3장. 메모리 관리 (Memory Management)

메모리 관리 서브시스템은 운영체제에서 가장 중요한 부분 중 하나이다. 초창기의 컴퓨터에 서부터, 시스템에 물리적으로 존재하는 것보다 더 많은 양의 메모리를 필요해왔다. 물리적 인 메모리의 한계를 극복하기 위한 여러 기법들이 개발되었는데, 가상 메모리 기법이 가장 성공적이다. 가상 메모리(virtual memory)는 메모리를 필요로 하는 서로 경쟁하는 프로세스 사이에 메모리를 공유하도록 하여, 시스템이 실제 가진 것보다 더 많은 메모리를 가진 것처 럼 보이도록 한다.

가상 메모리는 컴퓨터의 메모리를 늘리는 일만 하는 것은 아니다. 메모리 관리 서브시스템 은 다음과 같은 것을 제공한다.

- **넓은 주소공간** 운영체제는 시스템이 실제 가진 것보다 훨씬 많은 양의 메모리를 가지고 있 는 것처럼 보이게 한다. 가상 메모리는 시스템의 물리적 메모리보다 몇 배나 더 클 수 있다.
- **보호** 시스템의 각 프로세스는 각자의 독립된 가상 주소공간을 갖는다. 이들 가상 주소공간 은 서로 완벽하게 분리되어 있어서, 어떤 응용프로그램을 실행하는 프로세스는 다른 것 에 영향을 줄 수 없다. 또 하드웨어 가상 메모리 메커니즘은 메모리 영역에 쓰기를 금지 할 수 있게 한다. 이것은 코드와 데이터가 나쁜 프로그램에 의해 덮어 쓰여지는 것을 막 아준다.
- **메모리 매핑** 메모리 매핑은 이미지와 데이터 파일을 프로세스의 주소공간에 매핑하기 위해 사용된다. 메모리 매핑에서 파일의 내용은 프로세스의 가상 주소공간에 직접 연결된다.
- **공정한 물리적 메모리 할당** 메모리 관리 서브시스템은 시스템에서 실행중인 프로세스들이 서로 공정하게 물리적 메모리를 공유할 수 있게 한다.
- **공유 가상 메모리** 가상 메모리는 프로세스들이 분리된 (가상) 주소공간을 가질 수 있도록 하지만, 때로는 프로세스들이 메모리를 공유하는 것이 필요할 때가 있다. 예를 들어 시스 템에 명령쉘 bash를 실행하고 있는 여러 개의 프로세스가 있다고 하자. 각 프로세스의 가장 주소공간에 bash의 여러 복사본을 갖는 대신에, 물리적 공간에 하나의 복사본을 갖고 bash를 실행하는 모든 프로세스가 그것을 공유하는 것이 바람직하다. 동적 라이브 러리는 여러 프로세스가 실행 코드를 공유하는 대표적인 예이다.

  공유 메모리는 또한 두 개 이상의 프로세스가 그들 모두에게 공통적인 메모리를 통해 정 보를 교환함으로써, 프로세스간 통신(IPC) 메커니즘으로 사용될 수 있다. 리눅스는 유닉 스 시스템 V의 공유 메모리 IPC를 지원한다.

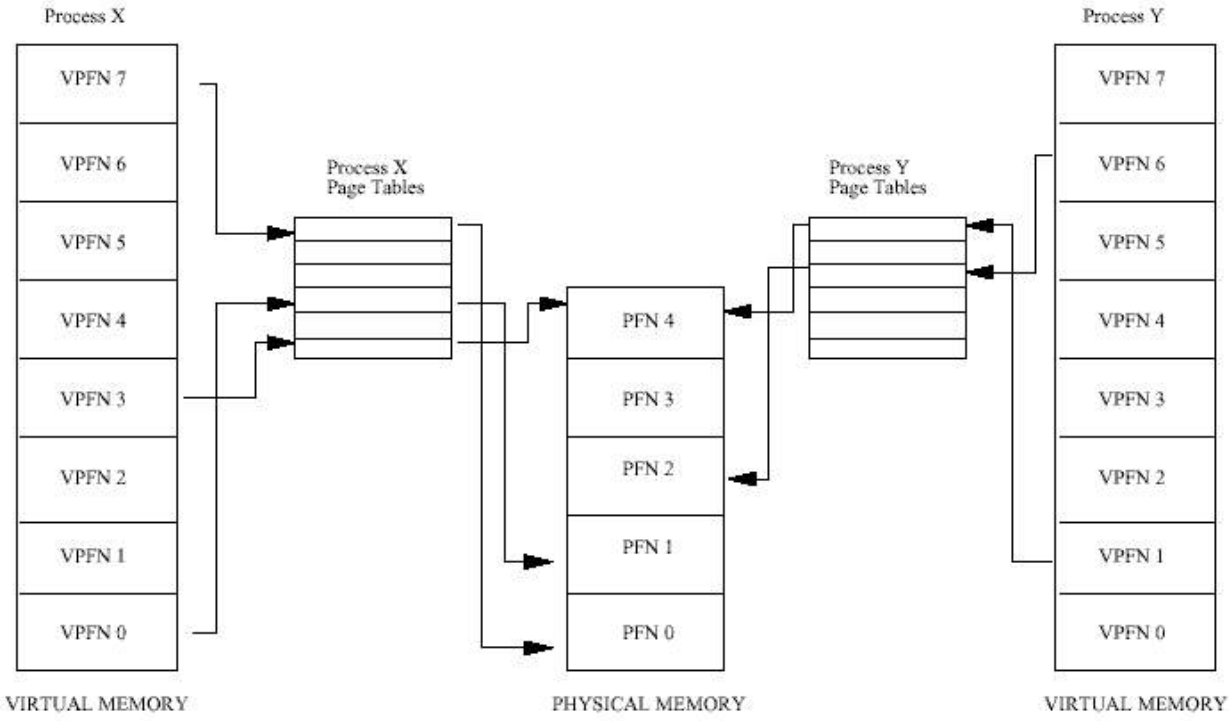

그림 3.1 : 가상 주소에서 물리 주소로 매핑의 추상적 모델

3.1 가상 메모리의 추상적 모델(abstract model)

리눅스가 가상 메모리를 지원하기 위해 사용하는 기법을 살펴보기 전에, 너무 자세히 파고 들어 혼란스럽지 않도록 먼저 추상적 모델을 검토하는 것이 도움이 될 것이다.

프로세서가 프로그램을 실행할 때, 프로세서는 메모리로부터 명령어를 읽어 와서 해석한다. 명령을 해석하는 데에는 메모리의 어떤 위치에 있는 내용을 가져오거나 저장해야 하기도 한 다. 프로세서는 명령어를 실행하고 프로그램의 다음 명령어로 이동한다. 이렇게 하여 프로세 서는 언제나 명령어를 가져오거나, 데이터를 가져오거나 저장하기 위해 메모리에 접근한다.

가상 메모리 시스템에서 이 주소들은 모두 물리적 주소가 아니라 가상 주소이다. 이 가상 주소들은 운영체제가 관리하는 테이블들에 저장된 정보를 바탕으로 프로세서에 의해 물리적 주소로 변환된다.

이 변환을 쉽게 하기 위해 가상 메모리와 물리적 메모리는 페이지라는 작은 조각으로 나뉜 다. 이 페이지들은 모두 같은 크기인데, 꼭 같은 크기일 필요는 없지만, 그렇지 않다면 시스 템을 관리하기가 무척 어려워질 것이다. 리눅스는 알파 AXP 시스템에서는 8KB 페이지를, 인텔 x86 시스템에서는 4KB 페이지를 사용한다[1]. 각 페이지에는 페이지 프레임 번호(page frame number, PFN)라는 유일한 번호가 부여된다. 이와 같은 페이지 모델에서 가상 주소는 가상 페이지 프레임 번호와 오프셋, 두 부분으로 이루어진다. 페이지 크기가 4KB라면 가상 주소의 0비트에서 11비트는 오프셋을 나타내고, 12번 비트 이상은 가상 페이지 프레임 번호 를 나타낸다[2]. 프로세서가 가상 주소를 처리할 때마다 오프셋과 가상 페이지 프레임 번호를 추출해야 한다. 프로세서는 가상 페이지 프레임 번호를 물리적 페이지 프레임 번호로 변환 하고 해당 물리적 페이지에서 오프셋에 해당하는 주소를 접근한다. 이렇게 하기 위해 프로세서는 페이지 테이블(page table)을 사용한다.

그림 3.1은 프로세스 X와 프로세스 Y 두 프로세스의 가상 주소공간과, 각자의 페이지 테이 블을 보여준다. 이 페이지 테이블은 각 프로세스의 가상 페이지를 메모리의 물리적 페이지 로 대응시킨다. 이 그림에서 프로세스 X의 가상 페이지 프레임 번호 0은 물리적 페이지 프 레임 번호 1로 대응되고, 프로세스 Y의 가상 페이지 프레임 번호 1은 물리적 페이지 프레임 번호 4로 대응된다. 이론적으로 페이지 테이블은 다음과 같은 정보를 가진다 :

- 유효 플래그. 이것은 페이지 테이블 엔트리가 유효한가를 나타낸다.
- 이 엔트리가 기술하는 물리적 페이지 프레임 번호.
- 접근 제어(access control) 정보. 이것은 페이지가 어떻게 사용될 수 있는지 기술한다. 데이터를 기록할 수 있는가? 실행가능 한 코드를 포함하는가?

페이지 테이블은 가상 페이지 프레임 번호를 오프셋으로 사용하여 접근한다. 가상 페이지 프레임 5는 테이블의 6번째 항목이 된다 (0이 첫번째 항목이다)

가상 주소를 물리적 주소로 변환하기 위해, 프로세서는 먼저 가상 주소 페이지 프레임 번호 와, 가상 페이지 안에서의 오프셋을 구해야 한다. 페이지 크기를 2의 제곱수로 하면, 이 계 산은 비트마스크와 쉬프트 연산으로 쉽게 처리할 수 있다. 다시 그림 3.1에서, 페이지 크기 가 0x2000바이트(8KB, 십진수로 8192)라면, 프로세서는 프로세스 Y의 가상 주소공간에서의 주소 0x2194를 가상 페이지 프레임 번호 1과 오프셋 0x194로 변환한다.

프로세서는 가상 페이지 프레임 번호를 인덱스로 프로세스의 페이지 테이블을 참조하여, 페 이지 테이블 엔트리(page table entry, PTE)를 가져온다. 이 페이지 테이블 엔트리가 유효하다 면, 프로세서는 이 엔트리에서 물리적 페이지 프레임 번호를 가져온다. 엔트리가 유효하지 않다면, 프로세스는 가상 메모리 공간에 존재하지 않는 영역을 접근한 것이다. 이 경우에 프 로세서는 주소를 결정할 수 없고 운영체제에 제어를 넘겨서 운영체제가 처리하도록 한다.

프로세서가 운영체제에게, 정확하게 어떤 프로세스가 유효한 변환을 할 수 없는 가상 주소 에 접근하려 했는지를 알리는 방법은 프로세서마다 다르다. 이것은 페이지 폴트(page fault)라 고 하며, 프로세서가 이를 어떻게 전달하든지 간에, 운영체제는 폴트가 발생한 가상 주소와 페이지 폴트의 원인을 통보받는다.

그 페이지 테이블 엔트리가 유효한 경우, 프로세서는 물리적 페이지 프레임 번호에 페이지 크기를 곱해서 물리적 메모리에서의 베이스 주소를 얻는다. 마지막으로 프로세서는 오프셋 을 더하여 필요한 명령이나 데이터에 도달한다[3].

위의 예를 다시 보면, 프로세스 Y의 가상 페이지 프레임 번호 1은 물리적 페이지 프레임 번 호 4에 대응되고, 0x8000(4 x 0x2000)에서 시작된다. 여기에 0x194 바이트의 오프셋을 더하면 최종적인 물리적 주소 0x8194를 얻을 수 있다.

