널 디버거와 공유 메모리 다중프로세서 운영 체제용 커널 디버거인 주전산기II 커널 디버거에 대하여 간략히 기술한다.

5.1 SVR4 커널 디버거

SVR4 커널 디버거[27,48]에는 DEBUGGER와 GDEBUGGER의 두 가지 디버거가 있다. 이들 두 디버거는 설치(install)시에 한 가지 또는 두 가지 모두를 설치할 수 있으며, 사용자의 디버거 변경 명령에 의해서 다른 디버거로 변경할 수 있다. 본 절에서는 두 디버거 중에서 메인 루틴의 이름이 _gdebugger()인 GDEBUGGER를 중심으로 전체적인 실행 과정과 주요 기능 및 panic시 처리에 대해서 설명한다.

커널 디버거가 실행되면 GDEBUGGER는 우선 외부 인터럽트를 무시하도록 인터럽트 플래그를 클리어(clear)시킨다. 그 다음 예외상태 스택을 조사해서 스택의 내용을 출력하고, 예외상태 스택을 참조할 수 있도록 스택의 시작 주소를 가져온다. 계속해서 GDEBUGGER는 발생한 트랩의 종류를 출력하고, 몇 번째 breakpoint 또는 watchpoint인가 하는 정보와 트랩을 발생시킨 루틴의 이름을 출력한다. 그 다음, 레지스터의 내용을 출력하고 사용자의 디버깅 명령을 입력받아 수행한다. 명령의 수행이 끝나고 사용자가 single step 또는 continue 명령을 입력하면 인터럽트 가능 플래그를 세트하고 커널로 제어를

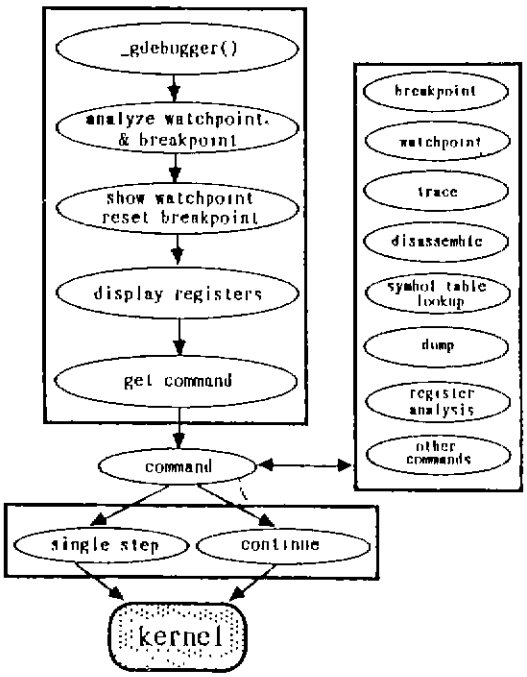

그림 8 커널 디버거의 실행 과정

넘긴다.

디버거가 실행되는 전반적인 과정은 그림 8과 같다.

GDEBUGGER가 제공하는 주요 기능은 breakpoint, single step, 레지스터 검색 및 변경, 심볼 테이블 검색, 그리고 메모리 검색 및 변경 기능이다. 그 밖에도 하드웨어에서 제공하는 DR0~DR3의 디버그 레지스터를 이용하여 커널의 동작을 일시적으로 정지시키는 watchpoint와 일종의 조건부 breakpoint로서 단순 추적과 특정 메모리 값의 진위 여부에 의한 추적을 지원하는 trace 기능이 있다.

패닉이 발생하면 커널은 panic() 루틴을 호출한다. 이 루틴은 메모리의 내용을 덤프한 뒤, 페닉 레벨을 참조하여 시스템의 수행에 큰 지장이 없으면 경고 메시지를 출력한 후 디버거로부터 리턴하여 정지되었던 작업을 계속 수행하고, 그렇지 않으면 재부팅 메시지를 출력한 후 무한 loop에 들어가 시스템이 전체적으로 다운(down)된다. GDEBUGGER는 메모리 내용을 덤프하기 전에 호출되어 패닉이 발생한 순간의 상태를 온라인으로 파악하도록 지원한다.

