효과적인 메모리 테스트를 위한 가상화 커널
(A Virtualized Kernel for Effective Memory Test)

박 희 권 † 윤 대 석 † 최 종 무 ††
(Heekwon Park) (Deaseok Youn) (Jongmoo Choi)

요 약 본 논문에서는 64비트 다중-코어 컴퓨팅 환경에서 효과적인 메모리 테스트를 위한 가상화 커널을 제안한다. 이때 효과적이라는 용어는 커널이 존재하는 메모리 공간을 포함한 모든 물리 메모리 공간에 대한 테스트를 시스템 리부팅 없이 수행할 수 있음을 의미한다. 이를 위해 가상화 커널은 4가지 기법을 제공한다. 첫째, 커널과 응용이 물리 메모리를 직접 접근 할 수 있게 하여 원하는 메모리 위치에 다양한 메모리 테스트 패턴을 쓰고 읽는 것이 가능하게 한다. 둘째, 두 개 이상의 커널 이미지가 다른 메모리 위치에서 수행 가능하도록 한다. 셋째, 커널이 사용하는 메모리 공간을 다른 커널로부터 격리한다. 넷째, 커널 하이버네이션을 이용하여 커널 간에 문맥 교환을 제공한다. 제안된 가상화 커널은 인텔사의 Xeon 시스템 상에서 리눅스 커널 2.6.18을 수정하여 구현되었다. 실험에 사용된 Xeon 시스템은 2개의 Dual-core CPU와 2GB 메모리를 탑재하고 있다. 실험 결과 설계된 가상화 커널이 메모리 테스트에 효과적으로 사용될 수 있음을 검증할 수 있었다.

키워드 : 가상화, 가상화 커널, 메모리 테스트, 리눅스, 하이버네이션

Abstract In this paper, we propose an effective memory test environment, called a virtualized kernel, for 64bit multi-core computing environments. The term of effectiveness means that we can test all of the physical memory space, even the memory space occupied by the kernel itself, without rebooting. To obtain this capability, our virtualized kernel provides four mechanisms. The first is direct accessing to physical memory both in kernel and user mode, which allows applying various test patterns to any place of physical memory. The second is making kernel virtualized so that we can run two or more kernel image at the different location of physical memory. The third is isolating memory space used by different instances of virtualized kernel. The final is kernel hibernation, which enables the context switch between kernels. We have implemented the proposed virtualized kernel by modifying the latest Linux kernel 2.6.18 running on Intel Xeon system that has two 64bit dual-core CPUs with hyper-threading technology and 2GB main memory. Experimental results have shown that the two instances of virtualized kernel run at the different location of physical memory and the kernel hibernation works well as we have designed. As the results, the every place of physical memory can be tested without rebooting.

Key words : Virtualization, Virtualized Kernel, Memory Test, Linux, Hibernation

1. 서 론

본 논문에서는 메모리 테스트를 효과적으로 수행하기 위한 가상화 커널을 제안한다. 여기서 메모리 테스트는 메모리 모듈 출시 전후 셀의 결함 여부를 검증하는 과정을 말한다. 모든 메모리는 검증 과정을 통과해야만 출시될 수 있다. 이러한 과정을 통해 오류가 검출된 모듈에 대해서는 원인 분석과 함께 해결 방안에 대한 연구가 진행된다. 이는 메모리 모듈의 신뢰성 향상시키고 오류율을 감소시키는데 기여하며 비용 절감에도 영향을 미친다[1-3].

기존 대표적인 메모리 테스트 프로그램으로는 X86 기

† 학생회원 : 단국대학교 컴퓨터과학과
parkhk81@dankook.ac.kr
woodsmen@dankook.ac.kr
†† 종신회원 : 단국대학교 컴퓨터과학과 교수
choijm@dankook.ac.kr
논문접수 : 2007년 1월 22일
심사완료 : 2007년 10월 16일

정보과학회논문지: 시스템 및 이론 제34권 제12호(2007.12)

반 메모리 테스트 프로그램인 memtest86이 있다[4]. 이는 독자 구동 방식(standalone)으로서 32비트환경에서 동작하도록 설계되었다. 하지만 하드웨어의 고성능화에 따라 32비트에서 64비트로 진화된 프로세서에는 부적합하다. 또한 운영체제가 없는 환경에서 동작하기 때문에 다양한 테스트 패턴의 적용이 복잡하며 파일과 네트워크를 지원하는데 있어서 어려움을 내포하고 있다. 이에 따라 운영체제의 관리 하에 효과적으로 대용량의 메모리를 테스트 할 수 있는 기법에 대한 연구가 필요하게 되었다. 이는 기존 메모리 테스트 프로그램에 비하여 테스트 프로그램의 작성 및 이식이 용이하고 파일 시스템과 네트워크 등을 사용할 수 있다는 장점이 있다.

본 논문은 가상화 커널이라는 개념을 도입하여 운영체제의 관리 하에 효과적으로 전체 메모리 모듈을 테스트 할 수 있는 기법을 제안한다. 가상화 커널은 실행 시간 중에 커널 간 문맥교환을 유연하게 수행할 수 있는 커널로서 리눅스 커널 2.6.18버전에서 X86-64 아키텍처를 기반으로 구현하였다[5-9]. 세부 기술 사항은 크게 네 가지로 분류 될 수 있다. 첫 번째, 응용이나 커널에서 물리메모리 직접 접근을 위한 인터페이스를 제공함으로서 다양한 패턴의 메모리 테스트가 가능하도록 하였다. 두 번째, 커널의 메모리상 위치를 유연하게 조정할 수 있도록 함으로서 메모리 일정영역에 상주하는 커널을 사용자가 원하는 곳에 위치할 수 있도록 설계하였다. 세 번째, 메모리 격리성(Isolation)을 제공함으로서 여러 개의 커널이 독립적으로 메모리를 관리할 수 있도록 설계하였다. 네 번째, 커널 하이버네이션 기능을 구현함으로서 커널 간 문맥교환을 유연하게 하였다.

본 논문의 구성은 다음과 같다. 2절에서는 선행 연구, 3절에서는 리눅스 커널 2.6.18버전 기반 가상화 커널의 설계에 대하여 기술한다. 4절에서는 인텔 64비트 멀티코어를 기반으로 한 가상화 커널 구현사항에 대하여 기술하고, 5절에서는 실험 결과를 보여준다. 6절에서는 결론 및 향후 연구 계획에 관하여 기술한다.

2. 선행 연구

2.1 기존 메모리 테스트 프로그램 구동 방식

기존 메모리 테스트 프로그램은 운영체제 없이 독자적으로 메모리에 존재하면서 모듈의 오류를 검증 하도록 설계되었다. 즉 아키텍처 수준의 언어로 제작되며 부팅 과정 등을 포함한 기본적인 환경을 스스로 제공해야 한다. 이러한 종류의 대표적인 메모리 테스트 프로그램으로는 memtest86이 있다. memtest86 프로그램은 32비트 모드로 동작하는 X86 아키텍처를 기반으로 제작되었으며 GPL(Gnu Public License)로 등록되어 있다. 이 프로그램은 내부적으로 10가지 정도의 메모리 테스트 패턴을 적용하여 모듈의 결함을 검증한다. 대표적인 패턴으로는 Address test, Moving inversions, Block move, Random number sequence, Modulo 20, Bit fade test 등이 있다.

하지만 memtest86에서 제공하는 테스트 패턴 만으로는 모든 종류의 오류를 발견해 낼 수 없다. 그래서 대부분의 메모리 모듈 생산 업체는 자체 개발한 메모리 테스트 프로그램을 사용하여 테스트를 진행한다. 자체 개발되는 메모리 테스트 프로그램은 대부분 아키텍처 종속적인 코드로 작성되며 memtest86처럼 운영체제 없이 독자적으로 동작한다. 이 때문에 테스트 프로그램 작성이 어려우며 이식성이 낮다. 또한 GUI(Graphic User Interface)환경을 제공하지 못하고 디스크나 네트워크 장치 등의 사용이 불가능한 점 등 여러 가지 제약이 존재한다.

