25

물리 메모리 크기의 극복 : 정책

가상 메모리 관리자의 입장에서 비어 있는 메모리가 많을수록 일은 쉬워진다. 페이지 폴트가 발생하면 빈페이지 리스트에서 비어 있는 페이지를 찾아서 폴트를 일으킨 페이지에게 할당하면 된다. 오, 운영체제 씨 축하합니다! 또 다시 해냈군요.

불행하게도 빈 메모리 공간이 거의 없으면 일이 더 복잡해진다. 그런 경우 운영체제는 **메모리 압박**(**memory pressure**)을 해소하기 위해 다른 페이지들을 강제적으로 페이징 아웃(paging out)하여 활발히 사용 중인 페이지들을 위한 공간을 확보한다. **내보낼**(**evict**) 페이지(또는 페이지들) 선택은 운영체제의 **교체 정책**(**replacement policy**) 안에 집약되어 있다. 과거의 시스템들은 물리 메모리의 크기가 작았기 때문에 초기 가상 메모리 시스템의 가장 중요한 역할 중 하나가 바로 이 교체 정책이었다. 최소한, 좀 더 알아둘 만한 가치가 있는 정책이다. 우리가 살펴볼 핵심 문제는 다음과 같다.

핵심 질문 : 내보낼 페이지는 어떻게 결정하는가

운영체제는 어떻게 메모리에서 내보낼 페이지(또는 페이지들)를 결정할 수 있을까? 이 결정은 시스템의 교체 정책에 의해서 내려진다. 교체 정책은 보편 타당한 원칙들을 (다음에서 설명함) 따르지만 코너 케이스를 피하기 위한 수정 사항들도 포함되어 있다.

25.1 캐시 관리

정책에 대해서 논하기 전에 먼저 해결하고자 하는 문제에 대해서 좀 더 상세히 설명하도록 하겠다. 시스템의 전체 페이지들 중 일부분만이 메인 메모리에 유지된다는 것을 가정하면, 메인 메모리는 시스템의 가상 메모리 페이지를 가져다 놓기 위한 **캐시**로 생각될 수 있다. 이 캐시를 위한 교체 정책의 목표는 **캐시 미스**의 횟수를 최소화하는 것이다. 즉, 디스크로부터 페이지를 가져오는 횟수를 최소로 만드는 것이다. 한편으로 **캐시 히트** 횟수를 최대로 하는 것도 목표라고 할 수 있다. 즉, 접근된 페이지가 메모리에 이미 존재하는 횟수를 최대로 하는 것이다.

캐시 히트와 미스의 횟수를 안다면 프로그램의 **평균 메모리 접근 시간**(**average memory access time**, **AMAT**)을 계산할 수 있다(컴퓨터 아키텍트가 하드웨어

캐시의 성능을 측정할 때 사용하는 미터법이다 [HP06]). 히트와 미스의 정보를 안다면 프로그램의 AMAT는 다음과 같은 식으로 계산할 수가 있다.

$$AMAT = (P_{Hit} \cdot T_M) + (P_{Miss} \cdot T_D) \tag{25.1}$$

여기서 T_M은 메모리 접근 비용을 나타내고, T_D는 디스크 접근 비용, P_{Hit}는 캐시에서 데이터 항목을 찾을 확률을 나타내며(히트), P_{Miss}는 캐시에서 데이터를 못찾을 확률(미스)을 나타낸다. P_{Hit}와 P_{Miss}는 각각 0.0에서 1.0 사이의 값을 가지며 $P_{Miss} + P_{Hit} = 1.0$을 만족한다.

예를 들어, 어떤 컴퓨터가 4 KB의 작은 메모리를 가지고 있고, 페이지의 크기는 256 바이트라고 하자. 가상 주소는 4비트 VPN(최상위 비트)과 8비트 오프셋(최하위 비트)의 두 부분으로 구성된다. 그러므로 이 예제에서 한 프로세스가 2^4 또는 총 16개의 가상 페이지들에 접근할 수 있다. 프로세스는 가상 주소 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900의 메모리를 가리키고 있다. 이 가상 주소들은 주소 공간의 첫 열 개 페이지들의 첫 번째 바이트를 나타낸다(각 가상 주소들의 첫 16진수 숫자가 페이지 번호를 나타낸다).

또한, 가상 페이지 3을 제외한 모든 페이지가 이미 메모리에 있다고 가정하자. 주어진 순서의 메모리 참조는 다음과 같은 동작을 보일 것이다: 히트, 히트, 히트, 미스, 히트, 히트, 히트, 히트, 히트, 히트. **히트율(hit rate)**을(메모리에 참조 대상을 찾은 백분율) 계산하면 10개의 참조 중에서 9개가 메모리에 있었기 때문에 90%가 된다($P_{Hit} = 0.9$). **미스율(miss rate)**은 당연하게 10%($P_{Miss} = 0.1$)가 된다.

AMAT를 계산하기 위해서는 메모리를 접근하는 비용과 디스크를 접근하는 비용을 알아야 한다. 메모리를 접근하는 비용(T_M)이 약 100 nsec라고 하고 디스크를 접근하는 비용(T_D)이 약 10 μsec라고 가정할 때 AMAT은 $0.9 \cdot 100ns + 0.1 \cdot 10ms$ 이고 계산하면 $0.9ns + 1ms$ 또는 1.0009 msec 혹은 약 1 μsec가 된다. 만약 히트율이 99.9%가 되었었다면 결과는 확연히 달라진다. 즉, AMAT가 10.1 microseconds 혹은 대략 100배 더 빨라진다. 히트율이 100%에 가까워질수록 AMAT는 100 nsec에 가까워진다.

이 예제에서 볼 수 있듯이 현대 시스템에서는 디스크 접근 비용이 너무 크기 때문에 아주 작은 미스가 발생하더라도 전체적인 AMAT에 큰 영향을 주게 된다. 컴퓨터가 디스크 속도 수준으로 느리게 실행되는 것을 방지하기 위해서는 당연히 미스를 최대한 줄여야 한다. 이를 위해서는 좋은 정책을 잘 만드는 것이다. 이제 그 방법을 알아보자.

25.2 최적 교체 정책

교체 정책의 동작 방식을 잘 이해하기 위해서 **최적** 교체 정책(The Optimal Replacement Policy)과 비교하는 것이 좋다. 최적 정책(optimal replacement policy)은 수년 전 Belady에 의해 개발되었다 [Bel66](그는 원래 MIN이라고 불렀다). 최적 교체 정책은 미스를 최소화한다. Belady는 *가장 나중에 접근될* 페이지를 교체하는 것이 최적이며, 가장 적은 횟수의 미스를 발생시킨다는 것을 증명하였다. 이 정책은 간단하지만 구현하기는 어려운 정책이다.

팁 : 최적의 방법과 비교하는 것은 유용하다

최적의 방법은 비현실적이지만 시뮬레이션이나 다른 연구 목적에서 비교 대상으로 매우 유용하다. 새롭게 개발된 알고리즘이 80%의 히트율을 가진다고 주장하는 것은 의미가 없다. 주관적이기 때문이다. 그렇지만 최적 기법이 82%의 히트율을 가진다고 부가 설명하면(그러면서 새로운 방법이 최적의 결과에 매우 근접했다는 것을 보이면), 결과를 더 의미 있게 만들고, 받아들이는 사람이 맥락을 이해할 수 있게 된다. 어떤 연구를 수행하든 정답을 알고 있다면 가능한 성능 개선이 어느 정도인지 가늠할 수 있다. 또한 가장 이상적인 경우와 충분히 근접해지면 성능 개선 시도를 언제 중단할 수 있는지 알 수 있다 [Arp03].

