26

VAX/VMS 가상 메모리 시스템

가상 메모리에 대한 학습을 마치기 전에, VAX/VMS 운영체제 [LL82]의 잘 만들어진 가상 메모리 관리자에 대해 상세하게 살펴보자. 여기서는 이전의 장에서 살펴보았던 몇 가지 개념들이 완전한 메모리 관리자 내에 어떤 식으로 적용되었는지 설명할 것이다.

26.1 배경

VAX-11 미니컴퓨터의 구조는 1970년대 말에 **Digital Equipment Corporation**(**DEC**)에 의해서 소개되었다. DEC는 미니컴퓨터 시대에 컴퓨터 산업계의 일류 회사였다. 불행하게도 PC의 등장과 수차례의 잘못된 경영 판단으로 인해 서서히(하지만 분명하게) 사라졌다 [Chr03]. 이 구조는 VAX-11/780과 그 보다는 낮은 성능의 VAX-11/750을 포함한 컴퓨터에 구현되었다.

VAX/VMS(또는 단순하게 VMS)라고 알려진 운영체제의 주요한 아키텍트는 Dave Cutler로서 이후에는 Microsoft Windows NT 개발을 이끌었다 [CS93]. VMS는 다양한 종류의 기계들에서 실행되어야 한다는 문제를 안고 있었다. VMS가 설치될 컴퓨터는 저가의 VAX들부터 엄청난 고가의 고사양 기계들도 있었다. VMS 운영체제는 다양한 종류의 시스템들에서 동작하는(그것도 잘 동작하는) 기법들과 정책들을 필요로 했다.

핵심 질문: 일반론의 저주(The Curse of Generality)에서 어떻게 방지할까
운영체제는 종종 "일반론의 저주"라고 알려진 문제를 가진다. 운영체제는 다양한 응용 프로그램들과 시스템 부류를 범용으로 지원해야 할 책임이 있다. 이 저주의 결과는 운영체제가 어느 하나의 기계도 제대로 지원할 수 없게 된다는 것이다. VMS의 경우, 그 저주는 현실이 되었다. VAX-11 구조는 여러 다른 형태로 구현되었기 때문이다. 운영체제가 다양한 시스템에서 효과적으로 실행되려면 어떻게 구현되어야 하는가?

VMS는 컴퓨터의 구조적 결함을 소프트웨어로 보완한 훌륭한 사례다. 운영체제가 이상적인 개념과 환상을 제공하기 위해 하드웨어에 의존하지만, 하드웨어가 모든 것을 제대로 해내지 못할 경우도 있다. VAX 하드웨어에서 해당 사례들을 몇 가지 볼 것이다.

하드웨어 결함에도 불구하고, 시스템이 효과적으로 작동하기 위해 VMS 운영체제가 무엇을 하였는지 볼 것이다.

26.2 메모리 관리 하드웨어

VAX-11은 프로세스마다 512바이트 페이지 단위로 나누어진 32비트 가상 주소 공간을 제공한다. 가상 주소는 23비트 VPN과 9비트 오프셋으로 구성되어 있다. VPN의 상위 두 비트는 페이지가 속한 세그멘트를 나타내기 위해서 사용되었다. 이 시스템은 앞서 살펴보았던 페이징과 세그멘테이션의 하이브리드 구조를 갖고 있다.