한편, memtest86과 달리 운영체제의 관리 하에 동작하는 메모리 테스트 프로그램도 존재한다. 대표적으로는 memtester를 들 수 있다[10]. 하지만 이 프로그램은 운영체제로부터 메모리를 할당 받아서 테스트를 진행하기 때문에 사용자가 원하는 영역을 테스트 할 수 없을 뿐만 아니라, 실제 어느 물리 주소가 테스트 되고 있는지 조차 알 수가 없다. 운영체제는 가상 메모리를 사용하기 때문이다. 더욱이 운영체제는 메모리 일정영역에 상주하기 때문에 메모리 테스트 영역 확보에 있어서 치명적이다. 즉, 전체 물리메모리에 대한 테스트가 불가능하다.

본 연구진은 가상화 시스템의 개념을 도입하여 기존 메모리 테스트 프로그램들의 장점을 취합한 새로운 메모리 테스트 환경을 제안한다. 즉, 운영체제가 동작하는 시스템에서 독자 구동 방식의 메모리 테스트 프로그램과 유사한 방법으로 테스트를 할 수 있는 운영체제를 제안한다.

2.2 가상화 기술

현재 대표적인 가상화 시스템으로는 Xen과 VMware가 있다[11,12]. 이들 가상화 시스템의 기본 정책은 하나의 물리 머신위에 여러 개의 운영체제가 동작하는 것을 허용하는 것이다. 이것은 VMM(Virtual Machine Monitor)이라는 가상 계층을 운영체제와 물리머신 사이에 관리 계층으로 사용함으로서 가능하게 된다.

가상화 시스템의 구조는 그림 1과 같다. 그림과 같이

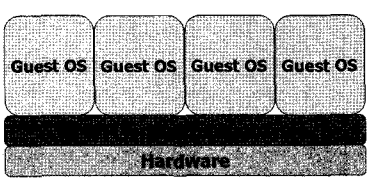
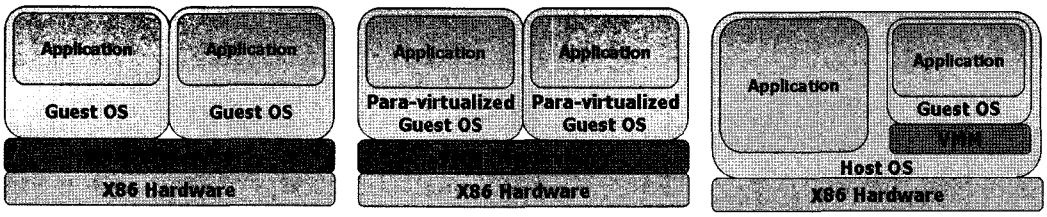


그림 1 가상화 시스템 구조



(a) Full-virtualization (b) Para-virtualization (c) HostOS-based virtualization

그림 2 가상화 시스템의 분류

실제 물리 장치는 가상화 계층에서 관리하고 이 계층에서 논리적으로 분할된 가상 자원을 이용하여 각각의 운영체제와 응용 프로그램을 수행한다. 이러한 기법으로 인하여 각각의 운영체제는 독립된 자원을 사용하는 것과 같은 효과를 얻게 된다. 즉, 상호 운영체제간의 격리성이 보장된다.

가상화는 크게 전가상화, 반가상화, 호스트 기반 가상화로 분류될 수 있다. 전가상화는 가상화 계층(VMM)이 수정되지 않은 GuestOS[1]에게 가상화 환경을 제공하는 것이다. 이 환경은 가상 자원을 VMM위에 구성해 주며, GuestOS의 수정 및 간섭 없이 완벽하게 물리적인 하드웨어 자원을 이용할 수 있도록 지원해준다. 반면 반가상화는 VMM과의 원활한 통신과 성능 향상을 위해 GuestOS의 일부 수정을 허용한다. 따라서 GuestOS에서 일부는 직접적으로 물리적인 하드웨어 자원에 접근을 하고 일부는 수정된 API를 통해 접근한다. 호스트 기반 가상화는 VMM이 호스트 운영체제 위에 설치되어 GuestOS를 생성하는 아키텍처로 GuestOS의 수정 및 간섭 없이 호스트 운영체제가 인식한 하드웨어를 가상 자원으로 재구성하여 사용한다. 그림 2는 위에서 언급한 3가지 가상화의 아키텍처를 보여주고 있다. VMware ESX, HP의 Integrity 등이 전가상화의 대표적인 예이며, Xen 등이 반가상화의 대표적인 예이다. 한편 VMware의 VMworkstation, Planet Lab의 Vserver등은 호스트 기반 가상화의 대표적인 예이다[13-16].

가상화 기술을 도입하면 효과적인 메모리 테스트가 가능하다. 그림 2의 (b)를 예로 들면, 왼쪽 GuestOS상의 메모리 테스트 프로그램이 오른쪽 GuestOS가 사용하는 물리 메모리 공간을 테스트하고, 그 이후에 오른쪽 GuestOS상의 메모리 테스트 프로그램이 왼쪽 GuestOS가 사용하던 물리 메모리 공간을 테스트 하면, 운영체제가 존재하던 물리 메모리 영역 테스트가 가능해 진다. 하지만 가상화 계층을 위한 소프트웨어가 존재하는 메모리 영역은 테스트가 불가능하다. Xen의 경우 이 영역의 크기가 64MB 정도이다. 이 문제를 해결하기 위해 본 연구진은 가상화 커널이라는 새로운 개념을 도입한다.

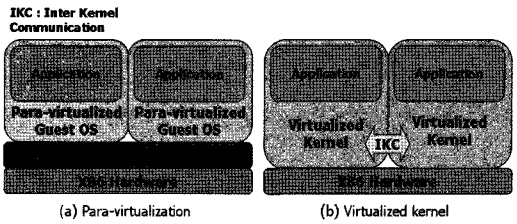
3. 가상화 커널의 설계

3.1 가상화 커널

본 논문에서 제안한 가상화 커널은 복수개의 운영체제가 하나의 플랫폼에서 동작할 수 있도록 설계된 운영체제이다. 이는 가상화 시스템과 유사하다. 하지만 기존 가상화 시스템과는 다르게 가상화를 위한 특별한 계층이 존재하지 않으며 상호 운영체제간의 통신을 통해 동작하도록 설계되었다.

그림 3은 반 가상화 시스템과 본 논문에서 제안한 가상화 커널의 차이점을 보여준다. 가상화 커널은 Guest-OS의 수정을 허용한다는 측면에서 반 가상화 기법과 유사하다. 하지만 VMM이 독립된 층으로 존재하는 것이 아니라 그림 3의 (b)처럼 GuestOS 내부에 포함되어 있다. 따라서 한 GuestOS가 다른 GuestOS의 물리 공간을 테스트하고, 그 이후 반대로 테스트함으로서 전체 물리 메모리 공간에 대한 테스트가 가능해 진다. 이러한 구조는 L4연구에서 제안한 IPVMM(In-Place VMM)과 유사한 접근 방법이다[14].

본 논문에서는 메모리 테스트를 위한 가상화 커널을 구현하기 위해 4가지 기법을 제공한다. 첫 번째, 가상 메모리를 사용하는 운영체제 위에서 물리 메모리 직접 접근을 위한 인터페이스를 제공한다. 이는 새로운 메모리 테스트 패턴의 적용을 용이하게 해주며 오류에 대한 직관적인 해석을 가능하게 해준다. 두 번째, 물리 메모리 여러 위치에 커널을 적재 시키는 기법을 제공한다.



(a) Para-virtualization (b) Virtualized kernel

그림 3 기존 가상화 시스템과 가상화 커널의 비교

1) 게스트 운영체제는 가상화 시스템 위에 설치되는 운영체제로서 하나의 머신 위에 여러 개의 게스트 운영체제가 동작가능하며, 가상 머신이라고도 부른다.

가상화 커널을 위해서는 여러 운영체제가 서로 다른 위치에서 수행될 수 있어야 한다. 세 번째, 메모리 격리성을 제공한다. 이는 각 운영체제가 상호간의 간섭 없이 동작 할 수 있음을 보장해준다. 네 번째, 커널 하이버네이션을 제공한다. 이 기법은 커널 간 문맥교환을 유연하게 하는데 기여한다.