5.2 주전산기 II 커널 디버거

주전산기II의 커널 디버거[1-3,6,7,46]는 3.6절에서 기술한 혼합형 모델을 기반으로 개발된 것으로서 커널모드 커널 디버거 kkdb와 펌웨어모드 커널 디버거 fkdb로 구성되어 있다.

5.2.1 Kkdb

Kkdb는 첫째, 수퍼 유저가 kdb를 입력하는 경우 시스템 콜에 의하여, 둘째 breakpoint나 single step 등의 트랩이 발생한 경우 커널의 trap() 루틴을 거쳐서, 세째 시스템 패닉이 발생한 경우 panic() 루틴을 거쳐 각각 호출될 수 있다. Kkdb는 호출되자마자 자신이 실행될 프로세서를 제외한 모든 프로세서들에 실행을 정지하도록 인터럽트를 보낸다. 각 프로세서는 디버거로 진입하여 자신의 레지스터를 디버거의 내부 자료구조에 저장하고 자신의 지역 클럭을 정지시킨 뒤 디버깅이 완료될 때까지 실행을 정지한다. 이렇게 함으로써 디버거가 호출된 순간의 정확한 커널 상태를 파악할 수 있다.

디버거는 커널에 삽입되어 있는 breakpoint를 제거하고 user 블럭 페이지 테이블의 주소, 프로세서 번호, 프로세스 번호, PC, SP, SR 및 디버거가 호출된 원인 등을 출력한다. 그 다음 반복수행을 하면서 사용자의 명령을 입력받아 해당 기능을 실행시키고 처리 결과를 출력한다. 사용자가 "run"을 입력하면 디버거는 실행을 정지한 모든 프로세서들에 실행을 재개하도록 인터럽트를 보내고 커널로 리턴한다.

Kkdb가 제공하는 주요 기능에는 breakpoint, 제한된 조건부 breakpoint, scan-point, single step, branch trace, 레지스터 검색 및 변경, 메모리 검색 및 변경, 커널 자료 구조 검색, 시스템 버퍼 검색, 심볼 테이블 검색, 커널 스택 역추적, 그리고 패닉시 디버깅 기능 등이 있다. 그 밖에도 모든 전역 세마포어의 현재 값을 검색하는 세마포어 값 검색 기능, 디버거 사용 도중 화면에 출력되는 내용을 디스크 혹은 PC에 저장하거나 프린터로 출력하는 스크린 덤프 기능, 커널 실행

화일의 역어셈블 기능, 그리고 락과 세마포어에 관련된 디버깅 기능 등을 제공한다.

5.2.2 Fkdb

그림 7에서 보인 바와 같이 fkdb는 주 fkdb와 종 fkdb로 구성된다. 즉, 각 프로세서의 펌웨어에는 동일한 능력을 갖는 종(slave) 디버거가 설치되고 SCM의 펌웨어에는 fkdb를 전체적으로 제어하는 주 디버거가 설치된다. 주 fkdb는 사용자로부터 디버거 명령어를 입력받아 이를 직접 처리하기도 하고, 하나 이상의 종 fkdb에 전달하여 원격 실행시키기도 한다. 종 fkdb는 주 fkdb로부터 전달받은 명령을 처리하고 그 결과를 주 fkdb에 반환한다. 주 fkdb는 사용자 명령어의 처리 결과를 수집하여 출력한다.

펌웨어 모드에서 "kdebug"를 입력하면 fkdb를 실행시킬 수 있다. 이 때, 실제적으로는 주 fkdb만 실행되기 때문에 다른 프로세서들은 어떠한 영향도 받지 않고 자신의 작업을 계속 수행한다. 프로세서가 작업 수행 도중 주 fkdb로부터 사용자의 명령을 수행하도록 메시지를 받게 되면 우선 실행중인 프로세스를 정지시키고 종 fkdb 모드로 전환하여 명령을 처리한다. 그 다음, 명령의 처리 결과를 주 fkdb에 전송한 뒤, 종 fkdb 모드에서 빠져나와 정지되었던 프로세스를 재개시킨다. 사용자는 -P 옵션을 사용하여 명령을 수행시킬 프로세서를 지정할 수 있다. 디버깅이 완료되면 주 fkdb는 실행을 정지하고 있는 프로세서에 실행을 재개하도록 명령을 보낸 뒤 펌웨어 모드로 전환한다.