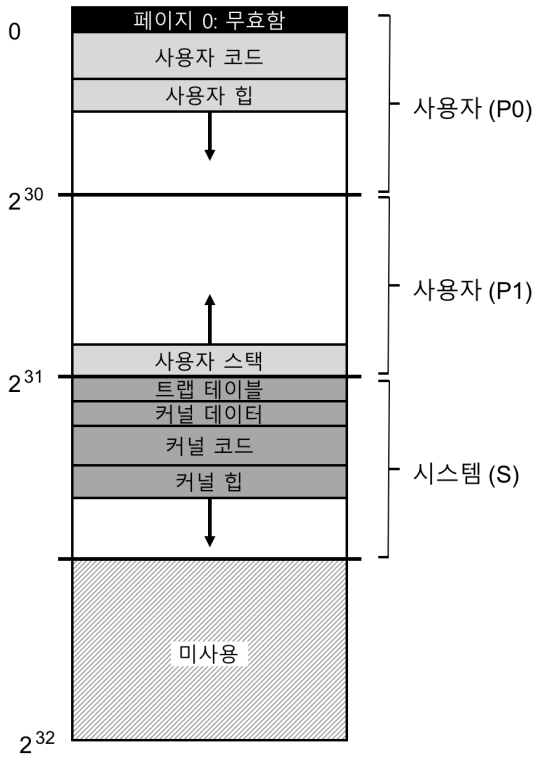
주소 공간의 하위 절반은 "프로세스 공간"으로 알려져 있으며 각 프로세스마다 다르게 할당된다. 프로세스 공간의 첫 번째 절반(`P0`으로 알려져 있음)에 사용자 프로그램과 힙(heap)이 존재한다. 힙은 주소가 큰 쪽으로 증가한다. 프로세스 공간의 두 번째, 즉 큰 쪽 절반은(`P1`) 주소가 작은 방향으로 증가하는 스택(stack)이 존재한다. 주소 공간의 상위 절반은 그 중 반만 사용되며 시스템 공간(`S`)으로 불린다. 운영체제의 보호된 코드와 데이터가 이곳에 존재하며, 이 방식으로 여러 프로세스가 운영체제를 공유한다.

VMS 설계자들의 주요 고민 중 하나는 믿을 수 없을 정도로 작은 VAX 하드웨어의 페이지 크기였다(512바이트). 역사적인 이유로 페이지 크기가 결정되었지만 선형 페이지 테이블의 크기가 지나치게 커진다는 것이 근본적인 문제였다. VMS 설계자들의 첫 목표 중에 하나는 VMS가 페이지 테이블 저장을 위해 메모리를 소진하는 것을 막는 것이었다.

이 시스템은 페이지 테이블로 인한 메모리 압박의 정도를 경감시키기 위한 두 가지 방법을 사용하였다. 첫째, VAX-11은 사용자 주소 공간을 두 개의 세그멘트로 나누어 프로세스마다 (P0과 P1) 각 영역을 위한 페이지 테이블을 가지도록 하였다. 스택과 힙 사이의 사용되지 않은 주소 영역을 위한 페이지 테이블 공간이 필요 없게 되었다. 베이스와 바운드 레지스터는 예상하는 그대로의 방식으로 사용된다. 베이스 레지스터는 해당 세그멘트의 페이지 테이블의 주소를 담고 있으며 바운드는 그 크기를 나타낸다(즉, 페이지 테이블 항목들의 수).

둘째로, 운영체제는 사용자 페이지 테이블들을(`P0`과 `P1`, 즉 프로세스마다 두 개) 커널의 가상 메모리에 배치하여 메모리 압박을 더 줄일 수 있었다. 페이지 테이블을 할당하거나 크기를 키울 때 커널은 자신의 가상 메모리, 세그멘트 `S` 내에 공간을 할당한다. 메모리가 고갈되면, 커널은 페이지 테이블의 페이지들을 디스크로 스왑하여 물리 메모리를 다른 용도로 사용할 수 있게 한다.

커널 가상 메모리에 페이지 테이블을 넣으면 주소 변환 과정이 훨씬 더 복잡해진다. 예를 들어, `P0`과 `P1` 내의 가상 주소를 변환하기 위해서는 하드웨어가 먼저 페이지 테이블(그 프로세스의 `P0` 또는 `P1` 페이지 테이블)에서 해당 페이지 테이블 항목을 찾아야 한다. 그러나 이 과정 중에 하드웨어는 시스템의 페이지 테이블(물리 메모리에 존재)을 먼저 검색해야 할 수도 있다. 변환이 완료되면 하드웨어는 페이지 테이블 페이지의 주소를 알게 되며, 최종적으로 원하는 메모리 접근에 대한 주소를 알게 된다. 다행스럽게도 이 모든 과정은 VAX의 하드웨어 TLB에서 빠르게 처리된다.