최적 기법이 직관적으로 쉽게 이해되었기 바란다. 이렇게 생각해 보자. 만약 페이지 하나를 내보내야 한다면 지금부터 가장 먼 시점에 필요하게 될 페이지를 버리는 것이 좋지 않을까? 핵심은 가장 먼 시점에 필요한 페이지보다 캐시에 존재하는 다른 페이지들이 더 중요하다는 것이다. 이 주장이 참인 이유는 간단하다. 가장 먼 시점에 접근하게 될 페이지보다 다른 페이지들을 먼저 참조할 것이기 때문이다.

간단한 예제를 통해 최적 기법의 동작을 살펴보자. 프로그램이 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1의 순서대로 가상 페이지들을 접근한다고 하자. 캐시에 세 개의 페이지를 저장할 수 있다고 가정할 때, 그림 25.1은 최적의 기법의 동작을 보여준다.

접근	히트/미스?	내보냄	결과적인 캐시 상태
0	미스		0
1	미스		0, 1
2	미스		0, 1, 2
0	히트		0, 1, 2
1	히트		0, 1, 2
3	미스	2	0, 1, 3
0	히트		0, 1, 3
3	히트		0, 1, 3
1	히트		0, 1, 3
2	미스	3	0, 1, 2
1	히트		0, 1, 2

〈그림 25.1〉 **최적의 교체 정책의 흐름**

그림에서 다음과 같은 동작을 볼 수 있다. 캐시는 처음에 비어 있는 상태로 시작하기 때문에 첫 세 번의 접근은 당연히 미스이다. 이러한 종류의 미스는 때로는 **최초 시작 미스**(**cold-start miss**) 또는 **강제 미스**(**compulsory miss**)라고 한다. 그 다음에 페이지 0과 1을 참조하면 캐시에서 히트된다. 마지막으로 또 다른 미스를 만나게 된다 (페이지 3). 그렇지만 이때는 캐시가 가득 차있기 때문에 교체 정책이 실행되어야 한다!

여담: 캐시 미스의 종류

컴퓨터 구조의 분야에서는 미스를 종류에 따라 분류하는 것이 때때로 유용하다. 그 분류는 세 가지로 강제 미스, 용량 미스, 그리고 충돌 미스가 있으며 이것들은 **세 개의 C**라고 부르기도 한다 [Hil87]. **강제 미스**(**compulsory miss**, 또는 **최초 시작 미스**(**cold-start miss**) [EF78])는 캐시가 비어 있었고 그 항목을 처음으로 참조하기 때문에 발생한다. **용량 미스**(**capacity miss**)는 캐시의 공간이 다찼기 때문에 새로운 항목을 캐시에 넣기 위해서 어떤 항목을 내보내야 할 때 발생한다. 세 번째 종류의 미스는(충돌 미스(conflict miss))는 **세트 연관 매핑**(**set-associativity**)이라고 불리는 것 때문에 하드웨어에서 발생한다. 페이지 캐시는 **완전 연관 매핑**(**fully-associative**)으로 되어 있기 때문에 운영체제에서는 발생하지 않는다. 즉, 메모리에 페이지를 탑재하는 위치에 제한이 없다는 말이다. 상세한 내용은 H&P에 나와 있다 [HP06].

이때 다음과 같은 질문이 나온다. 어떤 페이지를 교체해야 할까? 최적 기법의 경우 캐시에 현재 탑재되어 있는 각 페이지들(0, 1, 그리고 2)의 미래를 살펴본다. 그러면 0은 거의 즉시 접근이 될 것이며 1은 약간 후에, 그리고 2는 가장 먼 미래에 접근이 될 것이란 것을 알 수 있다. 그러므로 최적 기법은 쉬운 선택을 하면 된다. 페이지 2를 내보내고 결과적으로 캐시에는 페이지 0, 1, 3이 남게 된다. 그 후의 세 번의 참조는 히트된다. 그 다음으로 오래 전에 내보냈던 페이지 2를 만나서 또 한 번의 미스를 경험하게 된다. 여기서 최적 기법은 다시 한 번 캐시 내의 각 페이지들(0, 1, 그리고 3)의 미래를 검사한다. 그리고 페이지 1만 아니면(바로 다음에 접근될 예정이라) 어느 페이지를 내보내도 괜찮다는 것을 알게 된다. 페이지 0을 선택하는 것도 괜찮은 결정이었을 수도 있지만, 예제에서는 페이지 3을 내보냈다. 최종적으로 페이지 1이 히트되고 종료된다.

캐시의 히트율을 계산할 수 있다. 캐시 히트가 6번 미스가 5번이었으므로 히트율은 $\frac{Hits}{Hits+Misses}$으로 계산되며 $\frac{6}{6+5}$ 또는 54.5%가 된다. 강제 미스들(즉, 해당 페이지가 처음 미스된 경우)을 제외한 히트율을 계산한다면 85.7%의 히트율을 얻는다.

불행하게도 스케줄링 정책을 만들 때도 보았지만, 미래는 일반적으로 미리 알 수 없다. 그렇기 때문에 범용 운영체제[1]에서는 최적 기법의 구현은 불가능하다. 내보낼 페이지를 결정하기 위한 실제적이고 배포가 가능한 정책을 만들기 위해 다른 방법을 찾는 것에 집중할 것이다. 최적 기법은 비교 기준으로만 사용될 것이며, 이를 통해 "정답"에 얼마나 가까운지 알 수 있다.

25.3 간단한 정책: FIFO

초기의 많은 시스템들은 최적의 방법에 도달하려는 복잡한 시도를 피하고 대신 매우 간단한 교체 정책을 채용하였다. 예를 들면, 일부 시스템에서는 **FIFO**(먼저 들어온 것이 먼저 나간다, 선입선출) 교체 방식을 사용하였다. 이 방법은 시스템에 페이지가 들어오면

1) 만약 할 수 있다면, 우리에게 꼭 알려주기 바란다! 우리는 다같이 부자가 될 수 있다. 또는, 상온 핵융합의 방법을 "발견했다"던 과학자들처럼 엄청난 멸시와 조롱을 받게 될 것이다 [FP89].

접근	히트/미스?	내보냄	결과적인 캐시 상태
0	미스		선입 0
1	미스		선입 0, 1
2	미스		선입 0, 1, 2
0	히트		선입 0, 1, 2
1	히트		선입 0, 1, 2
3	미스	0	선입 1, 2, 3
0	미스	1	선입 2, 3, 0
3	히트		선입 2, 3, 0
1	미스	2	선입 3, 0, 1
2	미스	3	선입 0, 1, 2
1	히트		선입 0, 1, 2

〈그림 25.2〉 **FIFO 정책의 흐름**

큐에 삽입되고, 교체를 해야 할 경우 큐의 테일에 있는 페이지가("먼저 들어온" 페이지) 내보내진다. FIFO는 매우 구현하기 쉽다는 장점을 가진다.

