24

물리 메모리 크기의 극복 : 메커니즘

지금까지 가상 주소 공간이 비현실적으로 작아서 모두 물리 메모리에 탑재가 가능한 것으로 가정하였다. 사실 실행 중인 프로세스의 *전체* 주소 공간이 메모리에 탑재된 것으로 가정하고 있었다. 이제 그 가정을 완화한다. 우리는 다수 프로세스들이 동시에 각자 큰 주소 공간을 사용하고 있는 상황을 가정한다.

이를 위해, **메모리 계층**에 레이어의 추가가 필요하다. 지금까지는 모든 페이지들이 물리 메모리에 존재하는 것을 가정하였다. 하지만 큰 주소 공간을 지원하기 위해서 운영체제는 주소 공간 중에 현재는 크게 필요하지 않은 일부를 보관해 둘 공간이 필요하다. 일반적으로 그 공간은 메모리 공간보다 더 크며, 더 느리다(만약 더 빠르다면 그것을 메모리로 사용할 것이다. 그렇지 않을까?). 현대 시스템에서는 보통 **하드 디스크 드라이브**가 이 역할을 담당한다. 이제 메모리 계층에서 크고 느린 하드 디스크 드라이브가 가장 하부에 위치하고, 그 위에 메모리가 있다. 이렇게 하면 우리의 질문은 다음과 같이 변형된다.

핵심 질문 : 물리 메모리 이상으로 나아가기 위해서 어떻게 할까
운영체제는 어떻게 크고 느린 장치를 사용하면서 마치 커다란 가상 주소 공간이 있는 것처럼 할 수 있을까?

한 가지 짚고 넘어가야 할 것은, 왜 프로세스에게 굳이 "큰" 주소 공간을 제공해야 하는가이다. 이에 대한 답은 다시 한 번 편리함과 사용 용이성이다. 주소 공간이 충분히 크면, 프로그램의 자료 구조들을 위한 충분한 메모리 공간이 있는지 걱정하지 않아도 된다. 필요 시 메모리 할당을 운영체제에게 요청하기만 하면 된다. 운영체제가 이러한 "가상" 환경을 제공하면, 인생이 편해진다. 이와 대비되는 기법으로 과거 시스템에서 사용되던 **메모리 오버레이**(**memory overlay**)가 있다. 이 시스템에서는 프로그래머가 코드 또는 데이터의 일부를 수동으로 메모리에 탑재/제거했다 [Den97]. 상상만 해도 힘들다. 함수 호출이나 데이터 접근 시, 코드 또는 데이터를 먼저 메모리에 탑재해야 한다. 헉!

스왑 공간이 추가되면 운영체제는 실행되는 각 프로세스들에게 큰 가상 메모리가 있는 것 같은 환상을 줄 수 있다. 멀티프로그래밍 시스템(컴퓨터의 사용률을 높이기

여담: 저장 장치 기술들

I/O 장치의 동작 메커니즘은 추후 깊게 살펴보도록 하겠다(I/O 장치들에 대한 장을 보자). 조금만 기다리기 바란다. 느린 장치가 꼭 하드 디스크 드라이브일 필요는 없다. 좀 더 현대적인 플래시 기반의 SSD일 수도 있다. 그것에 대해서도 다룰 것이다. 이 시점에서는 대형 가상 공간을 위해, "크고 느린 저장 장치가 있다." 라고만 가정하겠다.

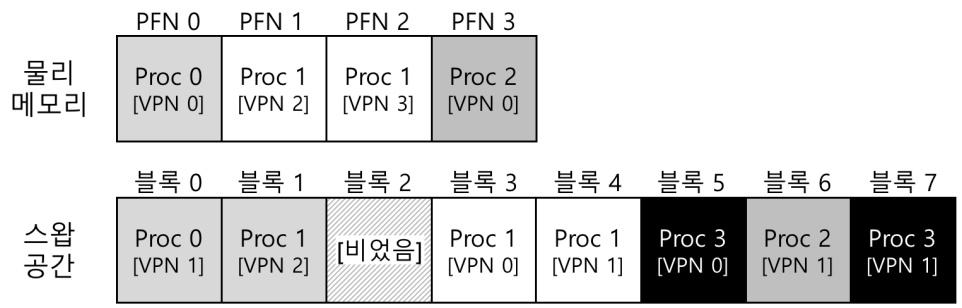
위해 "동시에" 여러 프로그램들을 실행시키는 시스템)이 발명되면서 많은 프로세스들의 페이지를 물리 메모리에 전부 저장하는 것이 불가능하게 되었다. 그래서 일부 페이지들을 스왑 아웃하는 기능이 필요하게 되었다. 멀티프로그래밍과 사용 편의성 등의 이유로 실제 물리 메모리보다 더 많은 용량의 메모리가 필요하게 되었다. 이것이 현대 Virtual Memory(가상 메모리)의 역할이다. 이제 그 방식을 배워보도록 하겠다.