이렇게 가상 주소를 물리적 주소로 대응시킴으로써, 가상 메모리는 시스템의 물리적 페이지 에 임의의 순서로 배열될 수 있다. 예를 들어, 그림 3.1의 프로세스 X의 가상 페이지 프레임 번호 0은 물리적 페이지 프레임 번호 1로 대응되는 반면, 가상 페이지 프레임 번호 7은 가 상 페이지 프레임 번호 0보다 높음에도 물리적 페이지 프레임 번호 0으로 대응된다. 이것은 가상 메모리의 재미있는 부산물을 보여준다. 가상 메모리의 페이지들은 물리적 메모리에 어 떤 특정한 순서로 존재하지 않아도 된다.

3.1.1 요구 페이징(Demand Paging)

실제로 가상 메모리보다 훨씬 적은 물리적 메모리만 있기 때문에, 운영체제는 물리적 메모 리가 비효율적으로 사용되지 않도록 주의해야 한다. 물리적 메모리를 절약하는 방법 하나는, 실행중인 프로그램이 현재 사용하는 가상 페이지만을 로드하는 것이다. 예를 들어, 데이터베 이스 프로그램이 데이터베이스에 질의를 한다고 하자. 이 경우 모든 데이터베이스가 메모리 에 로드될 필요는 없다. 검색할 데이터 레코드들만 있으면 된다. 데이터베이스 질의가 검색 질의라면, 데이터베이스 프로그램에서 새로운 레코드를 추가하는 것을 처리하는 부분의 코 드를 읽어들일 필요는 없을 것이다. 이렇게 가상 페이지들이 접근되는 경우에만 메모리에 읽어들이는 기법을 요구 페이징이라고 한다.

프로세스가 현재 메모리에 없는 가상 주소를 접근하려고 하면, 프로세서는 참조된 가상 페 이지에 대한 페이지 테이블 엔트리를 찾을 수 없을 것이다. 예를 들어, 그림 3.1에서 프로세 스 X의 페이지 테이블에는 가상 페이지 프레임 번호 2에 대한 엔트리가 없으므로, 프로세스 X가 가상 페이지 프레임 번호 2에 포함된 주소에서 읽으려고 하면, 프로세서는 그 주소를 물리적 주소로 변환할 수 없을 것이다. 이 시점에서 프로세서는 운영체제에게 페이지 폴트가 발생했다고 통보한다.

만약 폴트가 발생한 가상 주소가 유효하지 않은 것이라면, 그 프로세스는 접근할 수 없는 가상 주소에 접근하려고 한 것이다. 대체로 이건 메모리의 아무 주소에나 값을 쓰는 것처럼, 응용프로그램이 잘못된 것이다. 이 경우 운영체제는 이 프로세스를 종료시켜, 시스템의 다른 프로세스들을 이 잘못된 프로세스로부터 보호한다.

만약 폴트가 발생한 가상 주소가 유효한 것인데, 주소가 가리키는 페이지가 메모리에 현재 없다면, 운영체제는 해당하는 페이지를 디스크의 이미지로부터 메모리에 가져와야 한다. 디 스크 접근은 상대적으로 긴 시간이 걸리므로, 프로세스는 페이지가 도착할 때까지 한참을 기다려야 한다. 시스템에 실행할 수 있는 다른 프로세스가 있다면 운영체제는 이들 중 하나 를 선택하여 실행한다. 가져온 페이지는 빈 물리적 페이지 프레임에 기록되고, 가상 페이지 프레임 번호를 위한 엔트리가 프로세스의 페이지 테이블에 추가된다. 이제 프로세스는 메모 리 폴트가 발생했던 기계어 명령에서부터 재실행된다. 이번에 다시 가상 메모리 접근이 이 루어질 때, 프로세서는 가상 주소를 물리적 주소로 변환할 수 있게 되고, 프로세스는 계속 실행된다.

리눅스는 실행 이미지를 프로세스의 가상 메모리에 로드하기 위해 요구 페이징을 사용한다. 명령을 실행할 때마다, 명령을 포함하는 파일을 열고, 파일의 내용이 프로세스의 가상 메모 리로 매핑된다. 이것은 이 프로세스의 메모리 맵을 기술하는 자료구조를 변경하여 이루어지 며, 이를 메모리 매핑이라고 한다. 어쨌든 이미지의 첫번째 부분만 실제로 물리적 메모리에 가져오며, 나머지 부분은 디스크에 남아 있다. 이미지가 실행됨에 따라 페이지 폴트가 발생 하고, 리눅스는 프로세스의 메모리 맵을 사용하여 이미지의 어느 부분을 실행할 수 있도록 메모리에 가져올지 결정한다.

3.1.2 스와핑(Swapping)

프로세스가 가상 페이지를 물리적 메모리에 가져와야 하는데, 비어 있는 물리적 페이지가 없다면, 운영체제는 물리적 메모리에서 다른 페이지를 제거하여, 가져올 페이지를 위해 공간 을 마련해야 한다.

물리적 메모리에서 제거될 페이지가 이미지나 데이터 파일에서 온 것이고, 이 페이지에 쓰 여진 것이 없다면, 페이지의 내용을 저장할 필요는 없다. 대신 그냥 제거를 하고, 나중에 다 시 필요하게 되면 이미지나 데이터 파일로부터 다시 메모리에 읽어들이면 된다.

그러나 페이지가 변경되었다면, 운영체제는 페이지의 내용을 나중에 다시 사용할 수 있도록 보존해야 한다. 이런 페이지를 더티 페이지(dirty page)라고 하며, 이를 메모리에서 제거할 때 스왑 파일(swap file)이라는 특별한 파일에 저장한다. 스왑 파일에 접근하는 것은 프로세서나 물리적 메모리의 속도에 비해 매우 오래 걸리므로, 운영체제는 페이지를 디스크에 기록할 필요성과, 다시 사용될 수 있도록 메모리로 가져오게 될 필요성을 잘 다루어야 한다.

어떤 페이지를 제거 또는 스왑할지를 결정하기 위해 사용하는 알고리즘(스왑 알고리즘)이 효율적이지 않으면 쓰레싱(thrashing)[4]이라고 불리는 상태가 발생한다. 이 때 페이지는 계속 디스크에 기록되고 또 다시 읽어오게 되며, 운영체제는 너무 바빠서 실제 작업은 거의 못하 게 된다. 예를 들어 그림 3.1에서, 물리적 페이지 프레임 번호 1이 계속 접근된다면, 이것은 하드디스크로 스와핑할 좋은 후보가 아니다. 프로세스가 현재 사용하고 있는 페이지의 집합 을 작업 집합(working set)이라고 하는데, 효율적인 스왑 정책은 모든 프로세스들의 작업 집 합이 모두 물리적 메모리에 있도록 한다.

리눅스는 시스템에서 제거될 페이지를 공정하게 선택하기 위해, 가장 최근에 사용된(Least Recently Used, LRU) 페이지 수명(page aging) 기법을 사용한다. 이 기법에서 시스템의 모든 페 이지는, 그 페이지에 접근될 때마다 변경되는 수명을 갖고 있다. 페이지는 자주 접근될수록 젊어지고, 적게 접근될수록 나이가 들게 된다. 나이든 페이지는 스와핑의 좋은 후보이다.

3.1.3 공유 가상 메모리(Shared Virtual Memory)

가상 메모리는 여러 프로세스가 메모리를 쉽게 공유하게 해준다. 모든 메모리 접근은 페이 지 테이블을 통해서 이루어지며, 각 프로세스는 독립된 페이지 테이블을 갖고 있다. 두 개의 프로세스가 물리적 메모리의 페이지를 공유

하려면, 그 물리적 페이지의 프레임 번호가 두 프로세스의 페이지 테이블 모두에 페이지 테이블 엔트리로 있어야 한다.

그림 3.1은 두 프로세스가 물리적 페이지 프레임 번호 4를 공유하는 것을 보여준다. 이 물리 적 페이지는 프로세스 X 입장에서 가상 페이지 프레임 번호 4이고, 프로세스 Y 입장에서 가상 페이지 프레임 번호 6이다. 이것은 페이지 공유의 재미있는 점을 보여준다. 공유되는 물리적 페이지는 이 물리적 페이지를 공유하는 어떤 프로세스에서도 가상 메모리의 같은 위 치에 있을 필요가 없다.

3.1.4 물리적 주소 모드(Physical Addressing Mode)와 가상 주소 모드(Virtual Addressing Mode)

운영체제 자신이 가상 메모리에서 동작하는 것은 별 의미가 없다. 그렇게 되면 운영체제가 자신을 위해 페이지 테이블을 유지해야 하는 끔찍한 상황이 벌어질 것이다. 대부분의 범용 프로세서들은 물리적 주소 모드와 가상 주소 모드를 함께 제공한다. 물리적 주소 모드에서 는 페이지 테이블이 필요없으며, 이 모드에서 프로세서는 아무런 주소 변환도 하지 않는다. 리눅스 커널은 물리적 주소공간에서 실행되도록 링크되어 있다.

알파 AXP 프로세서는 특별한 물리적 주소 모드를 갖고 있지 않다. 대신에 메모리 공간을 여러 부분으로 나누어, 그 중의 두 개를 물리적으로 매핑된 주소로 지정해 둔다. 이 커널 주 소공간은 KSEG 주소공간이라고 부르며, 0xfffffc0000000000부터 위쪽 주소 전부를 포함한다. KSEG에 링크된 코드(정의에 따라 커널 코드이다)를 실행하거나 KSEG의 데이터를 접근하기 위해서는 코드는 반드시 커널 모드에서 실행되어야 한다. 알파에서의 리눅스 커널은 주소 0xfffffc0000310000로부터 실행되도록 링크되어 있다.

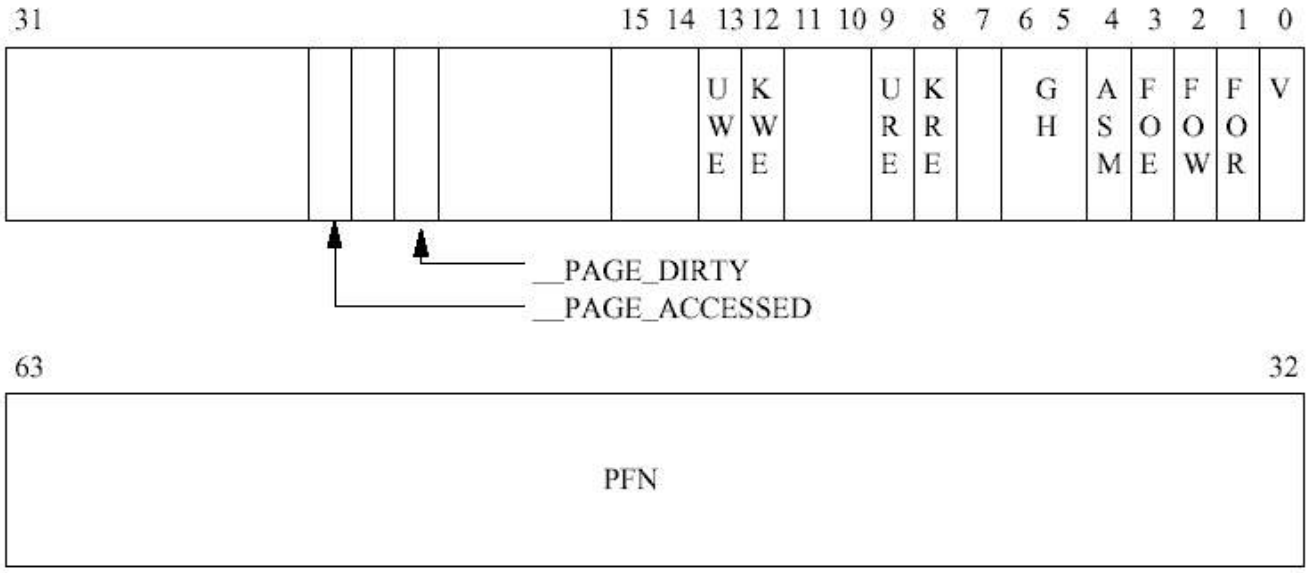

그림 3.2 : 알파 AXP 페이지 테이블 엔트리

3.1.5 접근 제어(Access Control)

페이지 테이블 엔트리는 접근 제어 정보도 가지고 있다. 프로세서는 프로세스의 가상 주소 를 물리적 주소로 변환하기 위해 이미 페이지 테이블 엔트리를 사용하기 때문에, 쉽게 접근 제어 정보를 사용하여, 이 프로세스가 허용되지 않은 방식으로 메모리를 접근하지 않도록 할 수 있다.