3.2 메모리 테스트를 위한 물리 메모리 직접 접근 설계

최근 운영체제는 가상 메모리를 사용한다. 하지만 메모리 테스트 프로그램은 자신이 원하는 물리 주소에 직접 접근할 수 있어야 한다. 물리 메모리를 직접 접근해야만 moving inversions 같은 여러 가지 테스트 패턴을 적용할 수 있을 뿐 아니라, 어떠한 메모리 모듈에 오류가 있는지를 직관적으로 해석할 수 있기 때문이다. 따라서 효과적인 메모리 테스트를 지원하는 운영체제는 테스트 프로그램이 원하는 물리 주소를 접근할 수 있는 사용하기 편한 인터페이스를 제공해야 한다.

가상 주소에서 물리 주소로 변환하는 과정에는 페이지 테이블이 이용된다. 그림 4는 본 논문의 연구 기반 운영체제인 리눅스 커널 2.6.18버전에서 X86-64 아키텍처의 가상 주소와 물리 주소 그리고 주소 변환 테이블들의 관계를 보여준다. Intel과 AMD등 64비트 컴퓨팅 환경에서는 PML4(Page Map Level-4), PDP(Page Directory Pointer), PD(Page Directory), PT(Page Table)을 이용하여 총 4단계의 페이징을 통해 주소를 변환을 한다. 한편, x86_64에서 상위 16비트는 sign extend로 실제 주소 변환에서는 무시된다.

결국 그림 4의 주소 변환 테이블들을 적절하게 제어하면 응용이 원하는 물리 공간을 가상 주소를 통해 접근하게 할 수 있게 된다. 구체적으로 사용자가 특정 물리 공간을 접근하기를 원하면 이 물리 공간에 사상되는 테이블들을 운영체제에서 생성하고 이 테이블의 인덱스로 사용할 수 있는 가상 주소를 복귀해주면, 응용은 이 복귀 주소를 이용하여 원하는 물리 주소를 접근할 수 있다.

3.3 가상화 커널을 위한 다중 커널 이미지 생성 설계

가상화 커널을 구현하기 위해서는 서로 다른 임의의 메모리 공간에서 동작 가능한 커널을 생성해야 한다. 하지만 리눅스 커널은 정적으로 위치가 정해지며 예약된 영역만을 사용한다. 또한 리눅스 가상 메모리 구조의 설계상 커널 텍스트는 물리 주소 0×0번지부터 시작하여 40Mbyte 내에서만 동작 할 수 있도록 정해져있다. 이러한 설계상 제한에 대하여 가상화 커널에 적합하도록 재설계 하였다.

일반적으로 응용의 수행 메모리 영역은 운영체제, 로더, 링커 등의 시스템 소프트웨어에 의해 결정된다. 구체적으로 링커가 응용의 시작 주소를 설정하며 로더가 시작 주소의 메모리 위치로 응용을 올리고 운영체제가 응용의 시작 주소로 분기한다. 그런데 운영체제의 수행 영역은 링커와 함께 운영체제 자신의 수행 코드 및 부트 로더에 의해 결정된다. 구체적으로 링커가 운영체제의 시작 주소를 설정하며 부트 로더가 운영체제를 설정된 위치로 올린 후 분기한다. 이 후 운영체제 자신이 직접 생성한 페이지 테이블을 이용하여 동작한다.

위에서 언급한 바와 같이 리눅스 커널은 메모리 하위 영역만 상주가 가능하며 컴파일시 선택되는 구성 요소에 따라 상주하는 영역의 크기가 정해지도록 설계 되어 있다. 기본적으로는 2Mbyte 영역에서부터 시작하여 5Mbyte정도의 공간을 차지하게 된다. 0~1Mbyte 사이

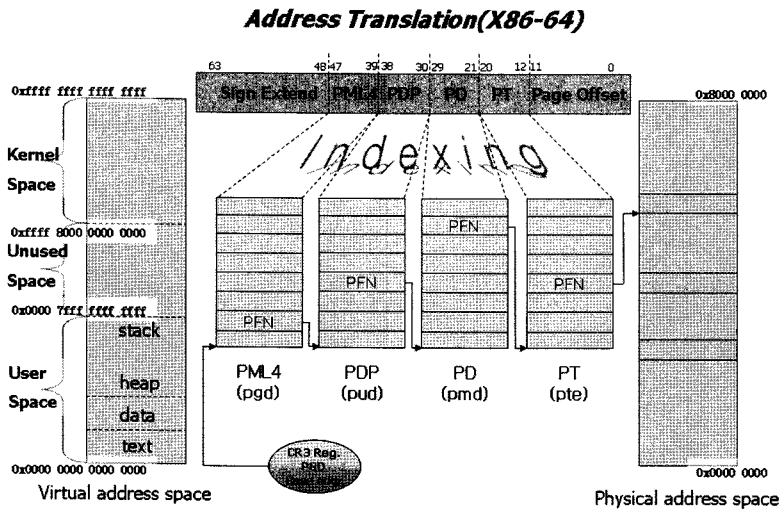


그림 4 물리 주소와 가상 주소

에는 커널 스택을 위한 페이지 테이블, I/O remap, 비디오 맵 등의 정보가 존재하며, 1~2Mbyte사이의 공간은 커널에서 부팅 초기에 사용되는 스택이 존재한다. 한편, 커널이 상주하는 5Mbyte정도의 공간은 커널의 코드와 데이타를 위한 영역이다.

커널을 메모리 임의의 영역에서 동작 가능하도록 수정하기 위해서는 운영체제 자신과 로더 스크립트, 그리고 로더에서 커널로 분기하는 부분에 대한 변경과 함께 가상 메모리의 설계도 수정되어야 한다. 구체적으로 현재는 커널이 상주할 수 있는 물리 메모리 공간이 40MB로 제한이 되어있고 이로 인해 커널이 재배치될 수 있는 영역은 40MB 이내로 제한된다. 그렇기 때문에 커널이 메모리 임의의 영역에 존재하기 위해서는 가상 메모리의 커널영역 중 커널 텍스트가 매핑되어 있는 영역과 모듈이 매핑 되어 있는 영역사이에서 공간 배분에 대한 설계를 가상화 커널에 적합하도록 재설계해야 하고, 커널을 위한 페이지 테이블을 확장해야 한다.

3.4 메모리 격리성 설계

가상화 커널의 구현을 위해서는 운영체제 사이에 서로의 간섭 없이 독립적인 메모리 사용이 가능해야한다. 또한 효과적인 메모리 테스트를 위해서 물리 메모리 영역은 운영체제와 테스트 프로그램 및 응용이 수행되는 영역과 메모리 테스트 가능 영역으로 구분되어야 한다. 일반적으로 메모리 테스트 과정은 미리 정의된 패턴으로 특정 내용을 메모리에 쓰고, 그 내용을 다시 읽어서 결과가 동일한지 비교하는 과정으로 구성된다. 결국 메모리 테스트는 메모리에 대한 쓰기 연산을 수행하는데, 만약 메모리 테스트 영역을 운영체제나 응용이 사용하고 있다면 시스템 오류나 응용의 예외적인 종료가 발생하게 된다. 따라서 메모리 테스트 영역과 운영체제나 응용이 사용하는 메모리 영역은 명확하게 구분되고 보호되어야 한다. 이러한 문제점들을 해결하기 위해 메모리 격리성을 설계하였다.