예제로 사용한 메모리 참조의 흐름에서 FIFO가 어떻게 동작하는지 살펴보자(그림 25.2). 처음에는 마찬가지로 페이지 0, 1과 2에 대한 세 개의 강제 미스로 시작하고, 0과 1이 다음에 히트된다. 다음으로 페이지 3이 참조되지만 미스를 일으킨다. FIFO를 사용할 때 교체 결정은 쉽다. 순서상 "첫 번째"로 들어온 페이지인 0을 선택하면 된다(그림에서 캐시 상태는 FIFO 순서로 정렬되었다. 처음 들어온 페이지가 좌측에 있다). 불행하게도 다음 접근은 페이지 0이라서 또 다른 미스를 만들어내고 교체가(페이지 1) 필요하다. 그 후에는 페이지 3에서 히트가 되지만 1과 2는 미스가 되며, 끝으로 3은 히트된다.

최적의 경우와 비교하면 FIFO는 눈에 띄게 성능이 안 좋다. 히트율은 36.4%가 된다(또는 강제 미스를 제외하면 57.1%가 된다). FIFO는 블럭들의 중요도를 판단할 수가 없다. 페이지 0이 여러 차례 접근이 되었더라도 단순히 메모리에 먼저 들어왔다는 이유로 FIFO는 페이지 0을 내보낸다.

25.4 또 다른 간단한 정책: 무작위 선택

또 다른 유사한 교체 정책은 무작위 방식이다. 이 방식은 메모리 압박이 있을 때 페이지를 무작위로 선택하여 교체한다. 무작위 선택 방식은 FIFO와 유사한 성질을 가지고 있다. 구현하기 쉽지만 내보낼 블럭을 제대로 선택하려는 노력을 하지 않는다. 앞선 예제에서 사용한 메모리 참조를 사용하여 무작위 선택 정책이 어떻게 동작하는지 살펴보자(그림 25.3을 보자).

물론 무작위 선택 정책은 선택할 때 얼마나 운이 좋은지(또는 운이 나쁜지)에 전적으로 의존한다. 위에서 사용한 참조 열에서는 무작위 선택 방식이 FIFO 보다는 약간 더 좋은 성능을 보이며 최적의 방법보다는 약간 나쁜 성능을 보인다. 무작위 선택 방식의

여담: Belady's Anomaly

Belady(최적 기법의)와 그의 동료들은 예상과는 약간 다르게 동작하는 흥미로운 경우를 발견하였다 [BNS69]. 그 메모리 참조 흐름은 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5이다. 그들이 연구 중이던 교체 정책은 FIFO였다. 흥미로운 부분: 캐시의 크기를 3에서 4개의 페이지로 증가시킬 때 캐시 히트율은 어떻게 변할까.

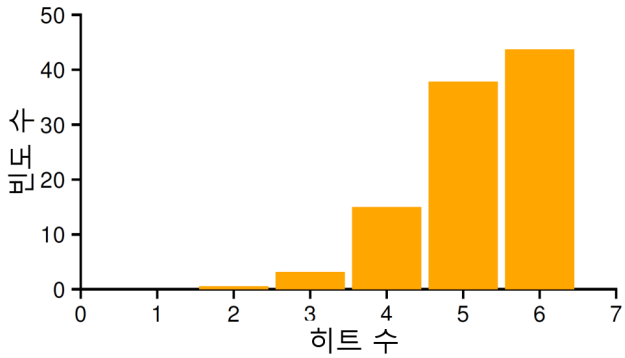
일반적으로 캐시의 크기가 커지면 캐시 히트율이 증가하는(더 좋아지는) 것을 기대할 것이다. 그런데 FIFO를 사용하는 경우에는 더 안 좋아진다! 히트와 미스를 직접 계산해서 확인해 보라. 이 이상한 현상은 일반적으로 **Belady's anomaly**라고 부른다(그의 동료들은 유감이겠지만).

LRU와 같은 다른 정책들은 이와 같은 문제를 겪지 않는다. 왜 그런지 추측할 수 있겠는가? 사실 LRU는 **스택 특성**(**stack property**)이라고 알려진 특성을 가진다 [Mat+70]. 이 특성을 가지는 알고리즘들의 경우에는 캐시 크기가 $N+1$로 증가하게 되면 자연적으로 캐시 크기 N일 때의 내용을 포함하게 되어 있다. 그러므로 캐시의 크기가 증가하면 히트율은 그대로 유지되거나 향상된다. 여러 정책들 중 FIFO와 랜덤 정책은 이 스택 특성을 따르지 않고 따라서 변칙적인 행동을 보이기 쉽다.

접근	히트/미스?	내보냄	결과적인 캐시 상태
0	미스		0
1	미스		0, 1
2	미스		0, 1, 2
0	히트		0, 1, 2
1	히트		0, 1, 2
3	미스	0	1, 2, 3
0	미스	1	2, 3, 0
3	히트		2, 3, 0
1	미스	3	2, 0, 1
2	히트		2, 0, 1
1	히트		2, 0, 1

〈그림 25.3〉 **무작위 선택 정책의 흐름**

일반적인 성능을 알아보기 위해 실험을 수천 번 반복해 볼 수 있다. 그림 25.4는 서로 다른 랜덤 시드를 가지는 실험을 만 번 수행했을 때 무작위 선택 방식이 몇 번의 히트를 얻었는지를 나타낸다. 보는 바와 같이 어떤 때는(40%가 살짝 넘는 경우들) 무작위 선택 방식이 최적 기법만큼 좋은 성능을 보이며 예제에서 6번의 히트를 얻었다. 때로는 매우 안 좋은 성능을 보이며, 2번의 히트 또는 그 이하를 얻었다. 무작위 선택 방식의 동작은 그때그때에 따라 달라진다.



〈그림 25.4〉 **10,000번 실행할 때의 무작위 선택 정책의 성능**

25.5 과거 정보의 사용: LRU

불행하게도 FIFO 또는 무작위 선택 방식처럼 단순한 정책들은 중요한 페이지들을 혹은 바로 다시 참조하게 될 것들을 내보낼 수 있다는 비슷한 문제를 겪게 된다. FIFO는 먼저 가져온 페이지를 먼저 내보낸다. 그 페이지에 중요한 코드 또는 자료 구조가 들어 있어다. 곧 다시 필요하다고 해도, 먼저 들어온 페이지라면 내보낸다. FIFO와 무작위 선택 방식 그리고 그와 유사한 정책들은 최적 기법의 성능을 따라갈 수가 없기 때문에 좀 더 정교한 방식이 필요하다.

스케줄링 정책에서와 같이 미래에 대한 예측을 위해서 *과거 사용 이력을* 활용해 보자. 예를 들어 어떤 프로그램이 가까운 과거에 한 페이지를 접근했다면 가까운 미래에 그 페이지를 다시 접근하게 될 확률이 높다.

페이지 교체 정책이 활용할 수 있는 과거 정보 중 하나는 **빈도수(frequency)**이다. 만약 한 페이지가 여러 차례 접근되었다면, 분명히 어떤 가치가 있기 때문에 교체되면 안될 것이다. 좀 더 자주 사용되는 페이지의 특징은 접근의 **최근성(recency)**이다. 더 최근에 접근된 페이지일수록 가까운 미래에 접근될 확률이 높을 것이다.

이러한 류의 정책은 **지역성의 원칙(principle of locality)**이라고 부르는 특성에 기반을 둔다 [Den70]. 이 원칙은 프로그램의 행동 양식을 관찰하여 얻은 결과이다. 이 원칙이 말하는 것은 단순하다. 프로그램들은 특정 코드들과 자료 구조를 상당히 빈번하게 접근하는 경향이 있다는 것이다. 코드의 예로는 반복문 코드를 들 수 있으며, 자료 구조로는 그 반복문에 의해 접근되는 배열을 예로 들 수 있다. 과거의 현상을 보고 어떤 페이지들이 중요한지 판단하고, 내보낼 페이지를 선택할 때 중요한 페이지들은 메모리에 보존하는 것이다.