6. 결 론

개발 중인 커널을 디버깅하기 위해서는 커널의 실행에 관한 주요 자료 구조, 레지스터, 메모리 등을 검색하여야 하며 프로세서와 프로세스를 제어할 수 있어야 한다. 뿐만 아니라 계속적으로 변화하는 커널 상태의 모니터링과, 그 결과를 이용하여 에러의 원인을 정확하게 파악할 수 있어야 한다. 현재까지 개발되어 사용되고 있는 대부분의 디버거는 일반 사용자들을 위한 디버거이므로 커널 개발의 특성을 고려하지 않은 것이어서 커널의 디버깅에 직접 이용하기가 매우 어려운 실정이다. 더구나, 다중프로세서 운영 체제 커널은 병행 프로그램이기 때문에 비결정성, 탐사 효과, 비재생산성 등의 문제를 내포하고 있어서 디버깅이 더욱 어렵다. 결과적으로 커널의 개발 또는 이식에 많은 시간이 필요하게 되어 개발된 시스템의 상품화와 안정화 및 신뢰성 향상에 많은 어려움이 있었다.

본 고에서는 Unix 커널의 디버깅에 이용될 수 있는 디버깅 기법에 대하여 간략하게 알아보고 커널 디버거가 갖추어야 할 요건들을 일반적인 요건, 구조적 요건 및 기능적 요건으로 구분하여 살펴보았다. 또한, 커널 디버거의 여섯 가지 모델을 알아보고 각각을 비교하였다. 그리고, 커널 디버거가 제공하여야 할 기능들을 실행 제어 기능, 검색 기능, 입출력 기능 그리고 그밖의 기능으로 구분하여 기술하였으며, 커널 디버거의 예로서 SVR4 kdb와 주전산기II 커널 디버거에 대하여 간략히 소개하였다.

지금까지 개발된 커널 디버거를 개선·발전시키기 위해서는 먼저, 사용자 인터페이스가 개선되어야 한다. 커널 디버거는 다른 디버거와는 달리 윈도우 시스템의 지원을 받기 힘들기 때문에 사용자 인터페이스가 상대적으로 취약하다. 이 때문에, 하위 레벨의 디버깅 정보로부터 상위 레벨의 정보를 추출하여 시각화하기 어려워 결과적으로 사용자의 이해를 종합적으로 돕지 못하는 경우가 있다. 그 다음, 커널 디버거의 이식성을 높이기 위한 방안이 마련되어야 한다. 커널 원시 코드를 수정하지 않고도 디버거를 설치할 수 있어야 하며, 커널 자료 구조가 변경될 때마다 디버거 원시 코드를 수정하지 않아도 이를 반영할 수 있는 커널 자료 구조 검색 지원 도구가 개발되어야 한다. 마지막으로, 디버깅 정보를 보다 효율적으로 표현하기 위한 방안으로서 이력 정보를 이용하여 특정 이벤트를 재실행시키기 위한 방안과 저장된 이벤트 정보로부터 상위 레벨 정보를 추출하여 시각화하기 위한 방안도 함께 연구되어야 한다.