메모리 영역에 대한 접근을 제한하려고 하는 이유는 몇 가지가 있다. 실행 코드를 담고 있 는 곳 같은 어떤 메모리는 자연히 읽기 전용 메모리이며, 운영체제는 프로세스가 자신의 실 행 코드 위에 데이터를 쓰는 것을 허락해서는

안 된다. 반대로, 데이터를 담고 있는 페이지 는 쓰여질 수 있지만 그 메모리를 명령어로 간주하여 실행하려는 시도는 실패해야 한다. 대 부분의 프로세서는 적어도 두 가지 실행 모드 - 커널모드와 사용자모드 - 를 가지고 있다. 프로세서가 커널 모드로 수행중이 아니라면, 사용자가 커널 코드를 실행하거나 커널 자료구 조에 접근하는 것을 막고 싶을 것이다.

접근 제어 정보는 PTE에 들어있으며 프로세서마다 다르다. 그림 3.2는 알파 AXP 프로세서 의 PTE를 보여준다. 각 비트 필드의 의미는 다음과 같다 :

- **V** 유효(Valid) 이 페이지 테이블 엔트리는 유효함.
- **FOE** "실행시 오류(Fault on Execute)" 이 페이지의 명령을 실행하려고 할 때마다 프로세서는 페이지 오류를 발생하고 컨트롤을 운영체제에게 넘긴다.
- **FOW** "쓰기시 오류(Fault on Write)" 위와 같으나 실행대신 이 페이지로 쓰려고 할 때 오류 가 발생한다.
- **FOR** "읽기시 오류(Fault on Read)" 위와 같으나 이 페이지에서 읽으려 할 때 오류가 발생한 다.
- **ASM** 주소공간 매치(Address Space Match). 변환 버퍼에서 일부 엔트리만을 지우려고 할 때 사용된다.
- **KRE** 커널 모드에서 실행 중인 코드에서 이 페이지를 읽을 수 있음.
- **URE** 사용자 모드에서 실행중인 코드에서 이 페이지를 읽을 수 있음.
- **GH** 입도 힌트(granularity hint)는 블럭 전체을 여러개의 변환 버퍼 엔트리가 아닌 하나의 엔 트리에 매핑할 때 사용된다.
- **KWE** 커널 모드에서 실행 중인 코드가 이 페이지에 쓸 수 있음,
- **UWE** 사용자 모드에서 실행 중인 코드가 이 페이지에 쓸 수 있음, 페이지 프레임 번호 V 비트가 세트된 PTE의 경우 이 항목은 그 PTE의 물리적 페이지 프레 임 번호를 갖는다. 유효하지 않은 PTE의 경우, 항목의 값이 0이 아니라면 페이지가 스왑 파일 어디에 저장되어 있는지에 대한 정보를 갖고 있다.

리눅스는 다음 두 비트를 정의하여 사용한다:

- **_PAGE_DIRTY** 이 비트가 설정되어 있으면 페이지는 스왑 파일에 기록될 필요가 있다.
- **_PAGE_ACCESSED** 접근된 페이지를 표시하기 위해 리눅스가 사용한다.

3.2 캐시(Cache)

만약 위에서 언급한 이론적 모델을 사용하여 시스템을 구현한다면, 동작하기는 하겠지만 그 다지 효율적이지는 않을 것이다. 운영체제와 프로세서 설계자들은 시스템에서 더 많은 성능 을 얻어내기 위해 애쓰고 있다. 프로세서, 메모리 등을 더 빠르게 만드는 것 외에, 가장 좋 은 방법은 어떤 작업들을 더 빠르게 실행할 수 있도록, 유용한 자료와 데이터의 캐시를 관 리하는 것이다. 리눅스는 메모리 관리와 관련하여 몇가지 캐시를 사용한다:

- **버퍼 캐시(Buffer Cache)** 버퍼 캐시는 블럭 디바이스 드라이버가 사용하는 데이터 버퍼들을 갖고 있다. 이들 버퍼는 고정된 크기로(예를 들어 512바이트), 블럭 장치에서 읽거나, 거 기에 쓰는 자료의 블럭을 갖고 있다. 블럭 장치는 고정된 크기의 데이터 블럭 단위로 읽 기/쓰기만을 할 수 있는 장치이다. 모든 하드 디스크는 블럭 장치이다.

  버퍼 캐시는 장치 식별자와 원하는 블럭 번호에 의해 색인되어 있고, 이 색인을 통해 데 이터 블럭을 빨리 찾을 수 있다. 블럭 장치는 버퍼 캐시를 통해서만 접근된다. 데이터가 버퍼 캐시에서 발견되면 하드 디스크같은 물리적 블럭 장치에서 읽을 필요가 없으며, 따 라서 훨씬 빠르게 접근된다.

- **페이지 캐시(Page Cache)** 페이지 캐시는 디스크상의 이미지와 데이터에 접근하는 속도를 높 이기 위해 사용된다. 이것은 파일의 논리적인 내용을 페이지 단위로 캐시하기 위해 사용 되며, 파일과 파일 내의 오프셋을 통해 접근된다. 디스크에서 메모리로 페이지들을 읽어 들이면, 페이지들은 페이지 캐시에 캐시된다.

- **스왑 캐시(Swap Cache)** 더티 페이지들만이 스왑 파일에 저장된다. 이들 페이지가 스왑 파일 에 기록된 다음 더이상 변경되지 않았다면, 그 페이지가 다음에 스왑 아웃될 때는 이미 그 페이지가 (동일한 내용으로) 스왑 파일에 있으므로, 스왑 파일에 기록할 필요가 없다. 대신 그 페이지는 그냥 폐기하면 된다. 스와핑이 심하게 일어나는 시스템에서는 이렇게 함으로써 불필요하고 값비싼 디스크 연산을 많이 줄일 수 있다.

- **하드웨어 캐시(Hardware Cache)** 흔히 구현되는 하드웨어 캐시는 프로세서 내부에 있는 페이 지 테이블 엔트리(PTE)의 캐시이다. 이 경우 프로세서는 항상 페이지 테이블을 직접 읽 는 것이 아니라, PTE를 필요로 할 때마다 페이지에 대한 변환 결과를 캐시한다. 이들은 변환 참조 버퍼(Translation Look-aside Buffers, TLB)라고 불리며 시스템의 여러 프로세스의 페이지 테이블 엔트리의 캐시된 복사본을 갖고 있다.

가상 주소를 참조할 때, 프로세서는 TLB 엔트리에서 일치하는 항목을 찾으려고 한다. 만 약 찾는다면, 가상 주소를 바로 물리적 주소로 변환하여, 데이터에 대한 올바른 연산을 수행할 수 있다. 프로세서가 일치하는 TLB 엔트리를 찾지 못하면, 운영체제의 도움을 받 아야 한다. 도움을 받기 위해 운영체제에게 TLB를 찾지 못했다는(TLB miss) 신호를 보낸 다. 문제를 해결하도록 운영체제에게 예외 신호를 전달하기 위해서는 시스템마다 특유한 메커니즘이 사용된다. 운영체제는 주소 변환을 위해 새로운 TLB 엔트리를 생성한다. 예 외가 처리된 다음, 프로세서는 같은 가상 주소 변환을 다시 시도한다. 이번에는 이 가상 주소에 해당하는 유효한 TLB 엔트리가 있기 때문에 잘 처리될 것이다.

하드웨어 캐시이든 다른 캐시이든 캐시를 사용하는 것의 단점은, 그렇게 효율을 높이기 위 해서 리눅스는 이들 캐시를 관리하는데 더 많은 시간과 공간을 사용해야 한다는 것과, 캐시 가 망가지는 경우 시스템이 죽는다는 것이다.

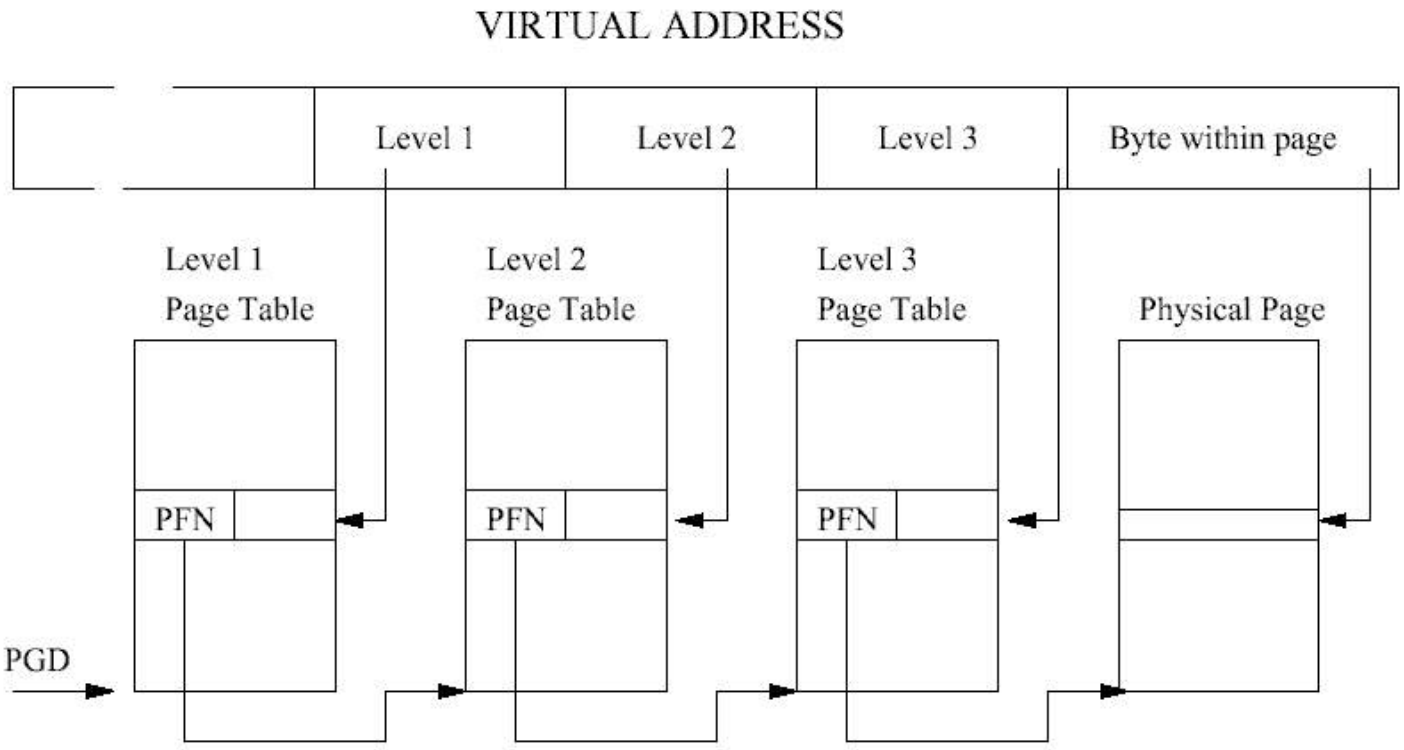

그림 3.3 : 3단계 페이지 테이블

3.3 리눅스 페이지 테이블(Linux Page Table)

리눅스는 3단계의 페이지 테이블을 가정한다[5]. 접근되는 각 페이지 테이블은 다음 단계의 페이지 테이블의 페이지 프레임 번호를 갖고 있다. 그림 3.3은 가상 주소가 어떻게 여러개의 항목으로 나누어지는지 보여준다. 각 항목은 특정 페이지 테이블에서의 오프셋을 제공한다. 가상 주소를 물리적 주소로 변환하기 위해, 프로세서는 각 단계의 항목의 내용을 가져와서 페이지 테이블을 갖고 있는 물리적 페이지에 대한 오프셋으로 변환하고, 다음 단계의 페이 지 테이블의 페이지 프레임 번호를 읽는다. 이 과정을 3회 반복하면 가상 주소를 포함하는 물리적 페이지의 페이지 프레임 번호를 얻을 수 있다. 그리고 가상 주소의 마지막 항목인 바이트 오프셋을 사용하여 페이지 내에 있는 데이터를 얻는다.