〈그림 26.1〉 **VAX/VMS 주소 공간**

26.3 실제 주소 공간

VMS 학습의 좋은 점은 실제 주소 공간의 구축 상황을 볼 수 있다는 것이다(그림 26.1). 지금까지는 사용자 코드, 사용자 데이터 그리고 사용자 힙을 위한 간단한 주소 공간을 가정하였지만 그림에서 본 것과 같이 실제 주소 공간은 훨씬 더 복잡하다.

예를 들어, 코드 세그멘트는 절대로 페이지 0에서 시작하지 않는다. 대신 이 페이지는 접근 불가능 페이지로 마킹되어 있으며 널-포인터(null-pointer) 접근을 검출할 수 있게 한다. 주소 공간 설계 시 한 가지 고려해야 할 사항은 효과적인 디버깅 지원 여부이다. 접근 불가능한 페이지 0은 그런 지원의 한 형태이다.

좀 더 중요한 사실은 커널의 가상 주소 공간이(즉, 커널의 자료 구조와 코드) 사용자 주소 공간의 일부라는 것이다. 문맥 교환이 발생하면 운영체제는 `P0`과 `P1` 레지스터를 다음 실행될 프로세스의 페이지 테이블을 가리키도록 변경한다. 하지만, `S` 베이스와 바운드 레지스터는 변경하지 않기 때문에 결과적으로 "동일한" 커널 구조들이 각 사용자 주소 공간에 매핑된다.

몇 가지 이유로 커널은 여러 주소 공간들로 매핑된다. 그러한 구조를 택하면 커널이 동작이 쉬워진다. 예를 들어 운영체제가 사용자 프로그램으로부터(예, `write()` 시스템 콜에서) 포인터를 전달 받았다면 그 포인터로부터 데이터를 자신의 구조로 그냥

여담: 널 포인터 접근은 왜 세그멘테이션 오류를 발생시키나

널 포인터 역참조(null-pointer dereference)를 할 때 정확히 어떤 일이 발생하는지 이제 잘 이해하고 있을 것이다. 프로세스는 다음과 같은 동작을 통해서 가상 주소 0 번지를 만들어 낸다.

```
int *p = NULL;  // p = 0 으로 설정
*p = 10;        // 가상 주소 0 에 10을 저장 시도
```

하드웨어는 TLB에서 VPN(여기서도 0) 검색을 시도하지만 TLB 미스를 경험한다. 페이지 테이블이 검색되며 VPN 0에 대한 항목이 무효로 표시되어 있는 것을 알게 된다. 무효한 접근이 발생하였으므로 운영체제로 제어가 전달된다. 그러면 운영체제는 해당 프로세스를 종료할 것이다(Unix 시스템에서는 이런 폴트에 대해서 대응할 수 있도록 프로세스에게 시그널이 전달된다. 시그널을 캐치하지 않으면 프로세스는 종료되게 된다).

복사하면 된다. 쉽다. 운영체제는 접근하는 데이터가 어디에서 오는지 고려할 필요 없이 자연스럽게 작성되고 컴파일될 수 있다. 만약 커널이 전부 물리 메모리에만 존재한다면 페이지 테이블의 페이지들을 디스크로 스왑하는 등의 작업은 상당히 어려웠을 것이다. 커널이 자체 주소 공간을 가진다면 사용자 프로그램들과 커널 간의 데이터 이동이 매우 복잡하고 고통스러운 일이 될 것이다. 이제는 보편적으로 사용되는 이러한 방법 덕택에 커널은 비록 보호된 영역이기는 하지만 응용 프로그램에게 마치 라이브러리처럼 보인다.

이 주소 공간에 관한 마지막 이슈는 보호와 관련 있다. 운영체제는 응용 프로그램이 운영체제의 데이터나 코드를 읽거나 쓰는 것을 분명히 원하지 않는다. 운영체제의 자료를 보호하기 위해서는 하드웨어가 페이지 별로 보호 수준을 다르게 설정할 수 있어야 한다. 이를 위해 VAX는 페이지 테이블의 protection bit(protection bits)에 보호 수준을 지정한다. 특정 페이지를 접근하기 위해서 필요한 CPU의 권한 수준이 기록된다. 시스템 데이터와 코드는 사용자의 데이터와 코드보다 더 높은 보호 수준으로 지정된다. 사용자 코드가 더 높은 보호 수준의 자료를 접근하려고 시도하면 운영체제로 트랩이 걸릴 것이다. 그리고 (예상한대로) 트랩을 일으킨 프로세스는 종료된다.