리눅스에서는 '버디 시스템(Buddy system)' 알고리즘을 이용하여 물리 메모리를 관리하고 있다. 버디 시스템은 모든 물리 메모리를 페이지 프레임(Page frame) 단위로 분할하고 이중 사용되지 않는 페이지 프레임을 그룹별로 묶어서 11개의 각기 다른 크기의 그룹으로 관리한다. 구체적으로 각각의 그룹은 20~210개의 연속된 페이지 프레임들로 구성된다. 리눅스에서 버디 시스템은 부팅 시 초기화되며 이후 버디의 정책에 따라 효율적으로 관리된다. 구체적으로 응용이 새로운 공간을 요청하거나(malloc()) 커널이 새로운 동적 메모리 공간을 요청하면(kmalloc()), 이 요청은 버디 시스템으로 전달되며 결국 버디 시스템이 관리하는 가용 메모리에서 적절한 공간을 할당(alloc_pages())하게 된다. 또한 사용하던 메모리 공간의 반납은 결국 버디 시스템에게 사용하던 공간을 반납(free_pages())하는 것이 되며, 버디 시스템 내부에는 버디의 정책에 따라 반납된 공간과 기존의 가용 공간 간에 통합이 수행된다. 그림 5는 버디 시스템의 구조를 도시한 것이다.

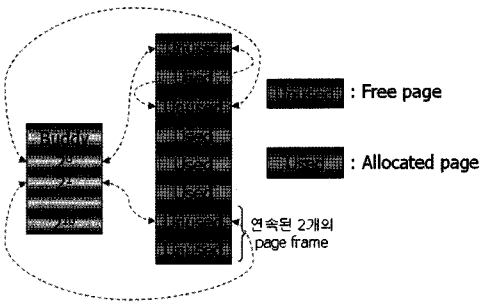


그림 5 리눅스 커널 2.6.18 물리 메모리 관리 기법

본 연구에서는 버디 시스템을 활용하여 메모리 격리성을 설계하였다. 구체적으로 부팅 시 버디 시스템을 초기화하는 과정에서 물리 메모리를 메모리 테스트 영역과 운영체제가 사용하는 영역으로 구분하고 이에 따른 접근제어로 서로의 영역을 격리하였다.

3.5 커널 하이버네이션 설계

커널 하이버네이션이란 가상화 커널 간의 유연한 문맥교환을 위하여 현재 수행중인 커널을 중지(suspend)하고 기존에 중지 되었던 커널을 재개(resume)하는 기법이다. 이것은 전체 메모리 모듈에 대한 테스트를 리부팅 없이 진행하기 위한 것이다. 즉, 현재 수행중인 운영체제로 인하여 테스트 되지 못한 영역에 대한 검증을 리부팅 없이 진행하기 위한 것이다.

하이버네이션은 개념적으로 마치 응용을 중지하고 재개하는 것과 비슷하다. 하지만 응용 하이버네이션의 경우 문맥을 저장하는 등의 모든 처리를 운영체제가 담당하여 관리하고 커널 하이버네이션의 경우는 운영체제 자신이 직접 담당해야만 한다는 점에서 응용보다 훨씬 복잡하고 민감하게 동작한다. 이는 사전에 커널간의 메시지 전송을 통해서 각자의 정확한 위치와 사용 중인 메모리의 크기 등을 파악한 후 적절한 동기화를 통하여 오차 없이 수행되어야 한다.

커널 하이버네이션은 메모리의 문맥과 프로세서의 문맥 그리고 이와 함께 논리적으로 구성되는 커널의 문맥도 고려되어야 한다. 또한 프로세서의 문맥이 중지되는 시점에서의 메모리 문맥과 커널 문맥을 보존하고 이를 동기화 하여 중지함으로서 일관성 있는 문맥을 구성해야만 한다. 한편, 다중 프로세서 지원을 위해 커널의 중지 시 모든 프로세서들 간에 동기화가 이루어 졌음을 보장해야 한다. 그리고 커널 잠금(locking)과 캐시의 초

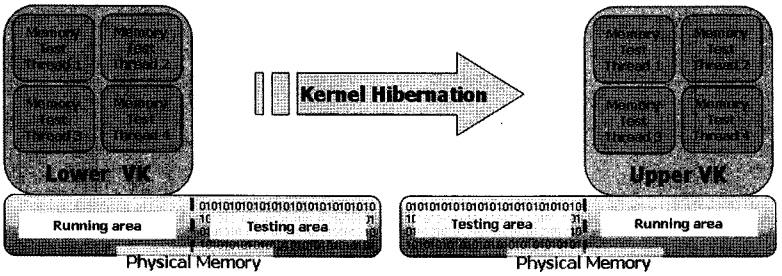


그림 6 커널 하이버네이션 동작 과정

기화 등을 적절한 시점에서 실행해야만 한다. 커널이 재개하는 시점에서도 중지하는 시점과 마찬가지로 동기화 이루어 져야 한다. 즉, 재개 시점에서는 모든 프로세서들이 동시에 문맥교환을 수행하도록 보장되어야 한다. 만약 이러한 규정에 어긋날 시 시스템에 심각한 장애를 초래할 수 있다.

그림 6은 커널 하이버네이션의 동작 과정을 보여주고 있다. 이것은 우선 (a)의 Lower VK와 같이 물리 메모리 하위 영역에서 동작하는 커널에 의하여 메모리 상위 영역에 대한 테스트를 진행한다. 이후 테스트가 진행된 상위영역에서 동작하는 (b)의 Upper VK와 같은 커널을 재개하고 재개된 커널로 분기한다. 그리고 Lower VK가 존재하던 즉, 테스트가 진행되지 못하였던 공간에 대해 Upper VK에 의하여 테스트를 진행한다. 이와 같이 하이버네이션 기법을 통해 전체 물리 메모리를 리부팅 없이 테스트 한다.

4. 가상화 커널의 구현

4.1 메모리 테스트를 위한 물리 메모리 직접 접근 구현

본 논문에서 물리 메모리를 직접 접근하는 인터페이스는 설계부에서 설명된 바와 같이 페이지 테이블들을 구성하고 사용자가 원하는 물리 주소를 참조 할 수 있는 주소를 가상 주소로 응용에게 복귀하는 형태로 구현 가능 하다. 하지만 이미 리눅스 커널 2.6.18버전의 X86-64아키텍처에서는 이러한 테이블이 구현되어 있으므로 본 논문에서는 기존의 테이블을 이용하였다.

그림 7은 X86-64를 위한 리눅스 커널의 가상메모리 구조를 보여주고 있다. 가상 메모리는 크게 사용자 공간과 커널 공간으로 구분된다. 현재 X86-64 CPU는 상위 16비트를 sign extension으로 무시하고 있다. 결국 사용자 공간은 0x0~0x0000 7fff ffff ffff, 커널 공간은 0xffff 8000 0000 0000~0xffff ffff ffff ffff이 된다. 구체적으로 사용자 공간은 text, data, stack, heap영역으로 구분되며, 커널 공간은 kernel image와 모듈이 동작하는 kernel영역, ioremap 영역, direct mapping 영역

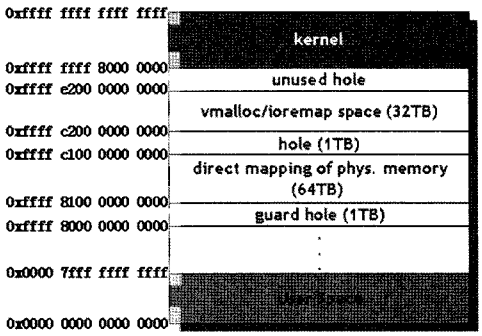


그림 7 X86-64 가상 메모리 구조

으로 구분된다. 여기서 사용자 공간과 커널 공간 사이에 아주 거대한 빈 영역이 존재하는데 이것은 아직 사용되지 않는 부분이다. 커널 공간 중 direct mapping 영역이 가상 주소와 물리 주소가 일대일로 사상되도록 만들어진 영역이다. 이 테이블을 사용하면 원하는 물리 주소를 접근할 수 있다.