리눅스를 실행하는 플랫폼들은, 반드시 커널이 특정 프로세스의 페이지 테이블을 탐색할 수 있도록 하는 매크로들을 지원해야 한다. 이같은 방식 덕택에 커널은 페이지 테이블 엔트리 의 형식이라든가 어떻게 배열되어 있는지 알아야 될 필요가 없다. 이런 방식은 매우 성공적 이어서 세단계의 페이지 테이블을 가지는 알파 프로세서와 두 단계의 페이지 테이블을 가지 는 인텔의 x86계열의 프로세서에 대해서 동일한 페이지 테이블 처리 코드를 사용하고 있다.

3.4 페이지의 할당(allocation)과 해제(deallocation)

시스템에 있는 물리적 페이지에 대해 여러 요구들이 있다. 예를 들어, 이미지를 메모리에 로 드할 때 운영체제는 페이지를 할당해야 있다. 그리고 이미지의 실행이 끝나고 언로드될 때 페이지를 해제해야 한다. 물리적 페이지의 또 다른 용도는 페이지 테이블 자체와 같은 커널 특유의 자료구조를 저장하기 위한 것이다. 페이지 할당과 해제에 사용되는 메커니즘이나 자 료구조는, 가상 메모리 서브시스템의 효율성에 가장 중요한 영향을 미친다.

시스템의 모든 물리적 페이지는 mem_map_t[6] 구조체의 리스트인 mem_map 자료구조로 나타 내며 이들은 부팅 시에 초기화된다. 각 mem_map_t 구조체는 시스템의 물리적 페이지 하나 를 기술한다. 메모리 관리에 관해 중요한 항목들은 다음과 같다 :

- **카운트(count)** 이 페이지를 사용하고 있는 사용자(프로세스)들의 수. 페이지를 여러 프로세스 가 공유하고 있다면 카운트는 1보다 크다.
- **나이(age)** 이 항목은 페이지의 나이를 기록하고 있으며, 그 페이지가 폐기 또는 스왑할 좋은 후보인지 결정하는데 사용된다.
- **map_nr** 이 mem_map_t가 기술하는 물리적 페이지의 프레임 번호이다.

free_area 벡터는 페이지를 할당하는 코드가 프리 페이지를 찾는데 사용된다. 전체적인 버퍼 관리 계획은 이런 메커니즘으로 이루어지며 세부적인 코드에 대해서라면, 프로세서가 사용하는 페이지의 크기와 물리적인 페이징 메커니즘은 서로 다를 수 있다.

free_area의 각 원소들은 페이지 블럭들에 대한 정보를 가지고 있다. 배열의 첫번째 원소 는 한 페이지를, 그 다음은 두 페이지의 블럭들을, 그 다음은 네 페이지의 블럭들을, 이런식 으로 계속 2의 제곱으로 증가하는 페이지의 블럭들을 기술한다. list 원소는 큐의 헤드로 사용되며, mem_map 배열 내의 page 자료구조에 대한 포인터를 갖고 있다. 페이지의 프리 블럭들은 이 큐에 저장된다. map은 이 크기의 할당된 페이지 그룹을 추적하여 관리하는 비 트맵에 대한 포인터이다. 비트맵의 비트 N은 페이지의 N번째 페이지 블럭이 프리이면 1로 설정된다.

그림 3.4는 free_area 구조체를 보여준다. 0번째 원소는 하나의 프리 페이지(페이지 프레임 번호 0), 2번째 원소는 두개의 4 페이지 크기의 프리 블럭을 보여준다. 앞의 것은 페이지 프 레임 번호 4에서, 뒤의 것은 페이지 프레임 번호 56에서 시작한다.

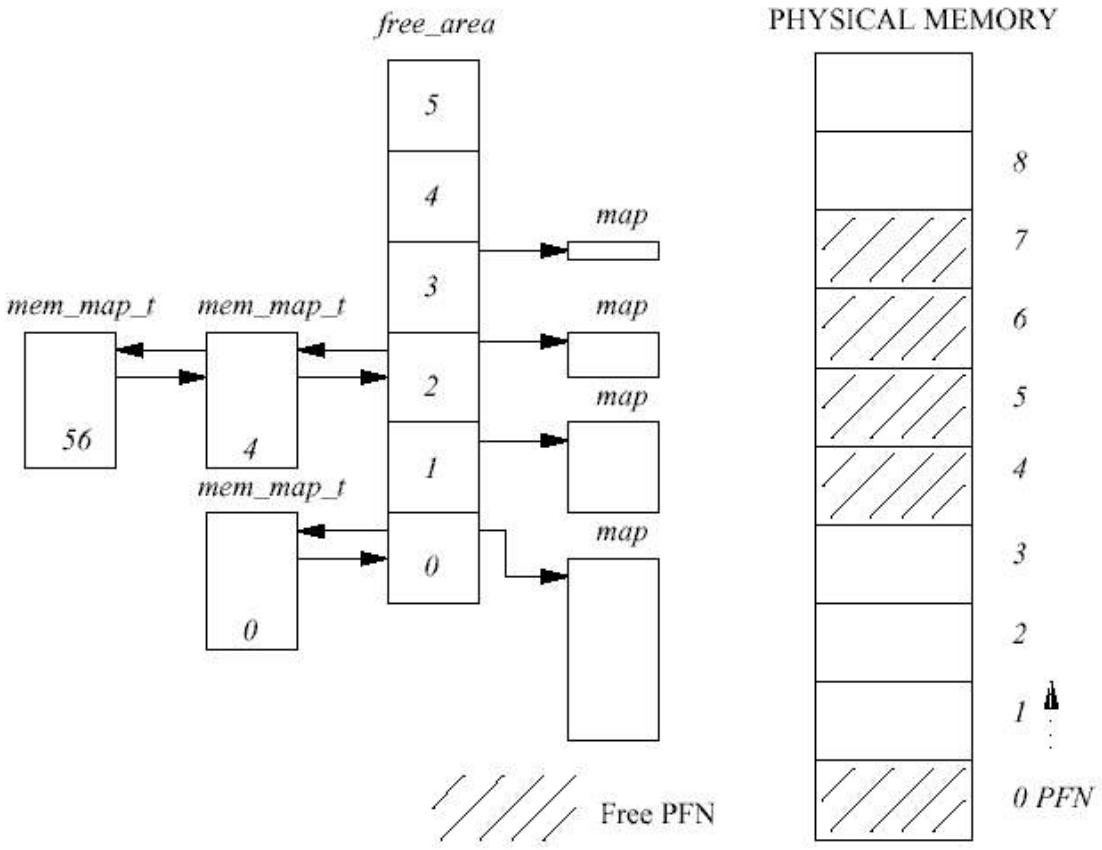

그림 3.4 : free_area 자료구조

3.4.1 페이지 할당(Page Allocation)

리눅스는 페이지 블럭을 효율적으로 할당하고 해제하기 위해 버디 알고리즘(Buddy algorithm)[7]을 사용한다. 페이지 할당 코드는 하나 이상의 물리적 페이지로 구성된 하나의 블 럭을 할당한다. 페이지들은 2의 제곱 크기인 블럭으로 할당된다. 즉 1 페이지, 2 페이지, 4 페 이지 식으로 블럭을 할당할 수 있다는 것이다. 시스템에 있는 프리 페이지가 요청을 처리하 기에 충분하다면(nr_free_pages > min_free_pages), 할당 코드는 free_area에서 요청 한 크기에 해당하는 페이지의 블럭을 탐색한다. free_area의 각 원소는 할당된 맵과, 해당 크기를 갖는 페이지의 프리 블럭의 맵을 가지고 있다. 예를 들어 배열의 두번째 원소는, 각 각 4 페이지 길이의 할당된 블럭과 프리 블럭을 기술하는 메모리 맵을 가지고 있다.

할당 알고리즘은 먼저 요청된 크기의 페이지 블럭을 검색한다. free_area 자료구조의 list 원소에 큐되어 있는 프리 페이지의 고리를 따라간다. 만일 요청된 크기의 프리 페이 지 블럭이 없다면, 그 다음 크기(요청된 크기의 두 배)의 블럭을 찾아본다. 이 과정은 모든 free_area를 다 검색하거나, 사용할 수 있는 페이지 블럭을 찾아낼 때까지 계속된다. 찾아 낸 페이지 블럭이 요청한 크기보다 크다면, 그 페이지 블럭은 요청한 크기가 될 때까지 분 할한다. 블럭에 들어있는 페이지의 수는 두 배씩 늘어나는 크기로 되어 있기 때문에, 분할과 정은 블럭을 반으로 잘라가기만 하면 된다. 프리 블럭은 해당하는 큐에 큐되며 할당된 페이 지 블럭은 호출자에게 되돌려진다.

예를 들어, 그림 3.4에서 2 페이지짜리 블럭을 요청했다면, 4 페이지짜리 첫번째 블럭(페이지 프레임 번호 4에서 시작하는)은 2 페이지짜리 블럭 두개로 나뉠 것이다. 프레임 번호 4에서 시작하는 첫번째 블럭은 할당된 페이지가 되어 호출자에게 되돌려지고, 페이지 프레임 번호 6에서 시작하는 두번째 블럭은 2 페이지 크기의 프리 블럭으로 free_area 배열의 첫번째 원소에 있는 큐에 저장된다.

3.4.2 페이지 해제(Page Deallocation)

페이지 블럭을 할당하는 것은 더 큰 프리 페이지 블럭을 작은 것으로 쪼개기 때문에 메모리 를 조각내게 된다. 페이지 해제 코드는 가능할 때마다 프리 페이지들을 더 큰 블럭의 프리 페이지로 합친다. 사실 페이지 블럭의 크기는 중요한데, 그것이 블럭들을 더 큰 블럭으로 쉽 게 합칠 수 있게 하기 때문이다.

페이지 블럭이 해제될 때마다, 같은 크기의 인접한 버디(buddy) 블럭이 프리인지 검사한다. 그렇다면 그 블럭과 새로 프리 블럭이 된 페이지들이 합쳐져서, 새로운 빈 블럭이 되어 다 음 크기의 프리 블럭을 이룬다. 두개의 페이지 블럭이 합쳐져서 더 큰 프리 페이지 블럭이 될 때마다, 페이지 해제 코드는 이 블럭을 다시 인접한 것과 합쳐서 더 큰 것으로 만들려고 한다. 이렇게 해서 프리 페이지 블럭은 메모리가 허락하는 만큼 커질 수 있게 된다.

예를 들어, 그림 3.4에서 페이지 프레임 번호 1이 해제되면, 이미 해제되어 있는 페이지 프 레임 번호 0과 합쳐져 2 페이지 크기의 프리 블럭이 되어, free_area의 첫번째 원소의 큐에 연결된다.