그리하여 과거 이력에 기반한 교체 알고리즘 부류가 탄생하게 되었다. **Least-Frequently-Used(LFU)** 정책은 가장 적은 빈도로 사용된 페이지를 교체한다. **Least-Recently-Used(LRU)** 정책은 가장 오래 전에 사용하였던 페이지를 교체한다. 이러한 알고리즘들은 기억하기 쉽다. 이름을 알면 정확하게 그 알고리즘이 어떻게 동작하는지

알 수 있다. 이름이 가지는 탁월한 성질이 아닐 수 없다.

LRU 방식을 좀 더 이해하기 위해서 앞서 사용한 참조 패턴에서 예제에서 어떻게 동작하는지 알아보자. 그림 25.5에 그 결과가 있다. 그림을 보면 상태를 유지하지 않는 랜덤이나 FIFO 정책에 비해 LRU가 과거 정보를 사용하여 더 좋은 성능을 보임을 알 수 있다. 이 예제에서 LRU는 처음으로 페이지를 교체해야 할 때가 되었을 때 페이지 2를 내보낸다. 왜냐하면 0과 1이 최근에 사용되었기 때문이다. 그 후에는 페이지 0을 교체한다. 1과 3이 더 최근에 접근되었기 때문이다. 두 경우 모두 LRU는 과거 정보를 활용하였고, 결과적으로 이후의 참조들에서 히트가 발생하였다는 것을 보면 결정이 옳았다는 것을 알 수 있다. 이 예제에서 LRU가 최적 기법과 같은 정도 수준의 성능을 얻을 수 있는 최고의 결과를 보여주고 있다[2].

접근	히트/미스?	내보냄	결과적인 캐시 상태	
0	미스		LRU	0
1	미스		LRU	0, 1
2	미스		LRU	0, 1, 2
0	히트		LRU	1, 2, 0
1	히트		LRU	2, 0, 1
3	미스	2	LRU	0, 1, 3
0	히트		LRU	1, 3, 0
3	히트		LRU	1, 0, 3
1	히트		LRU	0, 3, 1
2	미스	0	LRU	3, 1, 2
1	히트		LRU	3, 2, 1

〈그림 25.5〉 **LRU 정책의 흐름**

이와 반대되는 **Most-Frequently-Used**(**MFU**)와 **Most-Recently-Used**(**MRU**)와 같은 알고리즘도 존재하지만 대부분의 경우(모든 경우는 아님!), 이러한 정책들은 잘 동작하지 않는다. 대부분의 프로그램들에서 관찰되는 지역성 접근 특성을 사용하지 않고 오히려 무시하기 때문이다.

25.6 워크로드에 따른 성능 비교

지금까지 살펴본 정책들 동작을 더 잘 이해하기 위해 몇 가지 예제들을 더 살펴보자. 이번에는 짧은 흐름 대신 좀 더 복잡한 워크로드를 살펴볼 것이다. 하지만 이러한 워크로드조차도 상당히 간단화된 것이기 때문에 제대로된 연구를 위해서는 응용 프로그램의 트레이스 자료를 사용해야 할 것이다.

첫 번째 살펴볼 워크로드에서는 지역성이 없다. 그 말은 접근되는 페이지들의 집합에서 페이지가 무작위적으로 참조된다는 것을 의미한다. 이 예제에서는 100개의

2) 결과가 조작된 것이기는 하다. 하지만 주장을 증명하기 위해서는 약간의 조작도 필요하다.

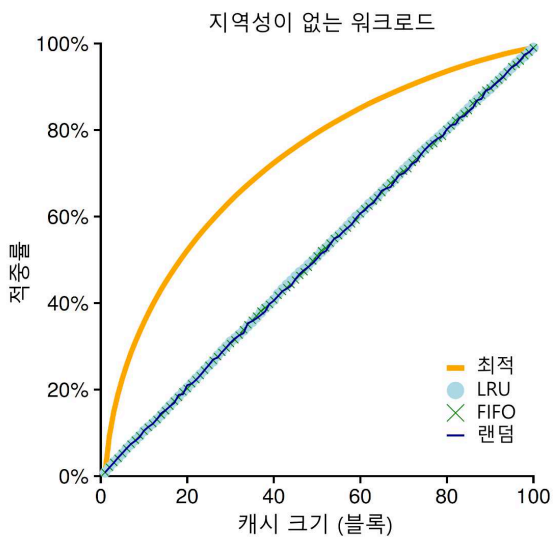
여담: 지역성의 종류

프로그램들이 보이는 지역성에는 두 가지 종류가 있다. 첫 번째는 **공간 지역성(spatial locality)**이다. 이것은 어떤 페이지 P가 접근되었다면 그 페이지 주변의 페이지들($P-1$ 또는 $P+1$이라 하자)이 참조되는 경향이 있다는 것을 말한다. 두 번째는 **시간 지역성(temporal locality)**으로 가까운 과거에 참조되었던 페이지는 가까운 미래에 다시 접근되는 경향이 있다는 것을 나타낸다. 하드웨어 캐시는 명령어와 데이터 그리고 주소 변환 등의 여러 계층에 구현되어 있으며, 프로그램은 이러한 지역성의 도움을 받아 더 빠르게 실행된다. 하드웨어 캐싱에서 지역성의 존재에 대한 가정은 매우 중요한 역할을 한다.

물론 **지역성의 원칙(principle of locality)**은 원칙이라는 이름을 가지고 있지만 모든 프로그램들이 따라야 하는 전혀 변경할 수 없는 엄격한 법칙은 아니다. 실제로, 어떤 프로그램들은 메모리(또는 디스크)를 무작위적으로 접근하며, 지역성이 명확하지 않거나 어떤 지역성도 보이지 않는다. 어떤 종류의 캐시든 설계할 때 지역성을 고려하는 것이 좋지만, 성공을 보장해주는 것은 아니다. 컴퓨터 시스템의 설계에 있어 어느 정도의 유용함이 입증된 결과일 뿐이다.

페이지들이 일정 시간 동안 계속 접근하는 워크로드를 사용한다. 접근되는 페이지는 무작위적으로 선택되며 페이지들이 총 10,000번 접근된다. 실험에서는 캐시의 크기를 매우 작은 것부터(한 페이지) 모든 페이지들을 담을 수 있을 정도의 크기까지(100페이지) 증가시켰으며, 이를 통해 각 정책이 캐시 크기에 따라 어떻게 동작하는지 살펴보았다.