그런데 사용자가 예상하는 물리 주소와 커널이 제공하는 물리 주소 간에 약간의 차이가 존재한다는 문제가 발생한다. 이 차이는 BIOS에서 특정 메모리 공간과 주소 영역을 예약해두고 사용하기 때문이며, 커널은 이를 고려하여 물리 메모리를 관리하기 때문에 커널 관점에서는 물리 메모리 공간의 불연속성이 야기된다. 그림 8은 커널 관점에서 메모리 주소 공간을 나타내는 e820 메모리 맵이다. 그림에서 usable로 표시된 부분은 물리 메모리가 존재하는 부분이며 reserved로 표시된 부분은

```
/*
 * E820 Memory Map
 */

BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)    //      0.62Mbyte
BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
BIOS-e820: 0000000000100000 - 00000007faf9000 (usable)    //  2041.97Mbyte
BIOS-e820: 00000007faf9000 - 00000007fb1d000 (reserved)
BIOS-e820: 00000007fb1d000 - 00000007fb4a000 (usable)    //      0.17Mbyte
BIOS-e820: 00000007fb4a000 - 00000007fb9d000 (ACPI NVS)
BIOS-e820: 00000007fb9d000 - 00000007fc00000 (ACPI data)
BIOS-e820: 00000000ffe00000 - 0000000100000000 (reserved)
```

그림 8 물리 메모리 2GB 기준 e820 memory map

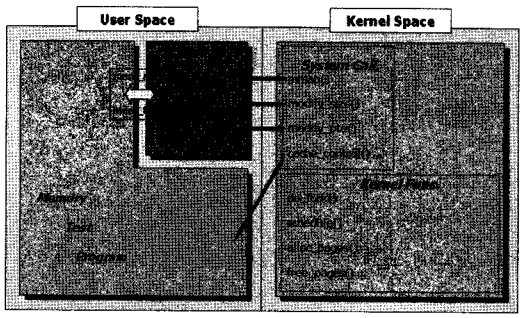


그림 9 메모리 테스트 인터페이스

비디오 카드, ACPI 제어 등을 위해 BIOS가 예약해 둔 부분으로 실제 물리 메모리는 연결어 있지 않다. 한편, 전체 주소 공간 중에는 0xa00000~0xf00000과 같이 어느 곳에도 사상되어 있지 않은 hole이 존재한다.

이러한 문제를 해결하기 위해 본 논문에서는 실제 물리 메모리(physical view)와 커널이 관리하는 메모리(kernel view) 사이에 새로운 추상화 계층을 추가하였다. 이 층은 kernel view 중에서 reserved와 hole을 고려하여 이 부분을 사용자에게 가리는 기능을 제공한다. 결국 메모리 테스트 프로그램 개발자는 연속된 물리 메모리 구조만을 고려하며 프로그램을 작성할 수 있다. 추상화 계층은 메모리 테스트 라이브러리와 시스템 호출의 추가를 통해 구현되었다. 그림 9는 추상화 계층이 추가된 메모리 테스트를 위한 물리 메모리 할당 방법을 도시하고 있다. 사용자 프로그램은 라이브러리가 제공하는 mem_alloc 인터페이스를 이용해 물리 메모리 중 테스트 영역에 대한 시작 주소와 크기를 요청한다. mem_alloc은 내부적으로 4단계에 걸쳐 메모리를 할당한다. 이 함수는 우선 /dev/mem을 open한다. /dev/mem은 메모리 공간에 접근할 수 있는 장치 파일로서 리눅스에서 제공된다. 그리고 reserved와 hole을 고려하여 매핑해야 할 크기를 구하기 위해 새로 추가한 시스템 호출 sys_modify_size를 호출한다. 이후 실제 매핑해야 할 크기를 인자로 리눅스에서 제공하는 시스템 호출인 mmap을 호출한다. mmap의 결과 사용자 주소 공간에 direct mapping에서 복사한 주소 변환 테이블들의 내용이 추가되는데, 여기에는 reserved와 hole 공간도 포함되어 있다. 사용자 프로그램이 연속적으로 메모리를 접근하려면 이 공간을 주소 변환 테이블에서 없애야 하는데, 이 역할을 새로 추가한 시스템 호출 sys_modify_pte가 담당한다. 이후 mem_alloc 인터페이스의 복귀 값은 테스트 공간 시작 가상 주소인 base_addr이다. 결국 테스트 하고자하는 물리 메모리로의 접근만 허용된다. 테스트가 완료되면 mem_free를 호출하여 테스트 공간을 반납한다. 이때 base_addr을 인자로 사용하며 결국 mem_alloc에서 내부적으로 할당해 놓았던 자료 구조와 사용한 메모리를 반납한다.

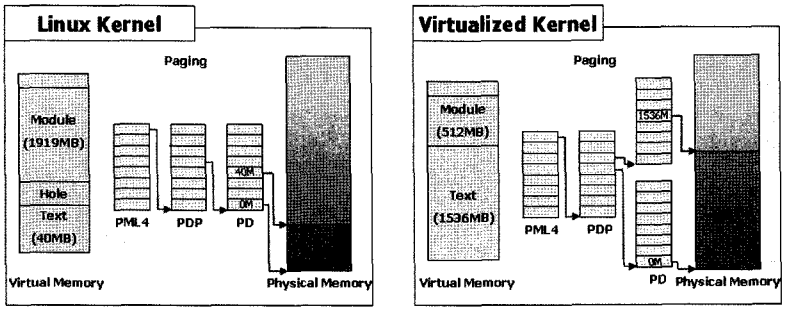
한편, 메모리 테스트에는 캐시모드를 설정할 수 있는 방법이 요구되는데 이것은 시스템 호출인 cache_control()함수로 구현하였다. 이 함수는 내부적으로 페이지 테이블의 비트를 수정함으로서 5가지의 캐시모드를 설정할 수 있다. 구체적으로 캐시 모드는 write back, write combine, write through, write protected, uncache모드로 구별된다.

4.2 가상화 커널을 위한 다중 커널 이미지 생성 구현

설계부에서 설명된 바와 같이 가상화 커널이 동작하기 위해서는 메모리 임의의 위치에서 수행 가능한 커널이 생성되어야 한다. 이를 위해 커널은 40MB의 제한을 넘어 메모리 상위영역에서도 동작 가능해야 한다. 커널을 메모리 상위 영역에서 동작하도록 수정하기 위해서는 운영체제 자신과 로더 스크립트, 그리고 로더에서 커널로 분기하는 부분에 대한 변경이 필요하다. 구체적으로 변경이 필요한 내용과 관련 파일은 다음과 같다.

- 가상 메모리 구조 수정: 커널 텍스트와 모듈 공간의 재설계
  - include/asm-x86_64/page.h
  - include/asm-x86_64/pgtable.h
- 커널 시작점 위치 수정: 기존 커널 시작 주소를 원하는 상위 메모리 위치로 수정.
  - include/linux/bootmem.h
- 커널을 위한 page table 수정: 기존 커널 페이지 테이블은 0MB부터 40MB내에서만 커널이 동작할 수 있도록 작성되었음. 이를 1536MB까지로 확장.
  - arch/x86_64/kernel/head.S
- 압축 해제 공간으로 분기하는 부분 수정: 기존 커널은 부트 로더 수행 후 커널 압축 푸는 루틴이 커널을 2MB 위치에 풀고 이 위치로 분기. 분기하는 위치를 상위 메모리 위치로 수정. 리눅스 커널 2.6.16 버전 이후 매크로로 작성되었음.
  - arch/x86_64/boot/compressed/head.S

그림 10은 커널이 메모리 임의의 영역에서 동작할 수 있도록 재설계된 구조를 보여준다. 구체적으로 0MB~40MB사이의 가상메모리 공간에 매핑 되어있던 커널의 위치를 0MB~1536MB사이의 가상메모리 공간으로 확장 매핑 함으로서 물리 메모리에서도 0MB~1536MB사이 어느 위치에서나 수행 가능하도록 하였다. 이것은 가상 메모리 구조의 변경과 커널을 위한 페이지 테이블 수정에 의해 구현된다. 즉, 가상 메모리 구조를 변경함으로서 커널의 매핑 공간을 확장하였고 페이지 테이블을 수정함으로서 확장된 매핑 공간이 실제 물리 메모리를 가리킬 수 있도록 하였다. 이 후 커널은 컴파일을 할 때



(a) Original kernel (b) Virtualized kernel

그림 10 메모리 임의의 위치에서 동작 가능한 운영체제 구현

구성 옵션으로 원하는 시작 위치를 지정해 줌으로서 1536MB내의 임의의 위치에서 동작하게 된다.