24.1 스왑 공간

가장 먼저할 일은 디스크에 페이지들을 저장할 수 있는 일정 공간을 확보하는 것이다. 이 용도의 공간을 **스왑 공간(swap space)**이라고 한다. 스왑 공간이라 불리는 이유는 메모리 페이지를 읽어서 이곳에 쓰고(*swap out*), 여기서 페이지를 읽어 메모리에 탑재시키기(*swap in*) 때문이다. 스왑 공간의 입출력 단위는 페이지라고 가정한다. 운영체제는 스왑 공간에 있는 모든 페이지들의 **디스크 주소**를 기억해야 한다.

스왑 공간의 크기는 매우 중요하다. 시스템이 사용할 수 있는 메모리 페이지의 최대수를 결정하기 때문이다. 일단 문제를 단순화하기 위해, 스왑 공간은 *매우* 크다고 가정하자.

간단한 예제(그림 24.1)를 보도록 하자. 물리 메모리와 스왑 공간에는 각각 4개의 페이지와 8개의 페이지를 위한 공간이 존재한다. 이 예에서는 세 개의 프로세스(Proc 0, Proc 1, 그리고 Proc 2)가 물리 메모리를 공유하고 있다. 이들 세 프로세스는 몇 개의 유효한 페이지들만 메모리에 올려 놓았으며 나머지 페이지들은 디스크에 스왑 아웃되어 있다. 네 번째 프로세스(Proc 3)의 모든 페이지들은 디스크로 스왑 아웃되어 있기 때문에



〈그림 24.1〉 **물리 메모리와 스왑 공간**

현재 실행 중이 아닌 것이 분명하다. 스왑 영역에 하나의 블럭이 비어있다. 예제를 통해, 스왑 공간을 이용하면, 시스템에 실제 물리적으로 존재하는 메모리 공간보다 더 많은 공간이 존재하는 것처럼 가장할 수 있다는 것을 알 수 있다.

스왑 공간에만 스왑을 할수 있는 것은 아니라는 사실에 유념하자. 예를 들어, 프로그램을 실행한다고 하자(예, `ls` 또는 당신이 컴파일한 `main` 프로그램). 이 실행프로그램의 페이지들은 디스크에 존재한다. 프로그램이 실행되면 각 페이지들은 메모리로 탑재된다(프로그램 실행 시 한 번에 모두 탑재되거나 또는 필요 시 한 페이지씩 탑재할 수 있다). 물리 메모리에 추가 공간을 확보해야 할 때, 코드영역의 페이지들이 차지하는 물리 페이지는 즉시 다른 페이지가 사용할 수 있다. 코드가 저장되어 있는 파일 시스템 영역이 스왑 목적으로 사용되는 셈이다. 해당 페이지들은 디스크에 원본이 있으므로, 언제든지 다시 스왑-인이 가능하기 때문이다.

24.2 Present Bit

디스크에 스왑 공간을 확보했으니 이제 페이지 스왑을 위한 기능을 다룰 차례다. 하드웨어 기반의 TLB를 사용하는 시스템을 가정하자.

먼저 메모리가 참조되는 과정을 상기해 보자. 프로세스가 가상 메모리 참조를 생성한다(명령어 탑재나 데이터 접근 등). 하드웨어는 메모리에서 원하는 데이터를 가져오기 전에, 우선 가상 주소를 물리 주소로 변환한다.