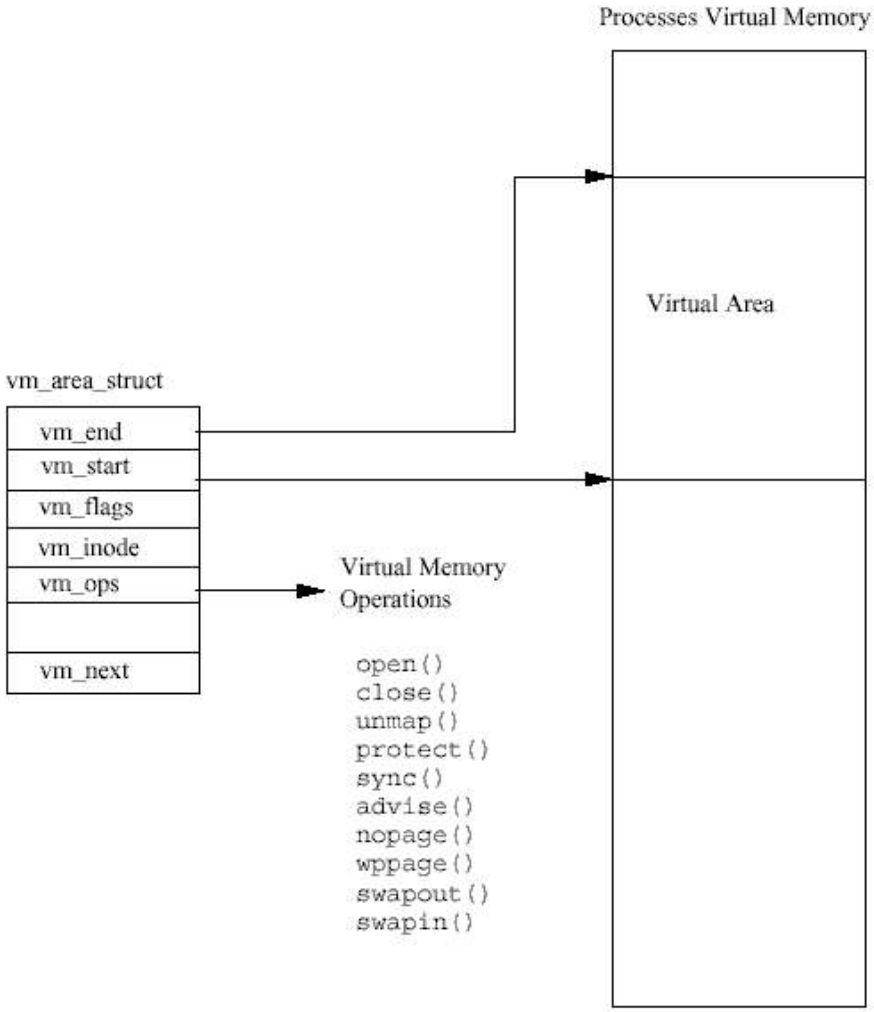

그림 3.5 : 가상 메모리의 영역들

3.5 메모리 매핑(Memory Mapping)

이미지를 실행하려면, 그 실행 이미지의 내용을 프로세스의 가상 주소공간으로 가져와야 한 다. 실행 이미지가 링크해서 사용하는 공유 라이브러리도 마찬가지다. 리눅스는 실행파일을 실제로 물리적 메모리에 가져오는 대신에, 단지 프로세스의 가상 메모리와 연결만 시킨다. 그리고 응용 프로그램이 실행되면서 프로그램의 일부가 참조됨에

따라, 실행 이미지로부터 해당하는 이미지 부분을 메모리로 가져온다. 이렇게 이미지를 프로세스의 가상 주소공간에 연결하는 것을 메모리 매핑이라고 한다.

모든 프로세스의 가상 메모리는 mm_struct 자료구조로 표현된다. 여기에는 현재 실행중인 이미지(예를 들어, bash의)에 대한 정보와, 여러개의 vm_area_struct 자료구조에 대한 포 인터가 들어 있다. 각각의 vm_area_struct 자료구조는 가상 메모리 영역의 시작과 끝, 프 로세스의 접근 권한, 메모리에 대한 연산들 등을 기술한다. 여기서 연산은 이 영역의 가상 메모리를 처리하기 위해 리눅스가 사용해야 하는 루틴들이다. 예를 들어, 가상 메모리 연산 중의 하나는, 프로세스가 가상 메모리를 접근하려다 (페이지 폴트를 통해) 그 메모리가 실제 로는 물리적 메모리에 없다는 것을 알았을 때, 이를 처리하는 올바른 작업을 수행한다. 이 연산이 nopage 연산이다. 리눅스는 실행 이미지의 페이지를 메모리로 옮길 것을 요구할 때 nopage 연산을 사용한다.

어떤 실행 이미지가 프로세스의 가상 주소에 매핑될 때, 한 세트의 vm_area_struct 자 료구조가 만들어진다. 각 vm_area_struct 자료구조는 실행 이미지의 한 부분을 나타낸다 - 실행 코드, 초기화된 데이터(변수), 초기화되지 않은 데이터(BSS) 등이다. 리눅스는 상당수 의 표준 가상 메모리 연산을 지원하며, vm_area_struct 자료구조가 만들어질 때, 그에 맞 는 일련의 가상 메모리 연산이 여기에 지정된다.

3.6 요구 페이징(Demand Paging)

실행 이미지가 프로세스의 가상 메모리에 매핑되고 나면, 실행할 수 있게 된다. 이미지의 맨 앞부분만 물리적으로 메모리에 올라와 있기 때문에, 곧 아직 물리적 메모리에 있지 않은 가 상 메모리 영역을 접근하게 된다. 프로세스가 유효한 페이지 테이블 엔트리를 갖지 않은 가 상 주소에 접근하면, 프로세서는 리눅스에 페이지 폴트를 보고한다. 페이지 폴트는 페이지 폴트가 발생한 페이지와, 페이지 폴트를 발생시킨 메모리 접근의 유형을 설명한다.

리눅스는 페이지 폴트가 발생한 곳을 포함하는 메모리 영역을 나타내는 vm_area_struct 를 찾아야 한다. vm_area_struct 자료구조를 검색하는 것은, 페이지 폴트를 효율적으로 처리하는데 있어 핵심적이기 때문에, 이들 자료구조는 AVL(Adelson-Velskii and Landis)[8] 트리 구조로 만들어져 있다. 만약 폴트가 발생한 가상 주소에 대한 vm_area_struct 자료구조 가 없다면, 이 프로세스는 금지된 가상 주소에 접근한 것이다. 리눅스는 SIGSEGV[9] 시그널 을 이 프로세스에 보내며, 이 프로세스가 그 시그널을 처리하는 핸들러를 갖고 있지 않다면, 프로세스는 종료될 것이다.

그런다음 리눅스는 발생한 페이지 폴트의 유형과, 가상 메모리의 이 영역에 대해 허용된 접 근 유형을 비교한다. 프로세스가 읽기만 허용된 영역에 쓰려고 하는 것처럼, 허용되지 않은 방법으로 접근하려고 하면 메모리 에러가 시그널로 전달된다.

페이지 폴트가 올바른 것이라도 판단했다면, 리눅스는 이를 처리해야 한다. 리눅스는 스왑 파일에 있는 페이지와, 디스크의 어딘가에 있는 실행 이미지의 일부인 페이지를 구분해야 한다. 구분을 위해 폴트가 발생한 가상 주소의 페이지 테이블 엔트리를 사용한다.

그 페이지의 페이지 테이블 엔트리가 유효하지 않지만 비어있지도 않다면, 페이지 폴트는 스왑 파일에 들어있는 페이지에 대하여 발생한 것이다. 알파 AXP의 페이지 테이블이라면, 유효 비트가 설정되지 않고, PFN 항목에 0이 아닌 값을 가진 엔트리들이 이에 해당된다. 이 경우 PFN 항목은 스왑 파일의(그리고 어떤 스왑 파일의) 어느 부분에 그 페이지가 들어있는 지에 대한 정보를 갖고 있다. 스왑 파일에 있는 페이지들을 어떻게 다루는가는 이 장의 뒤 에서 설명한다.

모든 vm_area_struct 자료구조가 가상 메모리 연산을 갖고 있는 것은 아니고, 가지고 있 다고 해도 nopage연산을 가지고 있지 않을 수도 있다. 이는 기본적으로 리눅스가 새로운 물 리적 페이지를 할당하고 이에 대한 유효한 페이지 테이블 엔트리를 생성하여, 이를 처리해 주기 때문이다. 이 가상 메모리 영역 용으로 nopage 연산이 있다면, 리눅스는 이를 사용할 것이다.

일반 nopage 연산은 메모리에 매핑된 실행 이미지를 위해 사용되며, 페이지 캐시를 사용하 여 요청한 페이지를 실제 메모리로 가져온다.

어쨌든 요청한 페이지가 물리적 메모리로 올라오면, 프로세스의 페이지 테이블이 갱신된다. 이 엔트리를 갱신하기 위하여, 특히 변환 참조 버퍼(translation look aside buffer)를 사용하는 프로세서의 경우에는, 특정한 하드웨어에 맞는 행동이 필요할 수도 있다. 이제 페이지 폴트 가 처리되었으므로 그 상황은 해제되며, 프로세스는 가상 메모리 접근에 대한 폴트를 발생 시켰던 명령에서부터 실행을 재개한다.

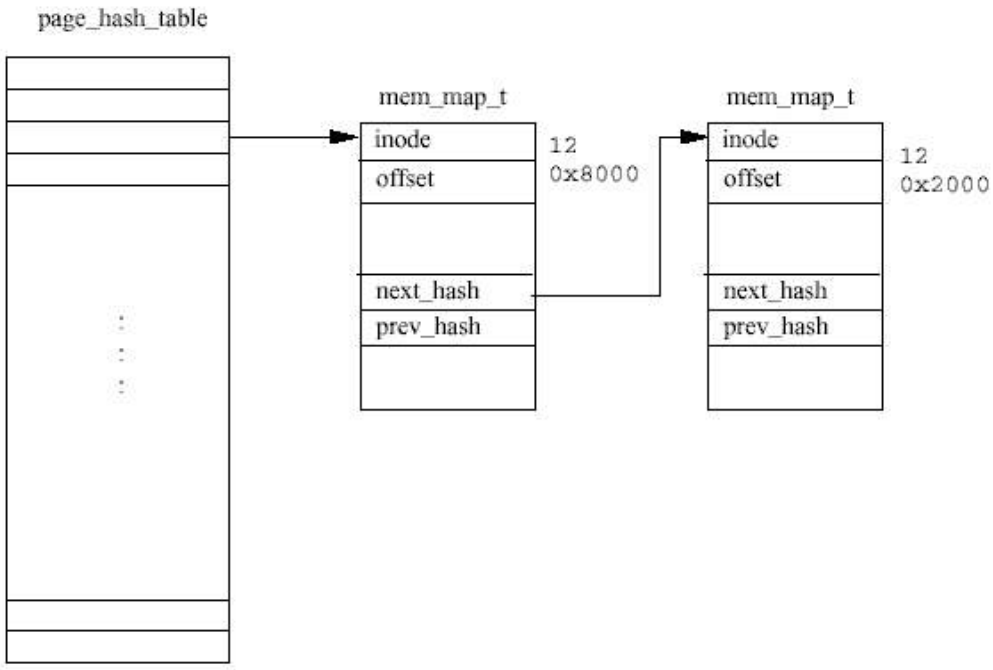

그림 3.6 : 리눅스 페이지 캐시

3.7 리눅스 페이지 캐시

리눅스 페이지 캐시의 역할은 디스크에 있는 파일로의 접근 속도를 높이는 것이다. 메모리 매핑된 파일은 한번에 한 페이지씩 읽혀지며, 이들 페이지는 페이지 캐시에 저장된다. 그림 3.6은 페이지 캐시가 mem_map_t 자료구조에 대한 포인터들의 벡터인 page_hash_table 로 구성되어 있는 것을 보여준다. 리눅스의 각 파일은 VFS inode 자료구조(9장, 파일 시스 템에서 설명)에 의해 식별되며, 각 VFS inode는 유일하며, 한 파일에 일대일로 대응되어 파 일을 완전히 기술한다. 페이지 테이블에 대한 인덱스는, 파일의 VFS inode와 파일에서의 오프셋을 가지고 만들어진다.