4.3 메모리 격리성 구현널 이미지 생성 구현

본 논문에서는 응용이나 운영체제가 사용하는 공간과 메모리 테스트 공간을 구분하고 보호하기 위하여 버디 시스템이 관리하는 메모리 크기를 제한하였다. 이를 위해 수정된 파일은 다음과 같다.

- 가용 메모리 공간 제한: 부팅 시 버디 시스템을 초기화할 때 가용메모리 공간을 제한
  - mm/bootmem.c
- 초기화 시 사용되는 메모리 공간 제한: 버디 시스템 초기화 전 커널에 의해 사용되는 스택 영역의 제한 (버디 시스템 초기화 이후 커널 스택은 버디로부터 할당됨)
  - include/linux/bootmem.h
- 구성 파일에 따른 커널 위치 저장 및 사용 공간 설정: 컴파일시 옵션으로 지정되는 커널의 위치를 참조하여 사용될 메모리 영역 지정
  - include/linux/bootmem.h

그림 11은 기존 리눅스와 가상화 커널의 메모리 사용면에서 차이점을 스냅샷 덤프(snapshot dump)를 이용하여 도시하고 있다. 그림에서 (a)는 기존 리눅스에서 메모리가 어떻게 사용되고 있는지를 도시한 것이다. 반면, (b)는 (a)와 다르게 버디 시스템이 관리하는 메모리 영역을 0~1GB로 제한한 것을 도시한 것이다. 결국 응용이나 커널의 새로운 메모리 공간 요청은 0~1GB내에서 이루어진다. 따라서 응용이나 운영체제가 사용하는 영역은 0~1GB로 제한되며, 1~2GB 메모리 공간은 테스트 가능한 공간이 된다. 한편, 상위에서 동작하는 upper kernel의 경우 버디 시스템이 관리하는 메모리 크기를 1~2GB로 제한하게 된다. 결국 응용이나 운용체제는 1~2GB 영역에서만 동작하게 되며 0~1GB 영역은 테스트를 위한 공간으로 할당된다.

메모리 격리성은 가상화 커널간의 메모리 보호에서도

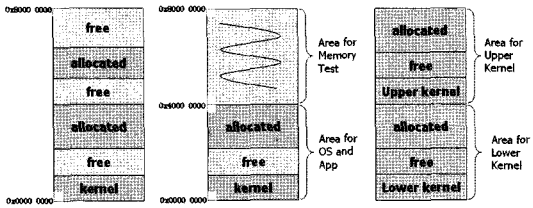


그림 11 메모리 격리성

유용한 역할을 해준다. 그림 11의 (c)는 가상화 커널에서의 메모리 격리성을 도시하고 있다. 각각의 가상화 커널은 자신만의 버디 시스템을 따로 구성하며 자신의 버디에 등록된 페이지에서만 즉, 커널이 사용할 수 있는 가용 메모리에서만 할당과 반납을 수행한다.

버디 시스템을 이용한 메모리 격리성은 부팅 초기에 페이지 프레임을 선택적으로 버디에 등록함으로서 가능하다. 구체적으로 버디는 첫 번째 페이지 프레임부터 마지막 페이지 프레임까지 free_pages()인터페이스를 이용하여 버디 시스템에 등록하게 된다. 이 과정에서 버디 시스템에 등록하기 전에 커널의 위치를 파악하고 그 위치에 따라 등록할 첫 번째 페이지 프레임과 마지막 페이지 프레임을 설정하게 된다. 결국 설정된 페이지 프레임 사이에 있는 페이지만 버디 시스템에 등록되며 나머지 페이지들은 메모리 테스트를 위한 공간으로 남게 된다.

4.4 커널 하이버네이션 구현

커널 하이버네이션은 다음 2가지 인터페이스로 구현된다.

- kernel_suspend: 현재 수행 중인 커널의 상태를 저장한다.
- kernel_resume: 저장되어 있던 커널의 상태를 복구하고 수행을 재개한다.

그림 12는 kernel_suspend의 처리 내용을 의사 코드(pseudo code)로 기술한 것이다. 코드를 간략히 살펴보면 흐름은 다음과 같다. 우선 모든 CPU는 kernel_sus-

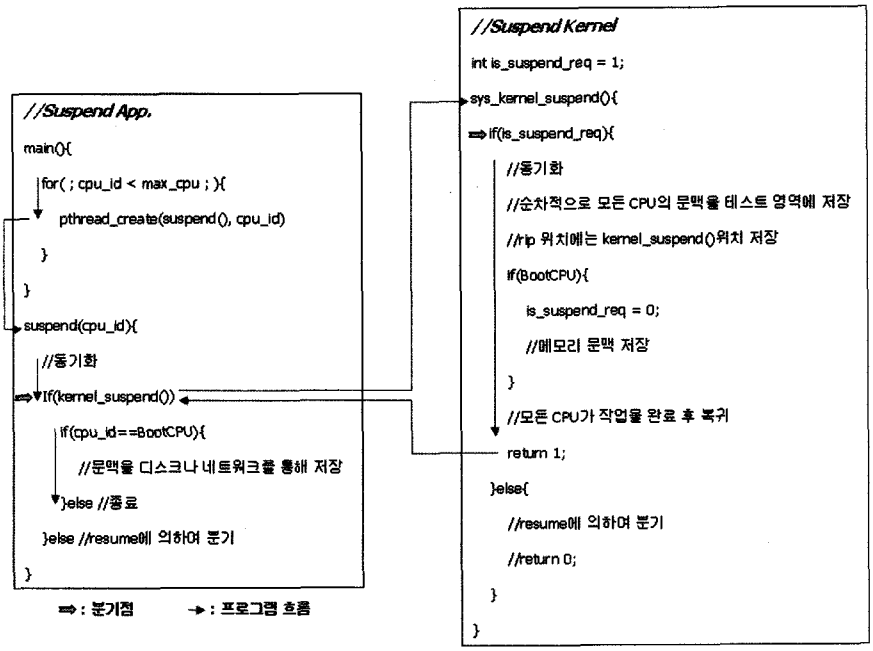


그림 12 커널 중지 의사 코드

pend라는 시스템 호출을 통해 커널로 진입한다. 진입 직후 is_suspend_reg하는 변수를 확인하는데 이 값은 1로 초기화 되었으므로 if문 내부, 즉 실제 중지를 진행하는 코드로 분기한다. 분기한 후 각각의 CPU는 자신의 문맥을 저장하는데 Boot CPU는 자신의 문맥과 함께 메모리 문맥도 저장한다. 이때 현재 수행중인 코드의 메모리 위치를 저장하고 변수 is_suspend_reg 값을 0으로 변경하여 중지한다. 이 결과 중지되었던 이미지가 재개할 때는 if문에서 참 조건을 만족하지 못하고 else 부분이 수행되게 된다. 한편, 일관성을 보장하기 위해 모든 CPU들은 작업을 완료할 때까지 커널모드에서 대기해야한다. 커널 모드에서의 작업이 모두 완료되면 1이라는 복귀 값을 가지고 유저 모드로 전환하게 된다.

중지된 커널을 재개하는 과정은 그림 13에 기술되어 있다. 그림에서와 같이 우선 중지되어 파일로 저장된 이미지를 메모리로 읽어 메모리 문맥을 복구한다. 메모리 문맥이 복구되면 kernel_resume이라는 시스템 호출을 통해 커널로 진입하게 된다. kernel_resume내에서는 동기화 후 저장되어있던 kernel_suspend의 주소를 이용하여 kernel_suspend로 분기한다. 이후 모든 메모리 참조는 이전에 중지되었던 메모리 문맥에서 수행된다. kernel_suspend로의 분기가 성공하면 각각의 CPU는 자신의 문맥을 복구하고 중지 시 수행되었던 응용으로 복귀하게 된다. 이때 복귀 값은 0이며, suspend 할 때의 복귀 값 1과 구별된다.