하드웨어는 먼저 가상 주소에서 VPN을 추출한 후에 TLB에 해당 정보가 있는지 검사한다(**TLB 히트**). 만약 히트가 되면 물리 주소를 얻은 후에 메모리로 가져온다. 매우 빠르다. 대부분의 경우가 여기에 해당하기를 바란다(추가적인 메모리 접근이 필요 없다).

만약 VPN을 TLB에서 찾을 수 없다면(즉, **TLB 미스**), 하드웨어는 페이지 테이블의 메모리 주소를 파악하고(**페이지 테이블 베이스 레지스터**를 사용), VPN을 인덱스로 하여 원하는 **페이지 테이블 항목**(**PTE**)을 추출한다. 해당 페이지 테이블 항목이 유효하고, 관련 페이지가 물리 메모리에 존재하면 하드웨어는 PTE에서 PFN 정보를 추출하고, 그 정보를 TLB에 탑재한다. TLB 탑재 후 명령어를 재실행한다. 이번에는 TLB에서 히트된다. 여기까지는 순조롭다.

페이지가 디스크로 스왑되는 것을 가능케 하려면, 많은 기법들이 추가되어야 한다. 특히, 하드웨어가 PTE에서 해당 페이지가 물리 메모리에 존재하지 않는다는 것을 표현해야 한다. 하드웨어는(또는 소프트웨어가 관리하는 TLB 기법인 경우에는 운영체제가) **present bit**를 사용하여 각 페이지 테이블 항목에 어떤 페이지가 존재하는지를 표현한다. Present 비트가 1로 설정되어 있다면, 물리 메모리에 해당 페이지가 *존재한다*는 것이고 위에 설명한 대로 동작한다. 만약 그 비트가 0으로 설정되어 있다면, 메모리에 해당 페이지가 존재하지 않고 디스크 어딘가에 존재한다는 것을 나타낸다. 물리 메모리에 존재하지 않는 페이지를 접근하는 행위를 일반적으로 **페이지 폴트**(**page fault**)라 한다.

페이지 폴트가 발생하면, 페이지 폴트를 처리하기 위해 운영체제로 제어권이 넘어간다. **페이지 폴트 핸들러**(**page-fault handler**) 가 실행된다. 그 세부내용을 다음에서

여담: 스와핑 기술 그리고 다른 것들

가상 메모리 시스템의 용어들은 운영체제와 사용하는 기기에 따라서 약간 혼란스럽고 변종이 많다. 예를 들어 **페이지 폴트**는 페이지 테이블 참조 시 발생가능한 다양한 종류의 오류를 의미할 수 있다. 이 오류에는 지금 여기서 우리가 다루고 있는 종류의 오류, 즉 페이지-존재하지-않음 오류, 포함할 수도 있지만 때로는 불법적인 메모리 접근을 나타낼 수도 있다. 사실, 분명하게 합법적인 접근을 "오류"라고 부르는 것이 이상하다. 실제로는 **페이지 미스**라고 불려야 할 것이다. 그렇지만 대체적으로 사람들이 어떤 프로그램에서 "페이지 폴트"가 발생하였다고 말할 때 의미하는 것은 운영체제가 디스크로 스왑 아웃한 가상 주소 공간의 한 부분을 접근하였다는 것이다.

이러한 동작이 "오류"라고 불리게 된 이유를 운영체제에 이것을 다루기 위해 적용한 기법과 관계가 있다고 추정한다. 일반적이지 않은 어떤 일이 발생한다면, 즉 하드웨어가 어떻게 다뤄야 할지 모르는 어떤 일이 발생한다면, 상황 개선을 기대하며 하드웨어는 제어권을 운영체제에게 넘긴다. 프로세스가 원하는 페이지가 메모리에 없다. 하드웨어는 "예외"를 발생시키며, 그 이후부터는 운영체제가 처리를 담당한다. 프로세스가 불법적인 일을 할 때 그를 처리하는 방법과 동일하기 때문에, 이 동작을 "오류"라고 해도 그리 이상하지 않다.

살펴보도록 하자.