페이지를 메모리 매핑된 파일에서 읽을 때, 예를 들어 요구 페이징에서 페이지를 메모리로 다시 가져올 때, 페이지는 페이지 캐시를 통해 읽게 된다. 페이지가 캐시에 있으면, 그 페이 지를 나타내는 mem_map_t 자료구조에 대한 포인터가 페이지 폴트 처리 코드로 되돌려진다. 캐시에 없다면 이미지를 갖고 있는 파일 시스템으로부터 페이지를 메모리로 가져와야 한다. 리눅스는 물리적 페이지를 할당하고 디스크 상의 파일로부터 페이지를 읽어 들인다.

가능하다면 리눅스는 파일의 다음 페이지에 대한 읽기를 시작한다. 이렇게 한 페이지를 미 리 읽는 것은, 프로세스가 파일의 페이지를 순차적으로 접근하는 경우, 다음 페이지가 (프로 세스가 다음 메모리를 읽기 전에) 프로세스를 위한 메모리에 기다리고 있게 한다.

시간이 흘러 이미지를 읽고 실행함에 따라 페이지 캐시가 증가하게 된다. 페이지는 더이상 필요없게 되면, 가령 이미지가 더이상 어떤 프로세스에 의해서도 사용되지 않게 되면, 캐시 로부터 제거된다. 리눅스가 메모리를 사용해 나감에 따라 물리적 페이지가 부족해지기 시작 한다. 이 때 리눅스는 페이지 캐시의 크기를 줄일 것이다.

3.8 페이지의 스왑 아웃(swap out)과 폐기(discarding)

물리적 메모리가 부족하게 되면 리눅스 메모리 관리 서브시스템은 물리적 메모리를 해제하 려 한다. 이 일은 커널 스왑 데몬(kswapd)에게 할당된다. 커널 스왑 데몬은 커널 쓰레드라 는 특별한 종류의 프로세스이다. 커널 쓰레드는 가상 메모리 없이, 물리적 메모리 공간에서 커널모드로 실행되는 프로세스이다. 커널 스왑 데몬이라는 이름은 약간 잘못되었는데, 이는 단지 페이지를 스왑 아웃하여 시스템의 스왑 파일에 저장하는 것 이상의 여러 일을 하기 때 문이다. 커널 스왑 데몬의 역할은 메모리 관리 시스템이 효율적으로 동작할 수 있도록 시스 템에 충분한 프리 페이지가 있도록 하는 것이다.

커널 스왑 데몬(kswapd)은 커널의 init 프로세스에 의해 시작되며 커널 스왑 타이머가 주기 적으로 만료될 때를 기다리고 있다. 타이머가 만료될 때마다, 스왑 데몬은 시스템의 프리 페 이지 수가 너무 적지 않은지 확인한다. free_pages_high와 free_pages_low라는 두개의 변수를 사용하여, 페이지를 해제해야 할 필요가 있는지 결정한다. 시스템에 남아있는 프리 페이지의 수가 free_pages_high보다 큰 동안은, 커널 스왑 데몬은 아무 일도 하지 않고 다시 잠들어 다음 타이머가 만료될 때를 기다린다. 이 확인 작업을 위해, 커널 스왑 데몬은 현재 스왑 파일에 씌어지고 있는 페이지의 수도 고려한다. 이 개수는 nr_async_pages라 는 카운트 값으로 유지된다. 이 값은 어떤 페이지가 스왑 파일에 씌어지기 위해 큐에 들어 갈 때마다 증가하고, 스왑 장치에 완전히 씌어질 때마다 감소한다. free_pages_low와 free_pages_high는 시스템이 부팅할 때 설정되며, 시스템에 있는 실제 페이지 수와 관련 이 있다. 만약, 시스템에 있는 프리 페이지 수가 free_pages_high보다, 심지어는 free_pages_low보다 작아지면, 커널 스왑 데몬은 시스템이 사용하는 물리적 페이지의 수 를 줄이기 위하여 다음 세가지 방법을 시도한다.

- 버퍼 캐시와 페이지 캐시의 크기를 줄인다.
- 시스템 V 공유 메모리 페이지를 스왑 아웃한다.
- 페이지를 스왑 아웃하고 폐기한다.

시스템의 프리 페이지의 수가 free_pages_low 이하로 떨어지면, 커널 스왑 데몬은 다음 에 실행되기 전에, 6개의 페이지를 해제하려 한다. 그렇지 않으면 3개의 페이지를 해제하려 고 한다. 충분한 페이지들이 해제될 때까지 위의 각 방법이 차례로 시도된다. 커널 스왑 데 몬은 물리적 페이지를 해제하기 위해 지난번에 어떤 방법을 사용했는지 기억하고, 매번 실 행될 때마다 최종적으로 성공한 방법을 사용해서 페이지를 해제시키려고 한다[10].

충분한 페이지를 해제한 후, 스왑 데몬은 다시 잠들어 타이머가 만료되길 기다린다. 커널 스 왑 데몬이 페이지를 해제한 이유가, 프리 페이지의 수가 free_pages_low 이하로 떨어져 서 였다면, 평소에 자던 시간의 절반만 잔다. 그래서 빈 페이지의 수가 free_pages_low보 다 커지면 커널 스왑 데몬은 더 오랬동안 자게 된다.

3.8.1 페이지 캐시와 버퍼 캐시 크기를 줄이기

페이지 캐시와 버퍼 캐시에 들어있는 페이지는 free_area 벡터로 해제할 좋은 후보들이다. 메모리에 매핑된 파일의 페이지를 갖고 있는 페이지 캐시는 시스템의 메모리를 채우고 있는 불필요한 페이지를 갖고 있을 수 있다. 마찬가지로 실제 장치로 쓰거나 읽은 데이터 버퍼를 갖고 있는 버퍼 캐시 역시 불필요한 버퍼를 갖고 있을 수 있다. 시스템의 실제 페이지가 고 갈되기 시작하면, 이들 캐시로부터 페이지를 버리는 것은, 메모리에서 스왑 아웃하는 경우와 달리 실제 장치에 기록할 필요가 없으므로 상대적으로 쉽다. 이들 페이지를 버리는 것은 실 제 장치와 메모리 매핑된 파일을 액세스하는 속도가 느려진다는 것을 제외하고는 다른 심각 한 부작용은 없다. 그리고 이들 캐시로부터 페이지를 제거하는 것이 공정하게 이루어진다면, 모든 프로세스들은 공평하게 손해볼 것이다.

커널 스왑 데몬이 이들 캐시를 줄이려고 할 때 마다, mem_map 페이지 벡터에 있는 페이지 블럭을 검사하여 실제 메모리에서 버려도 될 것이 있는지 확인한다. 커널 스왑 데몬이 심하 게 스와핑을 하고 있다면 - 즉, 시스템의 프리 페이지의 수가 심각하게 낮게 떨어졌다면 - 검사할 페이지 블럭의 크기가 커진다. 페이지 블럭은 돌아가며 검사된다. 즉 메모리 맵을 줄 이려고 할 때마다 서로 다른 페이지 블럭이 검사된다. 이 방법은 시계 알고리즘(clock algorithm)이라고 불리는데, 시계 바늘의 움직임처럼 전체 mem_map 페이지 벡터에서 한번에 몇 페이지씩 차례로 조사되기 때문이다.

조사되는 각 페이지는 그것이 페이지 캐시나 버퍼 캐시에 있는 것인지 검사된다. 이 단계에 서 공유 페이지는 고려되지 않으며, 한 페이지가 동시에 두 캐시에 모두에 있을 수 없다는 것을 기억해 두기 바란다. 페이지가 두 캐시 어디에도 속하지 않으면 mem_map 페이지 벡터 의 다음 페이지가 조사된다.

버퍼의 할당과 해제가 더욱 효율적으로 이루어지게 하기 위하여 (페이지 내의 버퍼가 캐시 되는 것이 아니라) 페이지 자체가 버퍼 캐시에 캐시된다. 메모리 맵 축소 코드는 검사되는 페이지에 포함된 버퍼를 해제하려고 한다. 페이지에 포함된 모든 버퍼가 해제되면, 그들을 갖고 있던 페이지도 해제된다. 조사된 페이지가 리눅스 페이지 캐시에 있다면, 페이지 캐시 에서 제거된 다음 해제된다.

이렇게 해서 충분한 페이지가 해제되었다면 커널 스왑 데몬은 다음에 주기적으로 깨어나는 시점까지 기다린다. 해제되는 페이지 중에는 어떤 프로세스의 가상 메모리에도 속하지 않으 므로 (모두 캐시된 페이지이므로), 아무런 페이지 테이블도 수정할 필요가 없다. 캐시된 페 이지를 제거하는 걸로 충분하지 않은 경우, 스왑 데몬은 공유 페이지를 스왑 아웃하려고 하 게 된다.

3.8.2 시스템 V 공유 메모리 페이지의 스왑 아웃

시스템 V 공유 메모리는 둘 이상의 프로세스가 가상 메모리를 공유하여 그들 사이에 정보 를 전송할 수 있는 프로세스간 통신(IPC) 메커니즘의 일종이다. 프로세스들이 이 방법으로 어떻게 메모리를 공유하는가는 5장에서 자세히 설명한다. 아직은 시스템 V 공유 메모리의 각 영역을 shmid_ds 자료구조로 기술한다고 알아두는 것으로 충분하다. 이 자료구조는 이 가상 메모리 영역을 공유하는 프로세스마다 하나씩 대응되는 vm_area_struct 자료구조 리스트에 대한 포인터를 갖고 있다. vm_area_struct 자료구조는 각 프로세스의 가상 메모 리의 어디에 이 시스템 V 공유 메모리가 대응하는지 나타낸다. 이 시스템 V 공유 메모리용 vm_area_struct 자료구조들은 vm_next_shared, vm_prev_shared 포인터로 서로 연결 되어 있다. 각각의 shmid_ds 자료구조는 이밖에 공유 가상 페이지가 매핑되어 있는 실제 페이지를 설명하고 있는 페이지 테이블 엔트리의 리스트도 갖고 있다.

커널 스왑 데몬은 시스템 V 공유 메모리 페이지를 스왑 아웃할 때에도 시계 알고리즘(clock algorithm)을 사용한다. 커널 스왑 데몬은 실행할 때마다 맨 마지막으로 스왑 아웃한 공유 가 상 메모리 페이지가 무엇이었는지를 기억한다. 이를 위해 두개의 인덱스 값을 유지 하는데, 하나는 shmid_ds 자료구조 집합에 대한 인덱스이고, 다른 하나는 시스템 V 공유 메모리 영역을 나타내는 페이지 테이블 엔트리의 리스트에 대한 인덱스이다. 이 방법은 시스템 V 공유 메모리 영역이 공정하게 희생되게 한다.