26.4 페이지 교체

VAX의 페이지 테이블 항목(PTE)은 다음과 같은 비트들을 가지고 있다. 유효(valid) 비트, 보호 필드(protection field, 4비트), 변경(modify 또는 더티(dirty)) 비트, 운영체제가 사용하기 위해 예약해 놓은 필드(5비트), 그리고 마지막으로 물리 메모리 페이지의 위치를 저장하기 위한 물리 프레임 번호(PFN)가 있다. 예리한 독자라면 **reference bit**가 없다는 것을 파악했을 것이다. VMS 교체 알고리즘은 어떤 페이지가 자주 사용 중인지를 하드웨어 지원 없이 판단하여야 한다.

개발자들은 **메모리 호그(memory hog)**에 대해서도 고민하였다. 메모리 호그는 메모리를 너무 많이 사용하는 프로그램을 의미한다. 지금까지 우리가 살펴보았던 대부분의

정책들은 메모리를 많이 소비하는 프로세스에 대한 대비책이 없다. 예를 들어, LRU는 프로세스 간에 공정한 메모리 분배는 고려하지 않는 전역적인(global) 정책이다.

세그멘트된 FIFO

위의 두 가지 문제를 해결하기 위해 개발자들은 **세그멘트된 FIFO** 교체 정책을 제안하였다 [TL81]. 개념은 간단하다. 각 프로세스는 **상주 집합 크기**(**resident set size**, **RSS**)라고 불리는 메모리에 유지할 수 있는 최대 페이지 개수를 지정 받는다. 각 페이지들은 FIFO 리스트에 보관되며, 페이지 개수가 RSS보다 커지면 "제일 먼저 들어왔던" 페이지가 쫓겨난다. FIFO는 하드웨어의 지원이 필요 없기 때문에 구현이 간단하다.

순수한 FIFO 성능은 앞서 보았던 것처럼 그리 좋지는 않다. FIFO의 성능을 개선하기 위해서 VMS는 전역 *클린-페이지 프리 리스트*(*global clean-page free list*)와 *더티-페이지 리스트*(*dirty-page list*)라고 하는 두 개의 **second-chance list**를 도입하였다. 이 리스트는 전역 자료 구조이다. Second-chance list는 메모리에서 제거되기 전에 페이지가 보관되는 리스트이다. 프로세스 `P`가 자신의 RSS를 넘긴다면 자신의 FIFO에서 페이지가 제거된다. 제거된 페이지가 클린 상태라면(수정 안된) 클린-페이지 리스트에, 더티 상태라면 (변경된) 더티-페이지 리스트에 추가된다.

다른 프로세스 `Q`에 빈 페이지가 필요하면 전역 클린 리스트에서 첫 번째 프리 페이지를 꺼낸다. 원래의 프로세스 `P`가 해당 페이지가 회수되기 전에 그 페이지에 대해 폴트를 발생시키면, `P`는 프리(또는 더티) 리스트에서 페이지를 가져가서 다시 사용하게 된다. 이런 식으로 디스크 접근을 피한다. 전역 second-chance list의 크기가 클수록 세그멘트된 FIFO 알고리즘은 LRU와 유사하게 동작한다 [TL81].

페이지 클러스터링

또 다른 최적화 기법은 VMS의 작은 페이지 크기를 극복할 수 있게 하였다. 페이지 크기가 작을수록 스왑할 때에 디스크 I/O가 비효율적이 된다. 디스크는 전송 단위가 클수록 성능이 좋기 때문이다. 스왑할 때 I/O의 효율을 개선하기 위해 VMS는 몇 가지 최적화 기법을 도입하였다. 그 중에서 가장 중요한 것은 **클러스터링**(**clustering**)이다. 클러스터링 기법을 써서 VMS는 전역 더티 리스트에 있는 페이지들을 작업 묶음을 만들어서 한 번에 디스크로 보낸다(그렇게 하여 페이지를 클린 상태로 만든다). 운영체제는 스왑 공간 어디에든 페이지들을 자유롭게 배치할 수 있기 때문에 페이지들을 그룹으로 묶어서 쓰는 것이 가능하다. 쓰기 횟수는 줄이고 한 번에 쓰는 양은 늘려서 성능을 향상시키기 때문에 클러스터링은 대부분의 현대 시스템에서 사용된다.