참고문헌

[1] 김성조, MOS 커널 개발 도구에 관한 연구, 최종

보고서, 컴퓨터 신기술 공동연구소, 1991.
[2] 김성조, Dynamic MOS Kernel 개발 Tool 연구, 최종 보고서, 한국 전자 통신 연구소, 1992.
[3] 김성조, Dynamic MOS Kernel 개발 Tool 연구, 최종 보고서, 한국 전자 통신 연구소, 1993.
[4] 남영호, Geometry Model을 이용한 병행 프로그램의 정적 교착상태 검출 도구(SDDT)의 설계 및 구현, 석사 학위 논문, 중앙대학교 전자계산학과, 1990.
[5] 남영호, 김성조, "기하학적 모델을 이용한 정적 교착상태 검출 도구(SDDT)의 설계 및 구현," 한국 정보 과학회 논문지, 제 18 권, 제 6 호, 1991.
[6] 박상서, 동적 MOS 커널 디버거의 설계 및 구현, 석사 학위 논문, 중앙대학교 전자계산학과, 1992.
[7] 박상서, 김성조, "펌웨어를 이용한 대칭형 다중 프로세서 커널 디버거," UNIEXPO '94 논문집, KUUG 1994, pp. 129~133.
[8] 박종순, MOS 커널용 정적 교착상태 검출 도구 설계 및 구현, 석사 학위 논문, 중앙대학교 전자계산학과, 1991.
[9] S. A. Zimmerman, "A Debugger for Unix Kernel," Proc. of the USENIX Summer Conf., 1985, pp. 151~154.
[10] E. Adams and S. S. Muchnick, "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations," Software-Practice and Experience, Vol. 16, No. 7, Jul. 1986, pp. 653~669.
[11] AT&T, Adb Source Code of Unix System V Release 4, 1991.
[12] CMU, Adb Source Code of Mach MSD 2.6, 1991.
[13] K. Araki, et al., "A General Framework for Debugging," IEEE Software, May 1991, pp. 14~20.
[14] Z. Aral and I. Gertner, "High-Level Debugging in Parasight," ACM SIGPLAN & SIGOPS Workshop on Parallel & Distributed Debugging, Vol. 24, No. 1, Jan. 1989, pp. 151~162.
[15] Z. Aral, et al., "Efficient Debugging Primitives for Multiprocessors," Proc. of the 3rd Int'l Conf. on ASPLOS, Apr. 1989, pp. 87~95.
[16] F. Baiardi, et al., "Development of a Debugger for a Concurrent Language," Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Mar. 1983, pp. 98~106.
[17] P. Bates and J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," The Journal of Systems and Software, Vol. 3, Dec. 1983, pp. 255~264.
[18] P. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems," Proc. of the Int'l Conf. on Distributed Computing Systems, 1988, pp. 308~315.
[19] T. A. Cargill, "The Feel of Pi," Proc. of the Winter USENIX Meeting, Jan. 1986, pp. 62~71.
[20] T. A. Cargill, "Pi-A Distributed Debugger," EUUG Autumn '86, Sep. 1986, pp. 137~141.
[21] P. Corsini and C. A. Prete, "Multibug: Interactive Debugging in Distributed Systems," IEEE Micro, Vol. 6, No. 3, 1986, pp. 26~33.
[22] R. Cooper, "Pilgrim: A Debugger for Distributed Systems," Proc. of the Int'l Conf. on Distributed Computing Systems, 1987, pp. 458~465.
[23] P. A. Emrath, et al., "Detecting Nondeterminacy in Parallel Program," IEEE Software, Feb. 1992, pp. 69~77.
[24] J. Fagerström, et al., Distributed Debugging-collected ideas, Research Report, Linkoping Univ., Jun. 1986.
[25] A. Fortin, "Debugging of Heterogeneous Parallel Systems," Proc. of Workshop on Parallel and Distributed Debugging, 1988, pp. 130~140.
[26] GNU, Gdb 4.3 Source Code, 1992.
[27] UNISYS, Kdb Source Code of Unix System V Release 4/MP, 1988.
[28] J. Gait, "A Debugger for Concurrent Programs," Software-Practice and Experience, Vol. 15, No. 6, Jun. 1985, pp. 539~554.
[29] J. Gait, "A Probe Effect in Concurrect Programs," Software-Practice and Experience, Vol. 16, No. 3, Mar. 1986, pp. 225~233.
[30] G. S. Goldszmidt, "High-Level Language Debugging for Concurrent Programs," ACM Trans. on Computer Systems, Vol. 8, No. 4, Nov. 1990, pp. 311~336.
[31] J. Griffin, Parallel Debugging System User's Guide, Technical Report, Los Alamos National Laboratory, 1987.
[32] L. Gunaseelan and R. J. LeBlanc, Jr., "Debugging Objects and Threads in a Shared Memory System," Proc. of the USENIX Symp. on Experience with Distributed and Multiprocessor Systems (SEDMS IV), Sep. 1993, pp. 157~