그림 25.6에서는 최적의 방법과 LRU, 랜덤 그리고 FIFO 방식을 사용하였을 때의 실험 결과를 나타낸다. 그림의 y 축은 각 정책이 달성한 히트율을 나타내며 x 축은 앞서

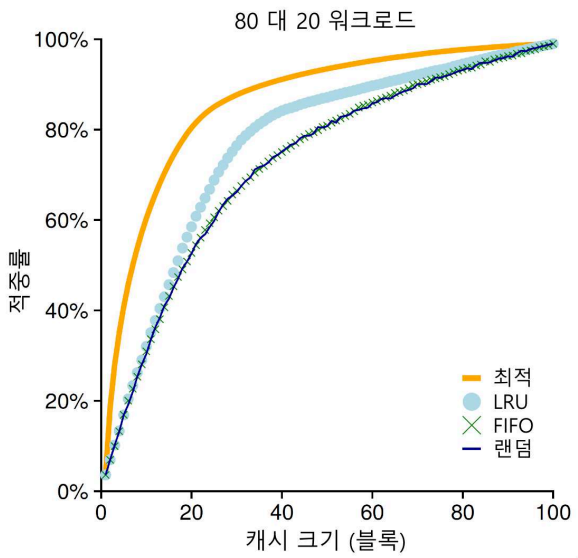


〈그림 25.6〉 **지역성이 없는 워크로드**

설명한 캐시 크기의 변화를 나타낸다.

그림에서 몇 가지 결론을 얻을 수 있다. 먼저, 워크로드에 지역성이 없다면 어느 정책을 사용하든 상관이 없다. LRU와 FIFO 그리고 무작위 선택 정책은 모두 동일한 성능을 보이며, 히트율은 정확히 캐시의 크기에 의해서 결정된다. 두 번째, 캐시가 충분히 커서 모든 워크로드를 다 포함할 수 있다면 역시 어느 정책을 사용하든지 상관 없다. 참조되는 모든 블럭들이 캐시에 들어갈 수 있으면 모든 정책들은(무작위 선택마저도) 히트율이 100%에 도달한다. 마지막으로, 최적 기법이 구현 가능한 기타 정책들보다 눈에 띄게 더 좋은 성능을 보인다. 미래를 알 수 있다면 교체 작업을 월등히 잘할 수 있다.

다음으로 살펴볼 워크로드는 "80 대 20" 워크로드라고 부른다. 20%의 페이지들에서("인기 있는" 페이지) 80%의 참조가 발생하고 나머지 80%의 페이지들에 대해서 20%의 참조만("비인기" 페이지) 발생한다. 이 워크로드에서는 마찬가지로 총 100개의 페이지들이 있다. "인기 있는" 페이지들에 대한 참조가 실험 시간의 대부분을 차지한다. 그림 25.7에는 각 정책들이 이 워크로드에서 어떻게 동작하는지 보여주고 있다.



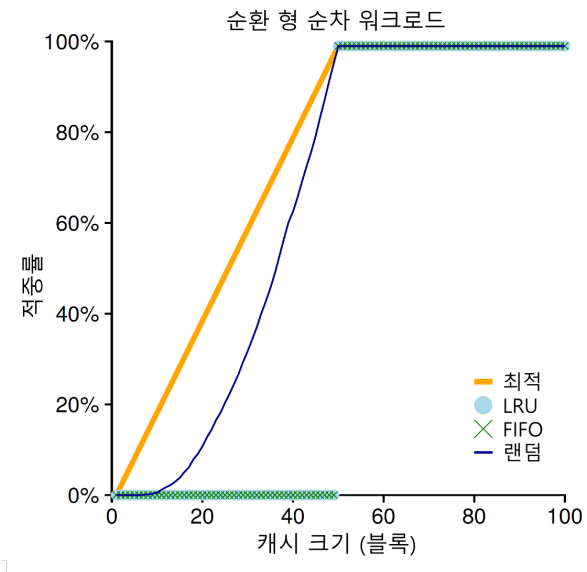
〈그림 25.7〉 **80 대 20 워크로드**

그림에서 볼 수 있듯이 랜덤과 FIFO 정책이 상당히 좋은 성능을 보이지만, 인기 있는 페이지들을 캐시에 더 오래두는 경향이 있는 LRU가 더 좋은 성능을 보인다. 인기 있는 페이지들이 과거에 빈번하게 참조되었기 때문에 그 페이지들은 가까운 미래에 다시 참조되는 경향이 있기 때문이다. 최적 기법은 여전히 더 좋은 성능을 보이고 있으며, 이는 LRU의 과거 정보가 완벽하지는 않다는 것을 보여준다.

이런 의문이 들 수도 있다. 무작위 선택 방식과 FIFO 방식을 개선하여 만든 LRU가 그렇게 대단한 것인가? 그 대답은 항상 그렇듯이 "상황에 따라 다르다" 이다. 만약 각

미스로 인해서 매우 비싼 값을 치러야 한다면(드물게 일어나는 일이 아니다) 히트율이 약간만 증가하더라도(미스율이 줄어들면) 성능에 큰 차이를 만들어 낼 수 있다. 만약 미스로 인한 영향이 그렇게 크지 않다면, LRU로 얻을 수 있는 장점은 당연히 그렇게 중요하지 않게 된다.

마지막으로 하나의 워크로드를 더 살펴보자. 이 워크로드를 "순차 반복" 워크로드라고 부른다. 이 워크로드는 50개의 페이지들을 순차적으로 참조한다. 0번 페이지를 참조하고 1번을 참조하고 ..., 49번째의 페이지를 참조한 후에 다시 처음으로 돌아가서 그 접근 순서를 반복한다. 50개의 개별적인 페이지들을 총 10,000번 접근한다. 그림 25.8에서 보이는 마지막 도표는 각 정책들이 이 워크로드에서 어떤 성능을 보이는지 나타낸다.



〈그림 25.8〉 **순환 형 워크로드**

여러 응용 프로그램에서 흔하게 볼 수 있는 이 워크로드는(데이터베이스와 같은 상업용 응용 프로그램들을 포함하여 [CD85]) LRU와 FIFO 정책에서 가장 안좋은 성능을 보인다. 순차 반복 워크로드에서 이 알고리즘들은 오래된 페이지들을 내보낸다. 불행하게도 워크로드의 반복적인 특성으로 인해서 오래된 페이지들은 정책들이 캐시에 유지하려고 선택한 페이지들보다 먼저 접근된다. 실제로, 캐시의 크기가 49라고 할지라도 50개의 페이지들을 순차 반복하는 워크로드에서는 캐시 히트율이 0%가 된다. 흥미롭게도 무작위 선택 정책은 최적의 경우에 못 미치기는 하지만 눈에 띄게 좋은 성능을 보인다. 무작위 선택 정책의 히트율은 최소한 0%는 아니다. 이렇게 무작위 선택 정책은 몇 가지 좋은 특성을 가지고 있다는 것을 알 수 있다. 그 중 한 가지는 이상한 코너 케이스가 발생하지 않는다는 것이다.

25.7 과거 이력 기반 알고리즘의 구현

살펴본 것처럼 중요한 페이지들을 내보낼 수도 있는 단순한 FIFO와 무작위 선택 정책들보다 LRU와 같은 알고리즘이 더 좋은 성능을 보인다. 하지만 과거 정보에 기반을 둔 정책은 새로운 문제점이 있다. 이런 정책을 어떻게 구현할 것인가?

LRU를 예로 들어 보자. 이를 완벽하게 구현하기 위해서는 많은 작업을 해야 한다. 특히, 각 *페이지 접근마다*(즉, 명령어 탑재나 로드 또는 저장하려는 각 메모리 접근) 해당 페이지가 리스트에 가장 앞으로 이동하도록(즉, MRU 측으로) 자료 구조를 갱신해야 한다. FIFO 방식과 비교해 보자. FIFO 리스트는 어떤 페이지의 *제거*(가장 먼저 들어온 페이지를 제거하기) 또는 새로운 페이지의 삽입 시에만(가장 나중에 들어온 쪽으로) 접근된다. LRU에서는 어떤 페이지가 가장 최근에 또는 가장 오래 전에 사용되었는지를 관리하기 위해서 *모든 메모리 참조 정보를* 기록해야 한다. 세심한 주의 없이 정보를 기록하면 성능이 크게 떨어질 수 있다.