24.3 페이지 폴트

TLB 미스의 처리 방법에 따라, 두 종류의 시스템이 있었다. 하드웨어 기반의 TLB(하드웨어가 페이지 테이블을 검색하여 원하는 변환 정보를 찾는 것)와 소프트웨어 기반의 TLB(운영체제가 처리하는 것)이다. 둘 중 어느 것이던, 페이지 폴트가 발생하면, 운영체제가 그 처리를 담당한다. 운영체제의 **페이지 폴트 핸들러**가 그 처리 메커니즘을 규정한다. 페이지 폴트 핸들러, 이름 참 잘 지었다. 거의 대부분의 시스템들에서 페이지 폴트는 소프트웨어적으로 처리된다. 하드웨어 기반의 TLB도 페이지 폴트 처리는 운영체제가 담당한다.

만약 요청된 페이지가 메모리에 없고, 디스크로 스왑되었다면, 운영체제는 해당 페이지를 메모리로 스왑해 온다. 자연적으로 등장하는 질문이 있다. 원하는 페이지의 위치를 어떻게 파악할지? 많은 시스템들에서 해당 정보—즉 해당 페이지의 스왑 공간상에서의 위치—를 페이지 테이블에 저장한다. 운영체제는 PFN과 같은 PTE비트들을 페이지의 디스크 주소를 나타내는 데 사용할 수 있다. 페이지 폴트 발생 시, 운영체제는 페이지 테이블 항목에서 해당 페이지의 디스크 상 위치를 파악하여, 메모리로 탑재한다.

디스크 I/O가 완료가 되면 운영체제는 해당 페이지 테이블 항목(PTE)의 PFN 값을 탑재된 페이지의 메모리 위치로 갱신한다. 이 작업이 완료되면 페이지 폴트를 발생시킨 명령어가 재실행된다. 재실행으로 인해 TLB 미스가 발생될 수 있다. TLB 미스 처리 과정에서 TLB 값이 갱신된다(이를 피하기 위한 대안으로 페이지 폴트를 처리 시 함께

여담: 페이지 폴트를 하드웨어로 처리하지 않는 이유

TLB를 다뤄본 경험에서 우리는 하드웨어 설계자들이 운영체제가 하는 일을 신뢰하고 싶어하지 않는다는 것을 알았다. 어째서 운영체제에게 페이지 폴트 처리를 맡길까? 몇 가지 중요한 이유들이 있다. 첫째는 페이지 폴트로 인한 디스크 접근이 느리기 때문이다. 페이지 폴트 처리에 시간이 소요된다 하더라도, 디스크 작업 자체가 너무 느리기 때문에 소프트웨어를 실행하는 추가 부담은 절대적으로 작다. 둘째로 페이지 폴트 처리를 위해서는 하드웨어가 스왑 공간의 구조, 디스크에 I/O를 요청하는 방법, 그리고 그 외에 하드웨어가 파악하지 못하고 있는 세부적인 것들을 모두 알고 있어야 한다. 성능과 간편성이라는 두 가지 장점으로, 페이지 폴트 처리를 운영체제가 담당하게 되었고, 하드웨어 설계자도 이것을 만족하게 되었다.

TLB를 갱신하도록 할 수도 있다). 최종적으로, 마지막 재실행 시에 TLB에서 주소 변환 정보를 찾게 되고, 이를 이용하여 물리 주소에서 원하는 데이터나 명령어를 가져온다.

I/O 전송 중에는 해당 프로세스가 **차단된**(**blocked**) 상태가 된다는 것을 유의해야 한다. 페이지 폴트 처리 시 운영체제는 다른 프로세스들을 실행할 수 있다. I/O 실행은 매우 시간이 많이 소요되기 때문에 한 프로세스의 I/O 작업(페이지 폴트)과 다른 프로세스의 실행을 **중첩**(**overlap**)시키는 것은 멀티 프로그램된 시스템에서 하드웨어를 최대한 효율적으로 사용하는 방법 중 하나다.

24.4 메모리에 빈 공간이 없으면?

위 설명에서, 스왑 공간으로부터 페이지를 가져오기 위한 (**page-in**) 여유 메모리가 충분하다는 것을 가정했다. 항상 이런 경우만 있는 것은 아니다. 메모리에 여유 공간이 없을 수도 있다(또는 거의 다 찼을 수도 있다). 탑재하고자 하는 새로운 페이지(들)를 위한 공간을 확보하기 위해 하나 또는 그 이상의 페이지들을 먼저 **페이지 아웃**(**page out**)하려고 할 수도 있다. **교체**(**replace**) 페이지를 선택하는 것을 **페이지 교체 정책**(**page-replacement policy**)이라고 한다.