26.5 그 외의 VM 기법들

VMS는 이제는 표준화가 된 기법 두 가지를 더 가지고 있다. **요청 시 0으로 채우기**(**demand zeroing**)와 **쓰기-시-복사**(**copy-on-write**)가 그 두 가지이다. 이제 이 "**게으른**" 최적화 기법들에 대해 설명한다.

여담: reference bit 에뮬레이트하기

사용 중인 페이지를 찾기 위해서 하드웨어 reference bit가 꼭 필요한 것은 아니라는 것을 알았다. 사실, 1980년대 초에 Babaoglu와 Joy는 VAX의 protection bit가 reference bit를 에뮬레이트할 수 있다는 것을 보였다 [BJ81]. 기본적인 개념은 이렇다. 시스템에서 활발하게 사용 중인 페이지를 알고 싶다면 페이지 테이블의 모든 페이지들을 접근 불가능으로 표시해 놓는다(하지만 프로세스 접근 가능 정보를 별도의 페이지 테이블 항목에 보관한다). 프로세스가 페이지를 접근하면 운영체제로 트랩이 발생할 것이다. 운영체제는 페이지가 정말 접근 가능한지 검사한다. 가능하다면 해당 페이지를 정상적인 보호 상태로 되돌린다(예, 읽기 전용 또는 읽기-쓰기). 페이지 교체 시점에 운영체제는 접근 불가능으로 남아 있는 페이지들을 확인하여 최근에 사용되지 않은 페이지들이 어떤 것들인지를 파악한다.

reference bit "에뮬레이션"은 오버헤드를 줄이는 것이 핵심이다. 운영체제는 너무 자주 페이지를 접근 불가능으로 표시하면 안 된다. 오버헤드가 증가한다. 그렇다고 표시를 너무 주저하면 모든 페이지들이 사용된 상태로 남게 될 것이다. 운영체제는 여전히 어떤 페이지를 제거해야 할지 모르게 된다.

VMS(그리고 대부분의 현대의 시스템들)가 사용하는 기법의 한 형태는 페이지들을 **demand zeroing**이다. 이해를 위해 힙 등의 주소 공간에 페이지를 추가하는 예를 생각해 보자. 단순한 구현에서는 힙에 페이지를 추가하는 요청이 오면 운영체제는 물리 메모리에서 페이지를 찾아 0으로 채운다(보안을 위해 필요하다. 그렇지 않으면 다른 프로세스가 이전에 사용했던 페이지의 내용을 볼 수 있게 된다!). 그런 후에 주소 공간에 그 페이지를 매핑한다(즉, 물리 페이지를 원하는 대로 참조할 수 있도록 페이지 테이블을 설정한다). 하지만 이 경우 프로세스가 해당 페이지를 사용하지 않는다면 너무 많은 비용을 지불하는 셈이다.

Demand zeroing의 경우 페이지가 주소 공간에 추가되는 시점에는 거의 하는 일이 없다. 페이지 테이블에 접근 불가능 페이지라고 표기하고 항목을 추가한다. 프로세스가 추가된 페이지를 읽거나 쓸 때 운영체제로 트랩이 발생한다. 트랩을 처리하면서 운영체제는 demand zeroing 할 페이지라는 것을 알게 된다(일반적으로는 페이지 테이블 항목의 "운영체제를 위해 예약된" 부분에 일정 비트로 표기되어 있다). 이 시점에서 운영체제는 물리 페이지를 0으로 채우고 프로세스의 주소 공간으로 매핑하는 등의 필요한 작업을 한다. 프로세스가 해당 페이지를 전혀 접근하지 않는다면 이 모든 작업을 피할 수 있으며, 이것이 바로 demand zeroing의 장점이다.