이 작업을 좀 더 효율적으로 하는 방법은 약간의 하드웨어 지원을 받는 것이다. 예를 들어 페이지 접근이 있을 때마다 하드웨어가 메모리의 시간 필드를 갱신하도록 할 수 있다(예를 들어, 이 정보를 각 프로세스의 페이지 테이블에 포함시키거나, 물리 페이지마다 한 항목씩 할당된 배열로 보관할 수도 있다). 이렇게 하여 페이지가 접근되면 하드웨어가 시간 필드를 현재 시간으로 설정한다. 페이지 교체 시에 운영체제는 가장 오래 전에 사용된 페이지를 찾기 위해 시간 필드를 검사한다.

시스템의 페이지 수가 증가하면 페이지들의 시간 정보 배열을 검색하여 가장 오래 전에 사용된 페이지를 찾는 것은 매우 고비용의 연산이 된다. 4GB의 메모리를 가지는 현대의 기기에서 4KB 페이지들로 나눈 것을 상상해 보라. 그런 기계는 백만 개의 페이지들을 가지고 있다. LRU 정책의 교체 대상 페이지를 찾는 데 현대의 CPU 속도라 할지라도 오랜 시간이 걸리게 된다. 다음과 같은 질문이 떠오른다. 가장 오래된 페이지를 꼭 찾아야만 할까? 대신 비슷하게 오래된 페이지를 찾아도 되지 않을까?

핵심 질문: LRU 교체 정책을 구현하는 방법
완벽한 LRU를 구현하는 것은 비싼 비용이 든다는 것을 전제로, 어떻게 하면 LRU에 가까운 선택을 하고 유사한 행동을 보이는 정책을 구현할 수 있을까?

25.8 LRU 정책 근사하기

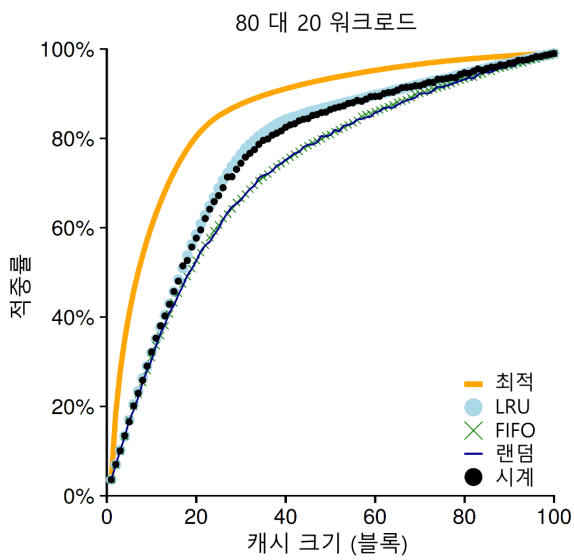
LRU 정책에 가까운 구현이 가능한가라는 질문에 대한 답은 "예"이다. 연산량이라는 관점에서 볼 때, LRU를 "근사"하는 식으로 만들면 구현이 훨씬 쉬워진다. 실제로 현대의 많은 시스템이 이런 방식을 택하고 있다. 이 개념에는 **use bit**(때로는 **reference bit**라고도 불린다)라고 하는 약간의 하드웨어 지원이 필요하다. 이 기법은 페이징을 최초로 적용한 Atlas one-level store 시스템에 처음으로 구현되었다 [Kil+62]. 시스템의 각 페이지마다 하나의 use bit가 있으며, 이 use bit는 메모리 어딘가에 존재한다(구현에

따라 프로세스마다 가지고 있는 페이지 테이블에 있을 수도 있고 또는 어딘가에 배열의 형태로 존재할 수도 있다). 페이지가 참조될 때마다(즉, 읽히거나 기록되면) 하드웨어에 의해서 use bit가 1로 설정된다. 하드웨어는 이 비트를 절대로 지우지 않는다(즉, 0으로 설정하지 않는다). 0으로 바꾸는 것은 운영체제의 몫이다.

운영체제는 LRU에 가깝게 구현하기 위해서 use bit를 어떻게 활용할까? 여러 가지 방법이 있을 수 있지만, 간단한 활용법이 **시계 알고리즘**(**clock algorithm**) [Cor69]에서 제시되었다. 시스템의 모든 페이지들이 환형 리스트를 구성한다고 가정하자. **시계 바늘**(**clock hand**)이 특정 페이지를 가리킨다고 해 보자(어떤 것이든 상관이 없다). 페이지를 교체해야 할 때 운영체제는 현재 바늘이 가리키고 있는 페이지 P의 use bit가 1인지 0인지 검사한다. 만약 1이라면 페이지 P는 최근에 사용되었으며 바람직한 교체 대상이 *아니라*는 것을 뜻한다. P의 use bit는 0으로 설정되고(지워짐) 시계 바늘은 다음 페이지 $P+1$로 이동한다. 알고리즘은 use bit가 0으로 설정되어 있는, 즉 최근에 사용된 적이 없는, 페이지 찾을 때까지 반복된다(최악의 경우에는 모든 페이지들이 사용된 적이 있어서 모든 페이지들을 모두 탐색하면서 use bit를 전부 0으로 설정해야 할 수도 있다).

이 방법 외에도 use bit를 사용하여 LRU를 근사하는 다른 방법들이 존재한다. 주기적으로 use bit를 지우고, 교체 대상 페이지 선택을 위해 use bit가 1과 0인 페이지를 구분할 수 있으면 된다. Corbato의 알고리즘은 미사용 페이지를 찾기 위해 매번 모든 메모리를 검사하지 않아도 되는 좋은 특성을 가진 하나의 초기 방법일 뿐이다.

변형된 시계 알고리즘의 동작이 그림 25.9에 나와 있다. 이 변형 방식은 교체할 때 페이지들을 랜덤하게 검사한다. reference bit가 1로 설정되어 있는 페이지를 만나게 되면 비트를 지운다(즉, 0으로 설정한다). reference bit가 0으로 설정되어 있는 페이지를 만나면 그 페이지를 교체 대상으로 선정한다. 완벽한 LRU 만큼 좋은 성능을 보이지는



〈그림 25.9〉 **시계 기법을 사용한 80 대 20 워크로드**

않지만 과거 정보를 고려하지 않는 다른 기법들에 비해서는 성능이 더 좋다.

25.9 갱신된 페이지(Dirty Page)의 고려

많은 곳에서 실제 사용되고 있는 시계 알고리즘에 대한 추가 개선 사항을 논의하고자 한다. 이 사항은 Corbato [Cor69]가 처음 제안하였으며, 운영체제가 교체 대상을 선택할 때 메모리에 탑재된 이후에 변경되었는지를 추가적으로 고려하는 것이다. 이것이 필요한 이유는 다음과 같다.

만약에 어떤 페이지가 **변경**(**modified**)되어 **더티**(**dirty**) 상태가 되었다면, 그 페이지를 내보내기 위해서는 디스크에 변경 내용을 기록해야 하기 때문에 비싼 비용을 지불해야 한다. 만약 변경되지 않았다면(그러므로 깨끗한 상태(clean)), 내보낼 때 추가 비용은 없다. 해당 페이지 프레임은 추가적인 I/O 없이 다른 용도로 재사용될 수 있다. 때문에 어떤 VM 시스템들은 더티 페이지 대신 깨끗한 페이지를 내보내는 것을 선호한다.