5. 실험 결과

대본 연구진은 구현된 가상화 커널을 실제 메모리 테스트에 이용해 보았다. 실험 환경은 표 1과 같다. 시스템은 2개의 Xeon dual Core 프로세서를 사용하며, Xeon은 Hyper-threading을 제공하기 때문에 논리적으로 8개의 CPU가 인식된다. 그리고 현재 시스템에는 1GB크기의 FBDIMM메모리 2개를 탑재하여 총 2GB메모리를 사용하고 있다. 테스트 패턴은 bit fade를 사용하였다. bit fade는 메모리에 테스트 패턴을 기록한 후 일정 시간이 경과하였을 때 그 값이 변경되지 않았는지를 확인하는 방식으로 진행된다. 이외에도 Address test, Moving inversionse 등과 같은 테스트를 진행하였으며 물리 메모리 전체 영역을 테스트하는데 적합함을 확인하였다.

표 1 실험 환경

항목	세부 내역
CPU	Intel Xeon 64-bit dual core 2.8GHz with hyper-threading × 2ea (Total 8 CPUs)
Memory	1GByte Samsung FBDIMM × 2ea (Total 2GByte)
VGA Card	ATI mach64
NIC	Linksys 100M
배포판 OS	Fedora Core 4
커널 버전	Linux-2.6.18(x86-64)
테스트 패턴	Bit fade

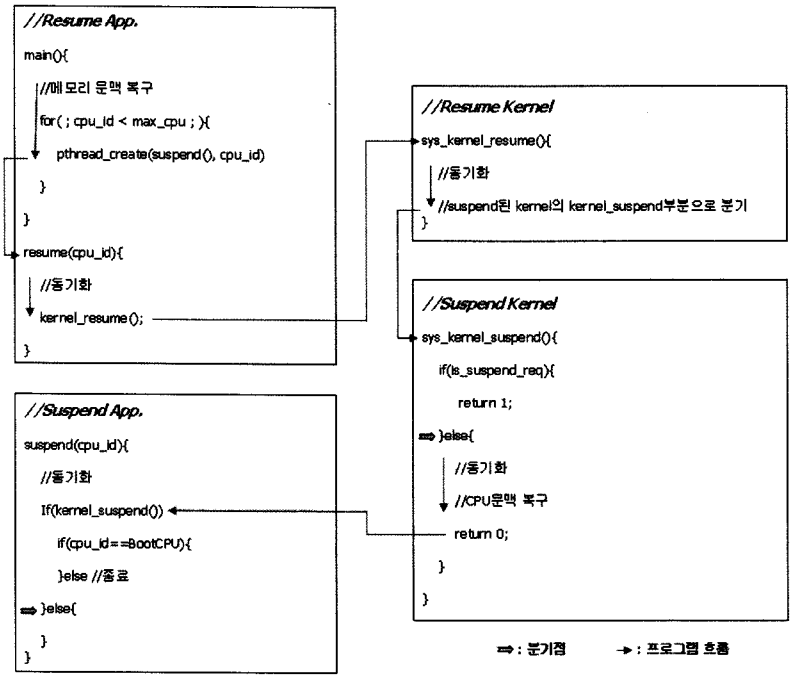


그림 13 커널 재개 의사 코드

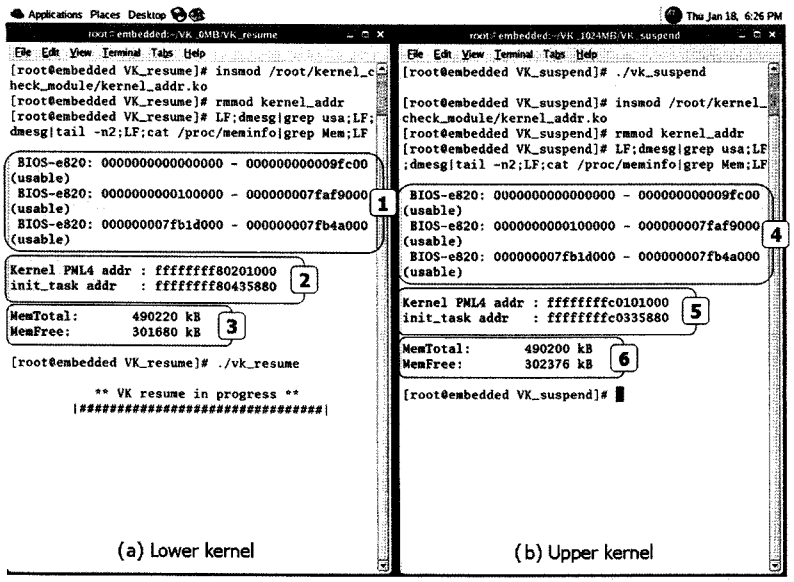


그림 14 Memory Isolation

메모리 테스트를 진행하기 위해서는 메모리 격리성이 보장되어야 한다. 그림 14는 이러한 메모리 격리성에 대한 검증을 해주는 실험결과 이다. ①과 ④는 BIOS에서 제공하는 물리 메모리 정보이다. 그림에서 알 수 있듯이 시스템에 탑재된 물리 메모리 전체 크기는 2GB이다. 그리고 ③과⑥의 정보를 보면 2GB중 490220KB정도 즉 490MB정도가 버디 시스템에 의하여 관리되고 있음을 알 수 있다. 실제로 운영체제에서 사용하는 메모리는 512MB이지만 490MB정도만 관리되는 이유는 커널의 크기와 부팅 시 커널에서 사용하는 스택의 크기를 제외

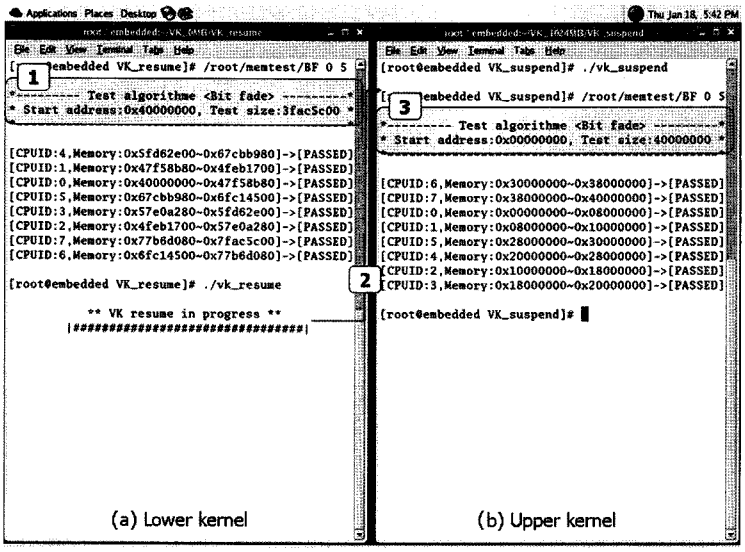


그림 16 Kernel Resume과 Memory test

```
[root@embedded VK_suspend]# ./vk_suspend

        ** VK suspend in progress **
      |##################################|

VK_start_addr:0x40000000, VK_end_addr:0x60000000
suspend_size:0x20000000 , VK_dump_area:0x100000

[root@embedded VK_suspend]#
```

그림 15 Kernel Suspend

하고 남은 부분을 버디 시스템이 관리하기 때문이다. 그림 7을 보면 커널의 가상주소는 0xffff ffff 8000 000부터 시작하는 것을 알 수 있다. 이 주소는 물리주소 0MB에 사상되어 있다. 따라서 Lower Kernel에서 PML4[2]와 init_task[3]의 주소 0xffff ffff 8020 1000과 0xffff ffff 8043 5880은 각각 물리 주소 2MB와 4.2MB에 사상된다. 한편, Upper Kernel에서 PML4와 init_task의 주소는 0xffff ffff c010 1000과 0xffff ffff c033 5880이다. 이는 물리주소 1025MB와 1027.2MB에 사상된다. 결국 각각의 커널은 약 0~512MB 그리고 1GB~1.5GB사이에 존재하고 있으며 이로 인하여 메모리 격리성이 보장되고 있음을 확인할 수 있다.