VMS에서 찾아 볼 수 있는 또 다른 멋진 최적화 방법은(마찬가지로 거의 대부분의 모든 현대의 운영체제에서 찾아 볼 수 있는) **copy-on-write**(**COW**)이다. 개념 자체는 TENEX 운영체제까지 거슬러 올라가며 [Bob+72], 개념도 간단하다. 운영체제가 한 주소 공간에서 다른 공간으로 페이지를 복사할 필요가 있을 때, 복사를 하지 않고 해당 페이지를 대상 주소 공간으로 매핑하고[1] 해당 페이지의 페이지 테이블 엔트리를 양쪽

1) 역자 주: 즉, 대상 페이지 테이블의 해당 페이지 엔트리가 원본의 물리 위치를 가리키게 한다. 하나의

팁: 게으름을 피우자

게으름을 피우는 것이 인생에서나 운영체제에서 덕목이 될 수도 있다. 게으름을 피우면 일을 미룰 수가 있는데, 운영체제의 경우에는 몇 가지 이유로 유익하다. 먼저 일을 미루면 현재 작업의 지연 시간을 줄일 수 있게 되어 응답성을 개선할 수 있다. 예를 들어 대부분의 경우 운영체제가 파일에 대한 쓰기가 성공하였다고 즉시 보고 하지만, 실제 디스크 쓰기는 나중에 백그라운드로 처리된다. 둘째 그리고 더 중요한 이유는 게으름으로 인해서 작업을 해야 할 필요 자체를 제거할 수 있기 때문이다. 예를 들어서 파일이 지워질 때까지 쓰기를 지연하면 쓰기 자체가 필요 없어진다. 게으름은 삶에서도 좋다. 예를 들면, 운영체제 프로젝트를 미루면, 해당 프로젝트가 가지고 있던 명세의 버그를 반 친구들이 고쳐줄 때가 있기 때문이다. 하지만, 프로젝트가 취소되는 경우는 거의 없기 때문에 게으름을 피우는 것은 문제가 될 수 있다. 프로젝트 기한이 지난 뒤에 제출하게 되고, 성적이 떨어지고, 교수님도 슬프게 된다. 교수님들을 슬프게 만들지 말자!

주소 공간에서 읽기 전용으로 표시한다. 만약 양쪽 주소 공간이 페이지를 읽기만 한다면 더 이상의 조치는 필요 없게 되며, 운영체제는 실제로 데이터 이동 없이 빠른 복사를 할 수 있게 된다.

두 주소 공간 중에 하나가 페이지 쓰기를 시도한다면, 운영체제로 트랩을 발생한다. 운영체제는 그때 해당 페이지가 COW 페이지라는 것을 파악한다. 그런 후에 새로운 페이지를 (게으르게) 할당하고, 데이터로 채우고, 이 새로운 페이지 폴트를 일으킨 페이지의 주소 공간에 매핑한다. 이제 프로세스는 독자적인 페이지의 사본을 가지게 되고 하던 일을 계속한다.

COW는 여러 가지 이유로 유용하다. 공유 라이브러리들을 여러 프로세스들의 주소 공간에 copy-on-write로 매핑하여 메모리 공간을 절약할 수 있다. Unix 시스템에서는 `fork()`와 `exec()`의 시맨틱 때문에 COW는 훨씬 더 중요하다. 기억하겠지만 `fork()`는 호출자의 주소 공간과 정확히 동일한 사본을 생성한다. 주소 공간이 크면 복사본을 만드는 시간이 오래 걸리고 짧은 시간에 많은 양의 데이터를 접근해야 한다. 더 심각한 것은, 뒤 따르는 `exec()` 호출에 의해서 대부분의 주소 공간이 덮어 쓰이게 된다. `exec()`은 곧 실행될 프로그램의 주소 공간으로 호출한 프로세스의 주소 공간의 내용을 덮어 쓴다. copy-on-write `fork()`를 수행하게 되면 운영체제는 상당한 불필요한 복사를 피할 수 있으며, 성능을 개선하면서 정확한 시맨틱을 유지할 수 있다.