어떤 시스템 V 공유 메모리의 가상 페이지에 대한 물리적 페이지 프레임 번호는, 이 가상 메모리 영역을 공유하는 모든 프로세스의 페이지 테이블에 들어있기 때문에, 커널 스왑 데 몬은 이들 페이지 테이블 모두를 변경하여, 이 페이지가 더이상 메모리에 없고 스왑 파일에 들어 있다는 것을 알려주어야 한다. 스왑 아웃되는 각 공유 페이지마다, 커널 스왑 데몬은 이 페이지를 공유하고 있는 프로세스들의 페이지 테이블로부터 페이지 테이블 엔트리를 찾는다 (각 vm_area_struct 자료구조에서 포인터를 따라감으로써). 이 시스템 V 공유 메모 리 페이지에 대한 프로세스의 페이지 테이블 엔트리가 유효하면, 데몬은 그것을 유효하지 않고 스왑 아웃된 페이지 테이블 엔트리로 변환하고, 이 (공유된) 페이지의 사용자 수를 1 감소시킨다. 스왑 아웃된 시스템 V 공유 페이지 테이블 엔트리에는, shmid_ds 자료구조 집 합에 대한 인덱스와, 이 시스템 V 공유 메모리 영역에 대한 페이지 테이블 엔트리의 인덱스가 들어 있다.

공유하는 프로세스들의 페이지 테이블이 모두 변경되어 그 페이지의 카운트가 0이 되면, 이 공유 페이지를 스왑 파일로 스왑 아웃할 수 있게 된다. 이 시스템 V 공유 메모리 영역에 대 한 shmid_ds 자료구조가 가리키고 있는 리스트에 들어 있는 페이지 테이블 엔트리들은 스 왑 아웃된 페이지 테이블 엔트리로 교체된다. 스왑 아웃된 페이지 테이블 엔트리는 유효하 지 않지만, 열린 스왑 파일들 중 하나를 가리키는 인덱스와, 그 파일 안의 어디에 스왑 아웃 된 페이지가 있는지를 나타내는 오프셋을 갖고 있다. 이 정보는 그 페이지를 다시 물리적 메모리로 가져올 때 사용된다.

3.8.3 페이지의 스왑 아웃과 폐기

스왑 데몬은 시스템에 있는 각 프로세스를 차례로 관찰하면서, 그것이 스왑하기 좋은 후보 인지 판단한다. 좋은 후보는 스왑될 수 있으면서(스왑될 수 없는 프로세스도 있다), 메모리 에서 스왑되거나 폐기될 수 있는 페이지를 하

나 이상 가진 프로세스들이다. 페이지들은 그 안에 저장된 데이터를 다른 방법으로 얻어올 수 있는 방법이 없을 때만, 물리적 메모리로부 터 시스템의 스왑 파일에 스왑 아웃된다.

실행 이미지의 상당수는 실행 파일에서 가져온 것이며, 그 파일에서 쉽게 다시 읽을 수 있 다. 예를 들어 이미지에 들어있는 실행 명령은 변경되지 않기 때문에 스왑 파일에 쓸 필요 가 없다. 이들 페이지는 그냥 폐기하고, 프로세스가 이들을 다시 참조할 때, 실행 이미지에 서 메모리에 다시 가져오게 된다.

스왑할 프로세스를 결정하면, 스왑 데몬은 그 프로세스의 가상 메모리 영역을 전부 보면서 공유되거나 락이 걸리지 않은 영역을 찾는다. 리눅스는 선택된 프로세스에 있는 스왑 가능 한 페이지를 모두 스왑 아웃하지는 않는다. 대신 페이지 몇 개만 제거할 뿐이다. 메모리에 락되어 있는 페이지는 스왑하거나 폐기할 수 없다.

리눅스 스왑 알고리즘은 페이지 에이징(page aging)을 사용한다. 각 페이지는 카운터를 가지 고 있어서 (mem_map_t 자료구조에 저장되어 있다), 커널 스왑 데몬이 어떤 페이지를 스왑 하는 것이 좋은지 결정하는데 도움을 준다. 페이지는 사용하지 않으면 나이를 먹고, 사용할 수록 젊어진다; 스왑 데몬은 나이가 많은 페이지만을 스왑 아웃한다. 페이지를 처음 할당할 때 페이지의 초기 나이는 3이다. 페이지가 사용될 때마다, 나이값은 3씩 증가되어 최대 20까 지 증가된다(이 값이 작을수록 오래된 페이지이다). 커널 스왑 데몬이 실행될 때마다 페이지 의 나이값을 1씩 감소시켜 페이지를 오래된 것으로 만든다. 이 기본 동작은 변경될 수 있으 며, 이런 이유로 (다른 스왑 관련 정보와 함께) swap_control 자료구조에 저장되어 있다.

페이지가 아주 오래되면 (나이가 0이 되면) 스왑 데몬은 그 페이지를 좀 더 처리하게 된다. 더티 페이지는 스왑 아웃될 수 있는 페이지이다. 리눅스는 PTE에서 아키텍쳐 특유의 비트 를 사용해서 페이지를 이와 같은 방식으로 기술한다 (그림 3.2 참조) 그러나, 모든 더티 페 이지가 반드시 스왑 파일에 기록되어야 하는 것은 아니다. 어떤 프로세스는 모든 가상 메모 리 영역에서 자신의 스왑 연산(vm_area_struct의 vm_ops 포인터가 가리킴)을 가질 수 있 으며, 이 경우 그 연산이 사용된다[11]. 연산이 정의되지 않았다면 스왑 데몬은 스왑 파일에 페이지를 할당하고 스왑 페이지를 스왑 파일에 기록한다.

이제 그 페이지의 페이지 테이블 엔트리는 유효하지 않다고 표시되었지만, 여기에는 이 페 이지가 스왑 파일의 어디에 저장되었는지에 대한 정보가 들어 있다. 이 정보는 어느 스왑 파일이 사용되었는지, 그리고 스왑 파일 내에서 페이지가 저장된 위치의 오프셋으로 구성된 다. 어떤 스왑 방법을 사용하였든, 원래의 물리적 페이지는 다시 free_area에 넣어져서 프 리 상태가 된다. 클린 페이지(더티하지 않은 페이지)는 폐기되어 재사용할 수 있도록 free_area에 들어간다.

스왑 가능한 프로세스 페이지를 충분히 스왑 아웃하거나 폐기하면, 스왑 데몬은 다시 잠든 다. 스왑 데몬이 다음에 깨어났을 때는, 시스템의 다음 프로세스를 검토하게 된다. 이런 방 식으로 스왑 데몬은 시스템이 다시 균형에 이를 때까지 각 프로세스의 물리적 페이지를 조 금씩 없앤다. 이것은 전체 프로세스를 스왑 아웃하는 것보다 훨씬 공정하다.

3.9 스왑 캐시(Swap Cache)

리눅스는 페이지를 스왑 파일에 스왑 아웃할 때, 페이지를 쓸 필요가 없을 땐 쓰지 않으려 고 한다. 어떤 페이지가 스왑 파일과 물리적 메모리에 (같은 내용으로) 동시에 존재하는 경 우가 있다. 이런 경우는 어떤 페이지가 메모리에서 스왑 아웃되었다가, 한 프로세스가 그 페 이지에 다시 접근하여 메모리로 다시 들어온 경우에 발생한다. 이 때 메모리상의 페이지가 덮어 씌어지지 않는 한 스왑 파일에 있는 페이지의 복사본은 유효하다.

리눅스는 이러한 페이지들을 추적하기 위해 스왑 캐시를 사용한다. 스왑 캐시는 페이지 테 이블 엔트리의 리스트로, 각 엔트리는 시스템에 있는 물리적 페이지 하나에 해당한다. 이 페 이지 테이블 엔트리는 하나의 스왑 아웃 페이지에 대한 것으로, 그 페이지가 어느 스왑 파 일에, 어느 위치에 있는지를 기술한다. 만약 스왑 캐시 엔트리 값이 0이 아닌 경우, 변경되 지 않은 페이지가 스왑 파일 내에 들어 있다는 것을 나타낸다. 페이지가 (덮어 씌어져서) 변경된 경우, 그 페이지의 엔트리는 스왑 캐시에서 삭제된다.

리눅스가 어떤 물리적 페이지를 스왑 파일에 스왑 아웃할 필요가 있을 때, 먼저 스왑 캐시 에 문의하며, 만약 이 페이지에 대한 유효한 엔트리가 있는 경우, 이 페이지는 스왑 파일에 기록할 필요가 없다. 왜냐하면 메모리에 있는 페이지의 내용이 스왑 파일로부터 마지막으로 읽은 다음 한번도 변경되지 않았기 때문이다.

스왑 캐시의 엔트리는 스왑 아웃된 페이지에 대한 페이지 테이블 엔트리이다. 이들은 유효 하지 않다고 표시되어 있지만, 리눅스가 올바른 스왑 파일과 그 스왑 파일 내에서의 올바른 페이지를 찾을 수 있도록 하는 정보를 갖고 있다.

3.10 페이지 스왑 인(Swapping Pages In)

응용 프로그램이 이미 스왑 아웃된 물리적 페이지에 있는 가상 메모리에 쓰려고 하는 경우 처럼 스왑 파일에 저장된 더티 페이지들이 다시 필요로 한 경우가 있다. 물리적 메모리에 있지 않은 페이지에 접근하면 페이지 폴트가 발생한다. 페이지 폴트는 프로세서가 가상 주 소를 물리적 주소로 변환할 수 없을 때 운영체제에 보내는 신호이다. 이 경우는 가상 메모 리 페이지가 스왑 아웃되었을 때에는 이 페이지를 기술하는 페이지 테이블 엔트리가 유효하지 않다고 표시되기 때문에 페이지 폴트가 발생하는 것이다. 프로세서는 가상 주소를 물리 적 주소로 변환할 수 없기에, 제어를 운영체제에 넘겨주면서 폴트가 발생한 가상 주소와 폴 트의 이유를 알린다. 이 정보의 형식과 프로세서가 운영체제에 제어를 넘기는 방법은 프로 세서에 따라 다르다. 프로세서마다 다르게 구현되어 있는 페이지 폴트를 처리하는 코드는 폴트가 발생한 가상 주소를 포함하고 있는 가상 주소 영역을 나타내는 vm_area_struct 자료구조를 찾아야 한다. 이 코드는 폴트가 발생한 가상 주소가 들어있는 자료구조를 찾을 때까지, 해당 프로세스가 사용하는 vm_area_struct 자료구조를 검색한다. 이 작업은 매우 짧은 시간 안에 이루어져야 하므로, 프로세스들이 가지고 있는 vm_area_struct 자료구조 는 이 검색을 가능한 빨리 할 수 있도록 배치되어 있다[12].

프로세서에 따라 적절한 작업을 수행하여 폴트가 발생한 가상 주소가 가상 메모리의 유효 영역이라고 판단하면, 페이지 폴트 처리는 이제 일반화되어 리눅스가 동작하는 모든 프로세 서에 적용되는 코드로 넘어가게 된다. 일반화된 페이지 폴트 처리 코드는 폴트가 발생한 가 상 주소에 대한 페이지 테이블 엔트리를 찾는다. 찾은 페이지 테이블 엔트리가 스왑 아웃된 페이지를 가리키고 있으면, 리눅스는 그 페이지를 다시 물리적 메모리로 가져와야 한다. 스 왑 아웃된 페이지에 대한 페이지 테이블 엔트리의 형식은 프로세서마다 다르지만, 어쨌든 모든 프로세서들은 이 페이지가 유효하지 않다고 표시하고, 스왑 파일에서 페이지의 위치를 찾는데 필요한 정보를 페이지 테이블 엔트리에 넣어두고 있다. 리눅스는 페이지를 다시 물 리적 메모리로 가져오기 위해 이 정보를 필요로 한다.