잘못된 선택은 프로그램의 성능에 큰 악영향을 미친다. 좋은 페이지 교체 정책을 만들기 위해 많은 노력들이 있었다. 잘못된 선택을 할 경우, 프로그램이 메모리 속도로 실행되는 것이 아니라 디스크와 비슷한 속도로 동작할 수도 있기 때문이다. 현재 기술에서 볼 때, 프로그램이 10,000 또는 100,000 배 더 느리게 실행된다는 것을 의미한다. 페이지 교체 정책은 면밀히 검토되어야 한다. 다음 장에서 각종 다른 정책들에 대해서 자세히 다룬다. 현재로서는 페이지 교체 정책이라는 것이 존재한다는 사실과 그 기법이 이전에 다룬 스와핑과 함께 동작한다는 점만 인지하고 넘어간다.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)        // TLB 히트
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = \gndx{AccessMemory}(\gndx{PhysAddr})
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                        // TLB 미스
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = \gndx{AccessMemory}(\gndx{PTEAddr})
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else
16         if (CanAccess(PTE.ProtectBits) == False)
17             RaiseException(PROTECTION_FAULT)
18         else if (PTE.Present == True)
19             // 하드위에를 기반으로 한 TLB를 가정함
20             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21             RetryInstruction()
22         else if (PTE.Present == False)
23            RaiseException(PAGE_FAULT)
```

〈그림 24.2〉 **페이지 오류 제어 흐름의 알고리즘(하드웨어)**

24.5 페이지 폴트의 처리

이제 메모리 접근이 이루어지는 전체 과정을 파악할 수 있게 되었다. "프로그램이 메모리에서 데이터를 가져올 때 어떤 일이 발생하는가?"라는 질문을 받았을 때, 다양한 상황에 대해 심도 있는 설명이 가능하게 된 것이다. 그림 24.2와 그림 24.3은 페이지 폴트 처리의 상세한 과정을 나타낸다. 첫 그림은 하드웨어를 통한 주소 변환을 나타낸다. 두 번째 그림은 페이지 폴트 발생 시 운영체제의 동작(소프트웨어)을 나타낸다.

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)                    // 비어있는 페이지 못 찾음
3      PFN = EvictPage()             // 교체 알고리즘 실행
4      \gndx{DiskRead}(\gndx{PTE.DiskAddr, pfn})    // 대기 (I/O를 기다리기)
5      PTE.present = True            // 존재한다고 페이지 테이블에 갱신
6      PTE.PFN = PFN                 // 비트와 변환 (PFN)
7      RetryInstruction()            // 명령어 재시도
```

〈그림 24.3〉 **페이지 오류 제어 흐름의 알고리즘(소프트웨어)**

그림 24.2의 하드웨어 처리 과정에서, TLB 미스 발생 시, 세 가지의 중요한 경우가 있다는 것을 알 수 있다. 첫째는 페이지가 **존재**하며 **유효**한 경우이다(18-21 라인). 이 경우에는 TLB 미스 핸들러가 PTE에서 PFN을 가져와서 (여러 차례) 명령어를 재시도한다(이때에는 TLB 히트가 된다). 두 번째 경우에는(22-23 라인) 페이지가 유효하지만 존재하지 않는 경우다. 페이지 폴트 핸들러가 반드시 실행되어야 한다. 프로세스가 사용할 수 있는 제대로 된 페이지이기는 하지만(유효한 경우이기 때문에) 물리 메모리에 존재하지 않기 때문이다. 세 번째, 마지막으로 페이지가 유효하지 않는 경우다. 프로그램 버그 등으로 잘못된 주소를 접근하는 경우의 처리를 나타낸다(13-14 라인). 이 경우에는 PTE의 다른 비트는 의미가 없다. 하드웨어는 이 무효한 접근이 운영체제의 트랩 핸들

러에 의해서 처리되도록 한다. 이때 문제를 일으킨 프로세스는 종료될 수가 있다.