그림 15는 가상화 커널에서의 메모리 테스트 첫 번째 단계인 가상화 커널의 이미지를 중지하는 단계이다. 이 단계는 메모리 테스트 환경 설정 단계로서 한번만 수행하면 된다. 현재 그림 15에서 중지를 수행한 커널은 Upper Kernel로 0×40000000(1GB)~0×60000000(1.5GB) 사이에 존재한다. 이미지 생성 시간은 대략 30초정도가 소요되며 이것은 커널의 크기에 따라 가변적이다.

그림 16은 하위에서 동작하는 가상화 커널을 기반으로 상위영역에 대한 메모리 테스트를 수행한 후 상위에서 동작하는 커널로 재개하고 하위영역에 대한 메모리 테스트를 수행하는 그림이다. 이때 ①과 ③은 각각의 커널에서 메모리 테스트 영역을 나타내고 있다. ①을 보면 0×40000000(1GB)부터 시작하여 0×3fac5c00(≒1GB)크기만큼 테스트하며 ③에서는 0×0(0GB)에서 시작하여 0×40000000(1GB)크기만큼 테스트 한다. 테스트는 멀티코어 프로세서의 성능을 최대한 활용하기 위해 8개의 스레드를 생성하여 병렬적으로 진행하였다. ②는 문맥교환이 되는 시점을 나타낸다. 화살표와 같이 (a)에서 (b)로 분기하며 이후 커널은 (b)영역에서 동작하게 된다. (b)에서 화살표가 가리키는 지점 이전에 출력된 문자열은 중지시점에서 출력되었던 것이 복원된 것이다.

그림 15와 그림 16을 비교해 보면 그림 15에서 중지되었던 커널이 그림 16의 (a)에 의해 재개되어 (b)와 같이 다시 정상적으로 동작함을 확인할 수 있다. 또한 그림 16의 ①과 ③의 메모리 테스트 영역을 보면 대략 1GB씩을 나누어 하위에서 동작하는 가상화 커널과 상위에서 동작하는 가상화 커널에 의해 테스트가 수행되었음을 알 수 있다. 즉, 운영체제의 관리 하에 전체 물리 메모리가 효과적으로 테스트되었음을 확인할 수 있다.

6. 결론 및 향후 연구 계획

본 연구에서는 고성능화된 컴퓨팅 환경에 적응하기 위해 운영체제를 기반으로 메모리를 테스트 할 수 있는 가상화 커널을 설계, 구현하였다. 가상화 커널은 물리

2) PML4는 동작 중인 커널을 위한 페이지 디렉토리의 시작 주소이며 고정되어 있다.

3) init_task는 커널이 생성하는 첫 번째 프로세스이며 주소가 고정되어 있다.

메모리 직접접근, 메모리 격리성, 다중 커널이미지, 커널 하이버네이션 기법을 사용하여 구현되었으며 메모리 테스트에 효과적임을 확인하였다. 이때 효과적이라는 용어는 리부팅 없이 전체 물리 메모리를 테스트할 수 있음을 의미한다. 또한 기존 메모리 테스트 프로그램과 달리 사용자 친화적인 환경을 제공해 준다. 즉, GUI(Graphic User Interface)를 제공하고 파일 시스템이나 네트워크, USB 저장 장치 등을 사용할 수 있다.

하지만 가상화 커널의 안정화를 위해서는 아직 해결해야할 문제점이 남아 있다. 첫 번째는 파일시스템과 관련된 문제이다. 현재 시스템에서 루트 파일시스템(root filesystem)은 읽기 전용(read only)으로 설정하거나 램 디스크(ram disk)를 사용하여 구성하는 방식으로 구현하고 파일을 생성하거나 내용을 변경하기 위해서는 새로운 파티션을 마운트 하여 사용하여야한다. 이것은 재개 이후 커널의 문맥에 있는 파일 시스템의 상태와 실제 디스크의 파일 시스템의 상태가 같음을 보장하기 위해서다. 즉, 중지시의 파일시스템 상태가 변하지 않았음을 보장해야 한다. 만약 중지 전후의 디스크 상태가 같지 않다면 파일 시스템의 일관성이 깨지는 치명적인 상황이 발생된다. 두 번째는 리눅스에서 성능향상을 위해 제공되는 스왑(swap) 기능의 사용이 불가능하다는 점이다. 가상화 커널은 메모리의 문맥을 suspend해야 한다. 하지만 스왑 기능을 사용한다면 스왑 아웃(swap out)된 내용을 알 수 없기 때문에 전체 메모리 문맥을 중지 할 수 없다.

위의 문제점들은 다음과 같이 성능에 영향을 미친다. 첫 번째, 루트 파일시스템을 읽기 전용으로 설정한다면 새로운 패키지를 시스템에 설치하는 과정이 복잡해진다. 두 번째, 스왑기능을 사용한다면 운영체제에서 사용하는 메모리를 더 작게 설정할 수 있을 것이다. 이는 페스트 패턴 중 스크램블 기법을 사용한 메모리 테스트 등 메모리 영역의 크기에 영향을 받는 기법에 대해 이점이 갖는다. 가상화 커널의 성능 향상을 위해 가상화 커널을 위한 파일시스템에 대한 연구와 스왑 아웃된 메모리 내용을 추적할 수 있는 기법에 대해 연구를 진행할 것이다.

참 고 문 헌

[1] M.S. Abadir & H.K. Reghbati, "Functional testing of semiconductor random access memories," Computing Surveys, Volume 15, Number 3, pp. 175-198, 1983.

[2] Amandeep Singh, Debashish Bose, "A Software Based Online Memory Test For Highly Available Systems," IEEE IOLTS'05, pp. 199-200, 2005.

[3] 한선경, 유영갑, "Memory test의 問題와 展望," 전자공학회지, 제24권, 제6호, pp. 688-697, 1997.

[4] memtest86, http://www.memtest86.com

[5] Linux kernel archive, http://kernel.org

[6] Robert Love, Linux Kernel Development, 2nd Ed., Novell Press, Indianpolis, 2005.

[7] Daniel P.Bovet, Marco Cesati, Understanding the Linux kernel, 3rd Ed., O'REILLY, Sebastopol, 2005.

[8] Intel, Extended Memory 64 Technology Software Developer's Guide, 2004.

[9] AMD, AMD64 Architecture Programmer's Manual Vol.1-Vol.5, 2003.

[10] memtester, http://pyropus.ca/software/memtester

[11] Xen, www.xensource.com/

[12] VMware, www.vmware.com

[13] Vmware "Streamlining Software Testing with the IBM Rational and VMware," VMware white paper, 2003.

[14] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, Gernot Heiser, "Pre-virtualization : Slashing the Cost of virtualization," Technical Report, National ICT Australia, 2005.

[15] Mendel Rosenblem, Tal Garfinkel, "Virtual Machine Moniters: Current Technology and Future Trends," IEEE Computer Society, Volume 38, Number 5, pp. 39-47, 2005.

[16] Paul Barhan, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Lan Pratt, Andrew Warfield, "Xen and the Art of Virtualization," SOSP'03, pp. 164-177, 2003.

박 희 권

2005년 단국대학교 컴퓨터 과학과 졸업(이학사). 2007년 단국대학교 대학원 정보컴퓨터 과학과(이학석사). 2007년~현재 단국대학교 대학원 정보컴퓨터 과학과 박사과정. 관심분야는 운영체제, 내장형 시스템

윤 대 석

2007년 단국대학교 컴퓨터 과학과 졸업(이학사). 2007년~현재 단국대학교 대학원 정보컴퓨터 과학과 석사과정. 관심분야는 운영체제, 내장형 시스템

최 종 무

1993년 서울대학교 해양학과 졸업(이학사). 1995년 서울대학교 대학원 컴퓨터 공학과(공학석사). 2001년 서울대학교 대학원 컴퓨터 공학과(공학박사). 2001년~2003년 유비쿼스 주식회사 책임 연구원. 2003년~현재 단국대학교 정보컴퓨터학부 컴퓨터과학 전공 조교수. 2005년~2006년 UC Santa Cruz 방문 교수. 관심분야는 운영체제, 내장형 시스템, 고성능 저장 장치 등