이와 같은 동작을 지원하기 위해서 하드웨어는 **modified bit**(**더티 비트**라고도 불림)를 포함해야 한다. 페이지가 변경될 때마다 이 비트가 1로 설정되므로 페이지 교체 알고리즘에서 이를 고려하여 교체 대상을 선택한다. 예를 들어, 시계 알고리즘은 교체 대상을 선택할 때 사용되지 않은 상태이고 깨끗한, 두 조건을 모두 만족하는 페이지를 먼저 찾도록 수정된다. 이러한 조건을 만족시키는 페이지를 찾는 데 실패하면, 수정되었지만 한동안 사용되지 않았던 페이지를 찾는다.

25.10 다른 VM 정책들

페이지 교체 정책만이 VM 시스템이 채택하는 유일한 정책은 아니다(그 중에 가장 중요한 것이기는 하다). 예를 들어, 운영체제는 언제 페이지를 메모리로 불러들일지 결정해야 한다. 이 정책은 운영체제에게 몇 가지 다른 옵션을 제공한다. 이 정책은 Denning이 불렀던 것처럼 때로는 **페이지 선택**(**page selection**) 정책이라고 불린다 [Den70]).

운영체제는 대부분의 페이지를 읽어 들일 때 **요구 페이징**(**demand paging**) 정책을 사용한다. 이 정책은 말 그대로 "요청된 후 즉시", 즉 페이지가 실제로 접근될 때 운영체제가 해당 페이지를 메모리로 읽어 들인다. 운영체제는 어떤 페이지가 곧 사용될 것이라는 것을 대략 예상할 수 있기 때문에 미리 메모리로 읽어 들일 수도 있다. 이와 같은 동작을 **선반입**(**prefetching**)이라고 하며 성공할 확률이 충분히 높을 때에만 해야 한다. 예를 들어, 어떤 시스템은 코드 페이지 P가 메모리에 탑재되면 $P+1$이 접근될 확률이 높기 때문에 P가 탑재될 때 $P+1$도 함께 탑재되어야 한다고 가정할 수 있을 것이다.

또 다른 정책은 운영체제가 변경된 페이지를 디스크에 반영하는 데 관련된 방식이다. 한 번에 한 페이지씩 디스크에 쓸 수 있지만, 많은 시스템은 기록해야 할 페이지들을 메모리에 모은 후, 한 번에 (더 효율적으로) 디스크에 기록한다. 이와 같은 동작을 **클러스터링**(**clustering**) 또는 단순하게 쓰기 **모으기**(**grouping** of writes)라고 부르며 효과적인 동작 방식이다. 왜냐하면 디스크 드라이브는 여러 개의 작은 크기의 쓰기

요청을 처리하는 것보다 하나의 큰 쓰기 요청을 더 효율적으로 처리할 수 있는 특성을 갖고 있기 때문이다.

25.11 쓰래싱(Thrashing)

마치기 전에 마지막으로 한 가지 질문에 대해 다루고자 한다. 메모리 사용 요구가 감당할 수 없을 만큼 많고 실행 중인 프로세스가 요구하는 메모리가 가용 물리 메모리 크기를 초과하는 경우에 운영체제는 어떻게 해야 하는가? 이런 경우 시스템은 끊임없이 페이징을 할 수밖에 없고, 이와 같은 상황을 **쓰래싱**(**thrashing**)이라고 부른다 [Den70].

몇몇 초기 운영체제들은 쓰래싱이 발생했을 때, 이의 발견과 해결을 위한 상당히 정교한 기법들을 가지고 있었다. 예를 들면, 다수의 프로세스가 존재할 때, 일부 프로세스의 실행을 중지시킨다. 실행되는 프로세스의 수를 줄여서 나머지 프로세스를 모두 메모리에 탑재하여 실행하기 위해서이다. **워킹 셋**(**working set**)이란 프로세스가 실행 중에 일정 시간 동안 사용하는 페이지들의 집합이다. 일반적으로 **진입 제어**(**admission control**)라고 알려져 있는 이 방법은 많은 일들을 엉성하게 하는 것보다는 더 적은 일을 제대로 하는 것이 나을 때가 있다고 말한다. 현대 컴퓨터 시스템에서 뿐만 아니라 실제 삶에서도 종종 부딪치는 상황이기도 하다(아, 슬프다).

일부 최신 시스템들은 메모리 과부하에 대하여 좀 더 과감한 조치를 취하기도 한다. 예를 들면, 일부 버전의 Linux는 메모리 요구가 초과되면 **메모리 부족 킬러**(**out-of-memory killer**)를 실행시킨다. 이 데몬은 많은 메모리를 요구하는 프로세스를 골라 죽이는, 그다지 교묘하지 않은 방식으로 메모리 요구를 줄인다. 메모리 요구량을 줄이는 데는 성공적일지 몰라도, 이 방법은 문제가 될 수 있다. 예를 들어, 만약 X 서버를 죽이게 되면 화면이 필요한 모든 응용 프로그램들을 쓸모없게 만든다.

25.12 요약

모든 현대 운영체제 VM 시스템의 구성 요소인 페이지 교체 정책들과 기타 다른 정책들에 대해 다루었다. 현대 시스템들은 시계 알고리즘과 같은 순수한 LRU 근사 방법에 몇 가지 성능 향상을 위한 기능을 추가하였다. 예를 들어, **탐색 내성**(**scan resistance**)은 ARC [MM03]와 같은 다수의 현대 알고리즘의 중요한 요소이다. 탐색 내성이 있는 알고리즘은 대개는 LRU와 유사하게 동작하지만 LRU가 최악의 경우에 보이는 행동을 방지하였다. 최악의 현상은 순차 반복 워크로드 예에서 보았다. 페이지 교체 알고리즘의 진화는 계속된다.

지금까지 언급했던 알고리즘의 중요성은 점차 퇴색되고 있다. 메모리 접근 시간과 디스크 접근 시간의 차이가 점차 증가하고 있기 때문이다. 디스크에 페이징 하는 비용이 너무 비싸기 때문에 빈번한 페이징을 감당할 수 없다. 과도한 페이징에 대한 최적의 해결책은 아주 간단하다. 그냥 더 많은 메모리를 구입해라.

참고 문헌

[Arp03] **"Run-Time Adaptation in River"**
Remzi H. Arpaci-Dusseau
ACM TOCS, 21:1, February 2003
저자 중 한 명이 River라 불리던 시스템에서 했던 학위논문 연구의 정리이다. 시스템 설계자에게는 이상적인 경우와 비교하는 것이 중요한 기법이라는 것을 확실하게 배웠던 연구.

[BNS69] **"An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine"**
L.A. Belady, R.A. Nelson, and G.S. Shedler
Communications of the ACM, 12:6, June 1969
Belady's anomaly라고 알려진 짧은 메모리 참조 순서에 대한 소개. Nelson과 Shedler는 이 이름에 대해서 어떻게 생각할지 궁금하다.

[Bel66] **"A Study of Replacement Algorithms for Virtual-Storage Computer"**
Laszlo A. Belady
IBM Systems Journal 5(2): 78-101, 1966
정책이 보일 수 있는 최적의 행동을 계산하는 간단한 방법(MIN 알고리즘)을 소개하는 논문.