173.
[33] M. Himelstein and P. Rowell, "Multi-process Debugging," Proc. of the USENIX Summer Conf., 1985, pp. 155~158.
[34] Intel Corp., iPSC Concurrent Debugger Manual, 1987.
[35] J. D. Johnson and G. W. Kenney, "Implementation Issues for a Source Level Symbolic Debugger," Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging, Mar. 1983, pp. 149~151.
[36] J. Joyce, et al., "Monitoring Distributed Systems," ACM Trans. on Compuer Systems, Vol. 5, No. 2, May 1987, pp. 121~150.
[37] R. Tischler, et al., "Static Analisys of Programs as an Aid to Debugging," Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging, Mar. 1983, pp. 155~158.
[38] T. Lehr, el al., "Visualizing System Behavior," Int'l Conf. on Parallel Processing, Vol. II, 1991, pp. 117~123.
[39] S. J. Leffler, et al., The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley Publishing Co., 1989.
[40] T. J. LeBlanc and J. M. Mellor-Crumney, "Debugging Parallel Programs with Instant Replay," IEEE Trans. on Computers, Vol. C-36, No. 4, Apr 1987, pp. 471~482.
[41] T. Lehr, et al., "Visualizing Performance Debugging," IEEE Computer, Oct. 1989, pp. 38~51.
[42] H. F. Li, et al., "cdb: A Toolkit for Debugging Distributed Programs," Proc of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991, pp. 242~244.
[43] P. Maybee, "pdb: A Network Oriented Symbolic Debugger," Proc. of the USENIX Winter Conf, Jan. 1990. pp. 41~52.
[44] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," ACM Computing Surveys, Vol. 21, No. 4, Dec. 1989, pp. 593~622.
[45] D. M. Ogile, et al., "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems." IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 7, Jul. 1993, pp. 762~778.
[46] S. Park and S. Kim, "A Unix Kernel Debugger for Shared Memory Multiprocessor Systems," Proc. of the InfoScience '93, Oct. 1993. pp. 628~639.
[47] B. Rieken and J. Webb, Advantures in UNIX-Kernel Structure and Flow, .sh Consulting Inc., 1989.
[48] AT&T, Kdb Source Code of Unix System V Relcase 4, 1991.
[49] R. Seidner and N. Tindall, "Interactive Debug Requirements," Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging, Mar. 1983, pp. 25~31.
[50] Sequent Corp., Dynix Pdbx Parallel Debugger User's Manual, 1986.
[51] T. Shimomura and S. Isoda, "Linked-List Visualization for Debugging," IEEE Software, May 1991, pp. 44~51.
[52] D. Socha, et al., "Voyeur: Graphical Views of Parallel Programs." ACM SIGPLAN Notice, Vol. 24, No. 1, Jan. 1989, pp. 206~215.
[53] SUN Micro Systems, Sdb Manual, 1990.

박 상 서

1991 중앙대학교 전자계산학과 공학사
1993 중앙대학교 전자계산학과 공학석사
1993 ~현재 중앙대학교 컴퓨터공학과 박사과정
관심 분야 : 병렬 및 다중처리, 운영체제, 디버깅, 성능 평가, 시스템 관리

김 성 조

1975 서울대학교 응용수학과 공학사
1977 한국과학기술원 전산과 이학석사
1977 ~1980 ADD 연구원
1980 ~현재 중앙대학교 컴퓨터공학과 부교수
1987 Univ. of Texas at Austin 이학박사
1987 ~1988 Univ. of Texas at Austin Research Fellow
관심 분야 : 병렬 및 다중처리, 디버깅, 시스템 망 관리, 멀티미디어