그림 24.3에서 운영체제가 페이지 폴트를 처리하는 과정을 대략적으로 볼 수 있다. 먼저 운영체제는 탑재할 페이지를 위한 물리 프레임을 확보한다. 만약 여유 프레임이 없다면, 교체 알고리즘을 실행하여 메모리에서 페이지를 내보내고 여유 공간을 확보한다. 물리 프레임을 확보한 후, I/O 요청을 통해 스왑 영역에서 페이지를 읽어 온다. 마지막으로 이 느린 작업이 완료되면 운영체제는 페이지 테이블을 갱신하고 명령어를 재시도한다. 재시도를 하게 되면 TLB 미스가 발생하며, 또 한 번의 재시도를 할 때에 TLB 히트가 된다. 그제서야 하드웨어는 원하는 것을 접근할 수 있게 된다.

24.6 교체는 실제 언제 일어나는가

이제까지의 설명에서는 메모리에 여유 공간이 고갈된 후에 교체 알고리즘이 작동하는 것을 가정하였다. 이 방법은 그리 효율적이지 않다. 뿐만 아니라, 다양한 이유로 인해, 운영체제는 항상 어느 정도의 여유 메모리 공간을 확보하고 있어야 한다.

메모리에 항상 어느 정도의 여유 공간을 비워두기 위해서, 대부분의 운영체제들은 여유 공간에 관련된 **최댓값**(**high watermark**, HW)과 **최솟값**(**low watermark**, LW)을 설정하여 교체 알고리즘 작동에 활용한다. 동작 방식은 다음과 같다. 운영체제가 여유 공간의 크기가 최솟값보다 작아지면 여유 공간 확보를 담당하는 백그라운드 쓰레드가 실행된다. 이 쓰레드는 여유 공간의 크기가 최댓값에 이를 때까지 페이지를 제거한다. 이 백그라운드 쓰레드는 일반적으로 **스왑 데몬**(**swap daemon**) 또는 **페이지 데몬**[1] (**page daemon**)이라고 불린다. 충분한 여유 메모리가 확보되면 이 백그라운드 쓰레드는 슬립 모드로 들어간다.

일시에 여러 개를 교체하면 성능 개선이 가능하다. 예를 들어 많은 시스템들은 다수의 페이지들을 **클러스터**(**cluster**)나 **그룹**(**group**)으로 묶어서 한 번에 스왑 파티션에 저장함으로써 디스크의 효율을 높인다 [LL82]. 디스크 관련 부분에서 좀 더 자세히 다루겠지만, 클러스터링은 디스크의 탐색(seek)과 회전 지연(rotational delay)에 대한 오버헤드를 경감시켜 성능을 상당히 높일 수 있다.

백그라운드 페이징 쓰레드를 사용하기 위해서는 그림 24.3의 과정이 약간 변경되어야 한다. 교체를 직접 수행하는 대신 알고리즘은 사용할 수 있는 페이지들이 있는지를 단순히 검사만 한다. 만약 없다면 백그라운드 페이징 쓰레드에게 여유 페이지들이 필요하다고 알려준다. 쓰레드가 페이지들을 비운 후에 원래의 쓰레드를 다시 깨워서 원하는 페이지를 불러들일 수 있도록 하며 계속 작업을 진행할 수 있도록 한다.

1) ‘‘데몬(daemon)”이라는 단어는 일반적으로 “데몬(demon)”으로 발음되는데, 이것은 유용한 어떤 동작을 하는 백그라운드 쓰레드 또는 프로세스를 나타내는 오래된 용어이다. (또 다시!) 이 용어의 근원은 Multics에서 시작되었다 [CS94].