[CD85] **"An Evaluation of Buffer Management Strategies for Relational Database Systems"**
Hong-Tai Chou and David J. DeWitt
VLDB '85, Stockholm, Sweden, August 1985
여러 일반적인 데이터베이스 접근 패턴을 대상으로 사용할 수 있는 여러 버퍼링 전략에 대한 유명한 데이터베이스 논문이다. 좀 더 일반적인 교훈은 이렇다. 워크로드의 특성을 잘 알고 있으면 운영체제에서 보통 사용되는 범용 정책들보다 더 잘 동작하는 정책으로 가다듬을 수 있다.

[Cor69] **"A Paging Experiment with the Multics System"**
F.J. Corbato
Included in a Festschrift published in honor of Prof. P.M. Morse, MIT Press, Cambridge, MA, 1969
시계 알고리즘에 관한 최초의 그리고 엄청 찾기 어려운 참고 문헌이다. 비록 use bit를 활용한 첫 사례는 아니더라도. 우리를 위해서 이 논문을 찾아준 MIT의 H. Balakrishnan에게 감사한다.

[Den70] **"Virtual Memory"**
Peter J. Denning
Computing Surveys, Vol. 2, No. 3, September 1970
가상 메모리 시스템에 대한 Denning이 쓴 초창기의 유명한 개관.

[EF78] **"Cold-start vs. Warm-start Miss Ratios"**
Malcolm C. Easton and Ronald Fagin
Communications of the ACM, 21:10, October 1978
캐시가 완전히 비어 있는 상태로부터 측정한 미스(cold-start miss)와 캐시를 일단 가득 채운 후 측정한 미스(warm-start miss)에 관한 훌륭한 논의.

[FP89] **"Electrochemically Induced Nuclear Fusion of Deuterium"**
Martin Fleischmann and Stanley Pons
Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989

작은 금속이 담겨 있는 물로 거의 무한한 전력을 손쉽게 생성할 수 있는 방법을 제공하여 세상에 혁명을 가져올 뻔한 아주 유명한 논문이다. 불행하게도, Pon과 Fleischmann이 게재하고 광고한 결과는 재현이 불가능한 것으로 판명 났으며, 이 두 선의의 과학자들은 평판이 떨어졌다(그리고 당연히 조롱거리가 되었다). 이 결과로 인해서 유일하게 행복했던 사람은 공동으로 작업을 하였지만 이 논문에서 이름이 제외되었던 Marvin Hawkins이다. 그렇게 Marvin은 20세기의 가장 커다란 과학적 실수에 자신의 이름이 엮이는 걸 피할 수 있었다.

[HP06] **"Computer Architecture: A Quantitative Approach"**
John Hennessy and David Patterson
Morgan-Kaufmann, 2006
컴퓨터 구조에 대한 훌륭하고 놀라운 책이다. 반드시 읽어라!

[Hil87] **"Aspects of Cache Memory and Instruction Buffer Performance"**
Mark D. Hill
Ph.D. Dissertation, U.C. Berkeley, 1987
Mark Hill은 그의 학위 논문 연구에서 3C를 소개하였고 그 이후에 H&P[HP06]에 포함되면서 대중적인 인기를 얻게 되었다. 책의 구절을 인용하면 다음과 같다. "미스를 구분하는 것이 유익하다는 것을 발견하였다... 미스의 원인에 따라 직관적으로 세 가지 그룹으로 나누었다(페이지 49)."

[Kil+62] **"One-level Storage System"**
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner
IRE Trans. EC-11:2, 1962
Atlas가 use bit를 가지고 있기는 했지만 페이지 개수가 매우 적었다. 그렇기 때문에 대용량 메모리에서 use bit를 탐색하는 문제는 저자들이 풀었던 문제에는 포함되지 않았다.

[Mat+70] **"Evaluation Techniques for Storage Hierarchies"**
R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger
IBM Systems Journal, Volume 9:2, 1970
이 논문은 주로 캐시 계층 구조를 어떻게 효율적으로 시뮬레이션 하는가를 다룬다. 분명히 그 분야의 고전이라 할 수 있으며, 여러 교체 알고리즘의 일부 특성에 관한 훌륭한 논의를 담고 있기도 하다. 스택 특성을 이용하면 다양한 크기의 캐시들을 한 번에 시뮬레이션 할 수 있는 이유를 생각해 낼 수 있겠는가?

[MM03] **"ARC: A Self-Tuning, Low Overhead Replacement Cache"**
Nimrod Megiddo and Dharmendra S. Modha
FAST 2003, February 2003, San Jose, California
일부 시스템에서 사용하고 있는 ARC라고 하는 새로운 정책을 포함하여 교체 알고리즘에 대한 훌륭한 최근의 논문이다. 2014년 FAST'14 학회에서 storage systems community가 선정한 "Test of Time" 상의 수상 논문이다.

숙제

`paging-policy.py`라는 시뮬레이터는 여러 페이지 교체 정책들을 대상으로 실험해 볼 수 있게 한다. 자세한 내용은 README를 읽어보자.

문제

1. 다음 인자를 주고 랜덤 주소를 생성하라: `-s 0 -n 10`, `-s 1 -n 10`, 그리고 `-s 2 -n 10`. FIFO 정책에서 LRU 정책으로 그리고 OPT 정책으로 변경하라. 생성된 주소 흐름의 모든 주소에 대해 히트인지 미스인지를 판별하라.
2. 캐시 크기가 5인 경우에 대해서 FIFO, LRU 그리고 MRU 정책을 사용할 때 최악의 주소 참조 스트림을 생성하라(최악의 참조 스트림은 가능한 최대 개수의 미스를 발생시킨다). 최악의 참조 스트림의 경우 성능을 극적으로 향상시켜 OPT의 성능에 근접하게 만들려면 캐시의 크기는 얼마나 커야 하는가?
3. 랜덤한 추적 기록을 생성하라(python 또는 perl을 사용하라). 이런 추적 기록을 적용할 때 여러 정책이 어떻게 동작할 것이라고 예상하는가?
4. 얼마간 지역성을 보이는 추적 기록를 생성하라. 그런 추적 기록을 어떻게 생성할 수 있을까? 생성된 추적 기록대로 메모리 참조가 일어날 때 LRU의 성능은 어떤가? RAND는 LRU에 비해서 얼마나 더 좋은 성능을 보이는가? CLOCK은 어떤 성능을 보이는가? 시계 비트의 수를 변경하였을 때 CLOCK은 어떤 성능을 보이는가?
5. Valgrind와 같은 프로그램을 사용하여 실제 응용 프로그램으로부터 정보를 추출하여 가상 페이지 참조 스트림을 생성하라. 예를 들어, `valgrind --tool=lackey --trace-mem=yes ls` 명령어를 실행시키면 `ls` 프로그램에 의해서 참조된 모든 명령과 데이터에 대한 거의 완벽한 참조 추적 기록을 출력할 것이다. 이 정보를 시뮬레이터에서 사용하기 위해서는 먼저 가상 메모리 참조를 가상 페이지 번호 참조로 변환해야 한다. 가상 주소의 오프셋 부분을 제거한 후(masking off), 오프셋 비트 수만큼 오른쪽으로 시프트하면 된다. 당신의 응용 프로그램의 경우, 참조 요청의 대부분을 만족시키는 캐시의 크기는 얼마인가? 캐시의 크기가 증가할 때 응용 프로그램의 작업 집합의 변화를 그래프로 나타내어라.