26.6 요약

전체 가상 메모리 시스템에 대하여 처음부터 끝까지 살펴보았다. 대부분의 기본적인 기법과 정책들을 잘 이해하고 있었으므로, 상세 사항 대부분을 이해하기 쉬웠을 것이다.

물리 페이지를 두 개의 페이지 테이블이 가리키는 것이다.

좀 더 상세한 내용은 훌륭한(그리고 짧은) Levy와 Lipman의 논문에 나와 있다 [LL82]. 읽어보기를 권한다. 이번 장에 대한 원전을 볼 수 있는 좋은 기회이다.

가능하다면 Linux와 다른 현대 시스템을 읽어 최신 기술에 대해서 학습하기 바란다. 어렵지 않게 쓰여진 책을 포함하여 [BC05] 상당한 읽을 내용들이 있다. 놀랄만한 한 가지 사실: VAX/VMS와 같이 오래된 논문들에 나와 있는 고전적인 개념들이 아직도 현대 운영체제의 개발에 영향을 주고 있다.

참고 문헌

[BJ81] **"Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits"**
Ozalp Babaoglu and William N. Joy
SOSP '81, Pacific Grove, California, December 1981
reference bit를 에뮬레이트하기 위해서 기존의 보호 기법을 활용하는 방법을 다룬 영리한 개념을 소개하는 논문. 버클리 시스템 배포(Berkeley Systems Distribution) 또는 BSD라고 부르는 자체적인 Unix 버전을 개발하던 그룹에서 나온 아이디어. 이 그룹은 Unix 발전 특히 가상 메모리와 파일 시스템 그리고 네트워크 분야 개발에 엄청난 영향을 끼쳤다.

[Bob+72] **"TENEX, A Paged Time Sharing System for the PDP-10"**
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson
Communications of the ACM, Volume 15, March 1972
수많은 좋은 개념들을 고안해 낸 초기의 시분할 운영체제이다. Copy-on-write는 그 중에 하나일 뿐이다. 현대 운영체제의 많은 다른 측면에 영감을 준 시스템. 이 책의 프로세스 관리, 가상 메모리 및 파일 시스템도 그 중 하나다.

[BC05] **"Understanding the Linux Kernel (Third Edition)"**
Daniel P. Bovet and Marco Cesati
O'Reilly Media, November 2005
Linux에 대하여 찾아 볼 수 있는 많은 책들 중 하나. 많은 책들이 금세 구닥다리가 되기는 하지만 많은 기본적인 개념들은 여전히 유지되며 읽을 가치가 충분하다.

[Chr03] **"The Innovator's Dilemma"**
Clayton M. Christenson
Harper Paperbacks, January 2003
디스크 드라이브 산업과 새로운 혁신이 기존의 혁신을 어떻게 붕괴시키는지를 보여주는 환상적인 책이다. 경영 전공 학생들뿐만 아니라 컴퓨터 과학자들에게도 읽기 좋은 책이다. 거대하고 성공적인 기업들이 어떻게 완전히 망하는지에 대한 통찰을 제공한다.

[CS93] **"Inside Windows NT"**
Helen Custer and David Solomon
Microsoft Press, 1993
Windows NT에 대한 책. 시스템의 꼭대기부터 바닥까지 원하는 것 이상보다 더 자세하게 설명한다. 그렇지만 진지하게, 이 책은 상당히 좋은 책이다.

[LL82] **"Virtual Memory Management in the VAX/VMS Operating System"**
Henry M. Levy and Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
이번 장의 대부분의 내용을 다루는 원전을 읽어라. 간결하게 쉽게 읽을 수 있다. 대학원에 진학할 예정이라면 특히 중요하다. 대학원에서 하는 일이라고는 논문을 읽고, 연구하고, 더 많은 논문들을 읽고, 좀 더 연구하고, 결국에는 논문도 쓰고, 그런 후 연구를 더 하고. 그러나 재미있다.

[TL81] **"Segmented FIFO Page Replacement"**
Rollins Turner and Henry Levy
SIGMETRICS '81, Las Vegas, Nevada, September 1981
어떤 워크로드에 대해서는 세그멘트된 FIFO가 LRU의 성능에 근접한다는 것을 보인 짧은 논문.