이 시점에서, 리눅스는 폴트가 발생한 가상 주소와, 이 페이지가 어디에 스왑되어 있는지에 대한 정보를 갖고 있는 페이지 테이블 엔트리를 알고 있다. vm_area_struct 자료구조는 자신이 기술하는 가상 메모리 영역의 어떤 페이지를 물리적 메모리로 스왑할 수 있는 루틴 에 대한 포인터를 가지고 있을 수 있다. 이것이 swapin 연산이다[13]. 이 가상 메모리 영역에 대해 swapin 연산이 정의되어 있으면 리눅스는 그것을 사용한다. 사실 시스템 V 공유 메 모리의 스왑 아웃이 이렇게 처리되는데, 스왑 아웃된 시스템 V 공유 메모리의 형식이, 일반 스왑 아웃된 페이지의 포맷과 약간 다르기 때문에, 특별한 처리가 더 필요하기 때문이다. swapin 연산이 없는 경우엔, 리눅스는 이를 일반 페이지여서 특별히 처리가 필요 없다고 생각한다. 이제 비어있는 물리적 페이지를 할당하고, 스왑 아웃 되었던 페이지를 스왑 파일 에서 읽어들인다. 어느 스왑 파일의 어디에 페이지가 있는지 알려주는 정보는, 해당하는 유 효하지 않은 페이지 테이블 엔트리에서 얻는다.

만약 페이지 폴트를 발생한 접근이 쓰기가 아니라면, 페이지는 여전히 스왑 캐시에 남아 있 으며, 메모리로 가져온 페이지 테이블은 쓰기가 안된다고 표시가 된다. 뒤에 이 페이지에 쓰 기를 시도하면, 또 다른 페이지 폴트가 발생하고, 이 시점에서 그 페이지는 더티로 표시되고, 스왑 캐시에서 엔트리를 제거하게 된다. 페이지에 기록한 것이 없고 다시 스왑 아웃될 필요 가 있다면, 그 페이지는 이미 스왑 파일에 있기 때문에 리눅스는 페이지를 스왑 파일에 쓸 필요가 없게 된다[14].

스왑 파일로부터 페이지를 가져오도록 한 접근이 쓰기 연산이었다면, 이 페이지는 스왑 캐 시에서 제거되고, 페이지 테이블 엔트리는 더티, 쓰기 가능으로 표시된다.

번역 : 심마로, 고양우, 정직한, 김기용, 신문석, 이대현
정리 : 이호

참고 자료. 인텔 386 보호모드 메모리 아키텍쳐

이호 (flyduck)

커널에서 메모리 관리 시스템의 구현은 해당 CPU의 도움을 받아야 한다. 리눅스의 메모리 관리 시스템을 이용하려면 CPU에서 페이징과 메모리 보호, 페이지 폴트 처리를 할 수 있는 메커니즘을 제공해야 하며 인텔 x86 계열의 CPU에서는 80386에서부터 이러한 메모리 아키 텍쳐를 제공하고 있다. 여기서는 x86 계열의 메모리 아키텍쳐를 간단히 살펴보도록 한다.

8086 CPU는 16비트 세그먼트(segment) 레지스터와 16비트 오프셋을 중첩하여 20비트, 즉 1MB 크기의 주소공간을 제공한다. 80286에서는 8086과 똑같은 주소공간을 제공하는 실제모 드(real mode)와 함께, 새로운 방식의 주소공간을 제공하는 보호모드(protected mode)가 도입되 었다. 80286 보호모드에서 세그먼트 레지스터는 셀렉터(selector)라는 이름으로 바뀌었고, 셀 렉터를 24비트의 베이스 주소(base address)로 바꾸어주는 테이블인 디스크립터 테이블 (descriptor table)이 등장했다. 이 모드에서는 24비트의 베이스 주소와 16비트의 오프셋을 더 하여 모두 24비트의 주소공간, 즉 16MB의 주소공간을 제공하였다. 여기서 셀렉터와 디스크 립터 테이블을 이용하여 선형 주소공간(linear address space)의 일부를 가리킬 수 있도록 하는 것을 세그멘테이션(segmentation)이라고 한다. 80386에서는 이러한 세그멘테이션 외에 메모리 관리에 필수적인 페이징 메커니즘이 추가되고 메모리 공간도 32비트, 즉 4GB로 확장되었다.

80386에서 메모리 상의 주소를 가리키는 데에는 16비트의 셀렉터(selector) 레지스터와 32비 트의 오프셋(offset)이 사용된다. 이들은 세그멘테이션 메커니즘을 거쳐 선형 주소(linear address)[15]로 변환되고, 다시 이 주소는 페이지 테이블을 이용한 페이징 메커니즘을 거쳐 물 리적인 실제 주소(physical address)로 바뀌게 된다. 알파 AXP와 같은 다른 CPU에서는 세그멘 테이션이라는 것을 제공하지 않으며, 이는 인텔 CPU의 특성이라고 할 수 있다. 이는 세그먼 트 레지스터에서부터 시작한 잔상이라고 할 수 있으며, 리눅스 역시 이 기능을 사용하지 않 고 있다. 다만 인텔 CPU에서 동작하는 다른 운영체계와 마찬가지로 세그멘테이션을 거쳐 나오는 선형 주소공간을 사용자 주소공간과 커널 주소공간으로 분리하여, 사용자 주소공간 에 3GB를 커널 주소공간으로 1GB를 할당해 놓고 있다.

```
Selector              Linear Address              Physical Address
       segmentation                  paging                          memory
       mechanism                     mechanism
Offset
```

셀렉터는 디스크립터 테이블에 대한 인덱스와, 어떤 디스크립터 테이블을 가리키는지를 나 타내는 TI (Table Indicator) 항목, 그리고 이를 사용할 수 있는 레벨을 나타내는 RPL(Requestor Privilege Level) 세가지로 이루어져 있다. TI 항목이 0이면 인덱스는 전역 디스크립터 테이블 (Global Descriptor Table, GDT)에 있는 디스크립터를 가리키고, TI 항목이 1이면 지역 디스크립 터 테이블(Local Descriptor Table LDT)를 나타낸다. 여기서 GDT는 커널 모드에서 사용되는 테 이블이고, LDT는 사용자 모드에서 사용되는 테이블이다. 보통 GDT'는 커널 모드용으로 하나가 있으며, LDT는 각 프로세스별로 하나씩 만들어진다. 이들 테이블의 시작 위치는 각각 GDTR, LDTR이라는 레지스터가 가리키고 있다.

디스크립터 테이블은 64비트 크기로, 32비트 크기의 베이스 주소와 20비트 크기의 범위 (limit), 그리고 기타 여러 항목으로 이루어져 있다. 여기서 베이스 주소는 4GB의 선형 주소 공간에서의 시작 위치를 가리키고, 범위는 베이스 주소에서 시작하여 접근이 가능한 메모리 범위를 나타낸다. 이것은 20비트 크기이긴 하지만 입도 비트(granularity bit)가 설정되어 있으 면 4KB 단위의 범위를 나타내므로 모두 4GB 크기의 범위를 가질 수 있다. 이렇게 나온 베 이스 주소에 오프셋을 더하면 실제 선형 주소공간에서의 주소가 나오게 된다. 즉, 세그멘테 이션 메커니즘에서는 셀렉터를 이용하여 디스크립터를 찾고, 여기 있는 베이스 주소에 오프 셋을 더하여 선형 주소공간에서의 주소를 얻는 역할을 한다.

이렇게 얻어진 선형 주소는 실제 주소가 아니며, 페이징 메커니즘을 거쳐야 실제 주소를 얻 을 수 있다. 페이징 메커니즘에서는 이 선형 주소를 다시 10비트 크기의 페이지 디렉토리 인덱스(page directory index), 10비트 크기의 페이지 테이블 인덱스(page table index), 12비트 크 기의 오프셋으로 쪼갠다. 페이지 디렉토리 인덱스를 가지고 페이지 디렉토리에서 페이지 테 이블의 주소를 얻을 수 있다. 다시 페이지 테이블 인덱스를 가지고 앞의 페이지 디렉토리가 가리키는 페이지 테이블에서 페이지 프레임(page frame)의 위치를 얻을 수 있다. 이렇게 얻어 진 페이지 프레임 주소에 오프셋을 더하면 실제 물리적인 주소가 나오게 된다. 이는 앞의 그림 3.3에서 나오는 3단계 페이지 테이블에서 하나를 빼서 2단계 페이지 테이블을 생각하 면 된다. 여기서 오프셋은 12비트이므로 하나의 페이지 프레임은 2^{12}, 즉 4KB의 크기를 가지 며, 리눅스에서 정의된 페이지 크기는 이 값이다. 이렇게 페이징 메커니즘을 통하여 선형 주 소는 실제 물리적인 주소로 변환되며, 리눅스는 CPU의 이런 지원을 통하여 페이징을 구현 할 수 있다.

역주 1) 실제 인텔 80386에서 메모리를 4KB 페이지 단위로 다루고 있으며, 이 페이지 크기 는 하드웨어에서 지원하는 크기를 따른 것이다. (flyduck)

역주 2) 4KB는 212이므로 이 한페이지의 주소를 나타내는데 12비트가 필요하다. 인텔 80385 CPU에서는 페이지 프레임 번호에 20비트, 오프셋에 12비트를 사용하여 모두 32비트의 주소공간 즉 4GB의 주소공간을 갖는다. (flyduck)

역주 3) 즉 물리적 주소는 Physical PFN * PAGE_SIZE + offset이다. (flyduck)

역주 4) 계속 스왑 파일을 접근하느라 디스크만 고생하고 실제 작업은 실행되지 않는 현상 (심마로)

역주 5) 이들은 각기 페이지 디렉토리(page directory), 페이지 중간 디렉토리(page middle directory), 페이지 테이블(page table)이라고 하며, 각기 pgd_t, pmd_t, pte_t 타입으로 정의되 어 있다. (flyduck)

6) 헷갈리게도 이 구조체를 페이지(page) 구조체라고도 부른다.

7) 여기에 참고 목록을 적을 것. 역주 8) 높이 균형을 이루는 이진 트리, 사실 이 사람 이름들은 몰라도 된다. (심마로)

역주 9) Segmentation Fault. (flyduck)

역주 10) 이 밖에 min_free_pages 라는 값이 있는데, 이는 커널이 필요로 하는 경우 바로 프리 페이지를 얻을 수 있도록, 프리 페이지의 갯수가 이 값 이하로 떨어지지 않도록 한 다. 이 값 역시 부팅시에 설정이 된다. (flyduck)

역주 11) vm_area_struct 자료구조에는 해당 가상 메모리 영역에 대한 연산을 할 때 사용 할 함수들에 대한 포인터가 들어 있다. 이것이 NULL 값이라면 기본 동작을 수행하지만, 따로 지정된 것이 있다면 해당하는 함수를 부르게 된다. swapout이나 swapin 함수가 여기에 들어있으며, 이전에 설명한 nopage 연산도 여기에 함수 포인터로 들어 있다. 여기 서는 swapout 연산에 대한 포인터가 사용된다. include/linux/mm.h의 struct vm_area_struct, struct vm_operations_struct 참조. (flyduck)

역주 12) 앞에서 설명한 바와 같이 프로세스에 관련된 메모리를 나타내는 mm_struct에는 vm_area_struct의 연결 리스트와 함께 AVL 트리를 같이 가지고 있다. AVL 트리를 관 리하는 것은 약간의 오버헤드가 있지만 페이지 폴트를 빨리 처리하기 위해서는 이를 감 수해야 한다. (flyduck)

역주 13) 앞의 swapout 연산과 마찬가지로 vm_area_struct 자료구조에 있는 vm_ops 포 인터에 (vm_operations_struct 구조체) 이 포인터가 들어 있다. (flyduck)

역주 14) 스왑 캐시에서 나온바와 같이 스왑 파일에 있는 내용과 메모리에 있는 내용이 달 라진 경우에만 스왑 캐시에서 제거할 수 있도록, 첫번째 페이지 폴트에서는 메모리로 가 져오기만 하고, 두번째 페이지 폴트가 발생할 때 스왑 캐시에서 제거하게 된다. (flyduck)

역주 15) 이 선형 주소는 커널에서 생각하는 가상 주소와 같은 것이라고 생각하면 된다. (flyduck)