팁: 백그라운드에서 작업을 하자

할 일이 있을 때, 그 일을 **백그라운드**로 처리하고, 여러 가지를 묶어서 한 번에 처리하는 것이 효율적일 수 있다. 운영체제는 자주 백그라운드에서 작업을 처리한다. 예를 들어, 많은 시스템들은 파일을 쓸 때에 데이터를 디스크에 실제로 쓰기 전에 메모리의 버퍼에 먼저 쓴다. 그렇게 하면 여러 가지 이득을 얻을 수가 있다. 디스크가 여러 쓰기 요청을 받지 않기 때문에 스케줄을 좀 더 잘할 수 있게 되어 디스크의 효율을 높일 수 있다. 또한, 응용 프로그램은 쓰기가 빠르게 완료된 것처럼 생각하도록 만들기 때문에 쓰기에 대한 지연 시간이 짧아진다. 디스크로 쓰기가 즉시 전달될 필요가 없을 수도 있기 때문에(파일이 삭제된 경우) 작업량이 줄어들 수도 있다. 백그라운드 작업은 대체적으로 시스템이 **유휴 상태**에 있을 때 진행된다. 시스템이 놀고 있는 시간을 잘 활용할 수 있게 된다. 하드웨어를 더 잘 활용할 수 있게 된다 [Gol+95].

24.7 요약

이 장에서는 시스템에 실제 존재하는 물리 메모리의 크기보다 더 많은 메모리를 사용하기 위한 개념을 소개하였다. 이를 위해서 메모리에 특정 페이지가 존재하는지를 알리기 위한 **present bit**와 좀 더 복잡한 페이지 테이블 구조가 필요하다. 운영체제는 **페이지 폴트(page fault)**를 처리하기 위해서 **페이지 폴트 핸들러(page-fault handler)**를 실행시킨다. 핸들러는 원하는 페이지를 디스크에서 메모리로 전송하기 위해 메모리의 일부 페이지들을 먼저 교체하여 새롭게 스왑되서 들어올 페이지를 위한 공간을 만드는 조치를 취한다.

중요하게(그리고 놀랍게!) 이 모든 작업은 프로세스가 인지하지 못하는 상황에서 처리된다는 것을 기억해야 한다. 프로세스가 보기에는 자신의 개별적인 연속된 가상 메모리를 접근하는 것처럼 보인다. 실제로 페이지들은 물리 메모리 임의의(불연속적인) 위치에 배치되며, 때로는 디스크에서 가져와야 할 때도 있다. 일반적인 경우의 메모리 접근 작업이 빠르다고 기대하지만, 어떤 경우 여러 번의 디스크 작업 후에 처리가 되기도 한다. 최악의 경우에는 하나의 명령어를 처리하는 간단한 작업이라 할지라도 수 millisecond 가 걸려야 완료될 수도 있다.

참고 문헌

[CS94] **"Take Our Word For It"**
F. Corbato and R. Steinberg
URL: `http://www.takeourword.com/TOW146/page4.html`
Richard Steinberg는 "컴퓨팅에 사용되고 있는 데몬이라는 단어의 기원에 대해서 누군가 물어보았어요. 내가 조사한 내용에 따르면, 1963년에 IBM 7094를 사용하는 프로젝트 MAC 의 당신의 팀원들이 처음 그 단어를 사용한 것으로 보입니다." 라고 썼다. 그에 대해 Corbato 교수가 다음과 같이 답신을 하였다. "물리와 열역학(내 전공이 물리학이에요)에서 사용하는 Maxwell 의 데몬에서 영감을 얻어서 데몬을 사용하기 시작했어요. Maxwell의 데몬은 상상의 요원으로서로 다른 속도를 갖는 분자들을 정렬하는 일을 위해 백그라운드에서 지지치 않고 돕는 것이죠. 시스템의 허드렛일을 지치지 않고 작업하는 백그라운드 프로세스들을 설명하기 위해 데몬이라는 멋진 이름을 붙여서 사용하기 시작했습니다."

[Den97] **"Before Memory Was Virtual"**
Peter Denning
From In the Beginning: Recollections of Software Pioneers, Wiley, November 1997
가상 메모리와 작업 세트를 개발한 선구자들 중의 하나로서 훌륭한 역사적인 자료이다.

[Gol+95] **"Idleness is not sloth"**
Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and JohnWilkes
USENIX ATC '95, New Orleans, Louisiana
여러 가지 예를 들어서 유휴 시간을 시스템에서 잘 활용하는 방법을 설명한 재미있고 읽기 쉬운 논문이다.

[LL82] **"Virtual Memory Management in the VAX/VMS Operating System"**
Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
클러스터링이 사용된 것을 나타내는 가장 첫 문서는 아니지만 동작하는 방식을 쉽게 설명한 글이다.