요구 페이징 기반 가상메모리 페이지 교체 알고리즘의 구현 및 성능 분석

박경모, 윤여훈
가톨릭대학교 컴퓨터정보공학부
e-mail: kpark@catholic.ac.kr

Implementation and Performance Analysis of Virtual Memory Page Replacement Algorithms Based on Demand Paging

Kyeongmo Park, Yeohoon Yoon
School of Computer Science and Information Engineering
The Catholic University of Korea

요 약

요구 페이징 방식의 가상메모리 시스템에서 페이지 참조 스트링에 따른 페이지 교체 알고리즘의 성능평가를 위한 시뮬레이션을 개발한다. 참조 집약성을 기반으로 워킹세트(working set) 모델을 수정한 MWS 교체 알고리즘을 제안하였고 다양한 교체 알고리즘(FIFO, SC, LFU, LRU, Rand)들과 비교 실험한 결과 MWS는 발생 페이지 부재 수 측면에서 다른 교체 정책 보다 성능이 우수하였다.

1. 서론·

가상메모리(virtual memory)[5,6,7]에서는 프로세스 전체가 메모리에 올라오지 않더라도 실행이 가능하다. 실제의 물리 메모리 개념과 사용자의 논리 메모리 개념을 분리함으로써 사용자 프로그램이 물리 메모리보다 커져도 된다는 것이 가상메모리의 장점이다. 프로그래머는 메모리의 크기나 중첩(overlay)과 관련한 문제로 제약을 받지 않는 반면 가상메모리는 구현하기 어렵고 잘못 사용하면 성능이 저하될 수 있는 단점이 있다. 본 연구에서는 요구 페이징[1] 방식의 가상메모리 시스템을 논의하며 페이지 교체 알고리즘의 성능을 측정한다.

요구 페이징 시스템에서는 그때그때 필요한 페이지만 메모리로 가져온다. 프로세스가 메모리에 올라와 있는 페이지들만 참조하면서 정상적인 실행이 진행된다. 그러나 프로세스가 메모리에 올라와 있지 않거나 참조할 수 없는 페이지를 참조(액세스)하려면 페이지 부재(page fault) 트랩이 발생된다. 이 트랩이 발생하면 운영체제(OS)는 합법적인 액세스인 경우 원하는 페이지를 디스크로부터 메모리로 가져오거나 그렇지 않으면 불법 액세스로 보고한다. 페이지 부재가 발생될 때 인터럽트된 프로세스의 상태를 잘 보관해두어 나중에 이 페이지가 메모리로 올라온 후에 다시 시작할 때 같은 장소의 상태에서 그 프로세스를 다시 수행할 수 있다. 이 방법으로 프로세스의 일부분만이 메모리에 존재하더라도 프로세스를 실행할 수 있다. 프로그램은 한 명령에서도 여러 개의 페이지 부재를 일으킬 수 있다. 이로 인한 시스템 성능의 저하를 초래할 것이다. 실제 수행중인 프로세스들을 분석해 보면 이런 경우는 거의 일어나지 않는다. 프로그램의 어느 한 특정한 부분만을 한동안 집중적으로 참조하는 성질, 다시 말해 참조의 집약성(locality of reference)으로 인하여 요구 페이징은 좋은 성능을 보인다.

다중 프로그래밍[4] 정도를 점차 높여 가면 프로그램당 페이지 개수가 모자라는 상황이 발생된다. 즉 메모리에 프로그램들이 과배당된 것이다. 메모리에 프로세스들을 과배당하면 사용자 프로그램을 수행하다가 페이지 부재가 발생한다. 그러면 하드웨어는 트랩을 발생시키고 OS는 디스크로부터 페이지를

읽어올 준비를 한다. 메모리가 과배당된 상태임으로 빈 페이지 프레임이 없다. 이를 위한 대처 방안으로 해당 프로세스를 잠시 스왑 아웃(swap-out)하여 내보내고, 그 프로세스가 가지고 있던 프레임 모두를 할당 해제시켜 다중 프로그래밍 정도를 줄일 수 있다. 만일 빈 프레임이 없다면 페이지 교체 알고리즘을 사용하여 사용이 덜 빈번하게 이루어지고 있는 프레임을 찾아서 그것을 비워버린다. 요구 페이징 시스템에서 페이지 교체 알고리즘은 페이지 교체가 요구될 때 어는 프레임을 희생자(victim)로 택할 것인가를 결정한다. 여러 가지 페이지 교체 알고리즘 중에서 가장 낮은 페이지 부재율을 얻을 수 있는 것을 선정한다. 본 연구에서는 워킹세트(working set) 모델[2,3]을 변형한 교체 알고리즘을 제시하고 다른 페이지 교체 알고리즘(FIFO, SC, LRU, LFU, Random)과 성능을 비교하고 평가한다.

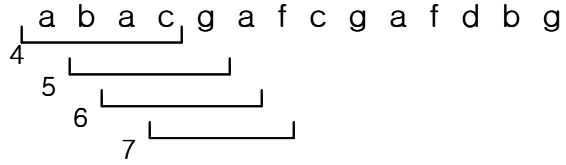
본 논문의 구성은 다음과 같다. 2장에서는 워킹세트 모델에 기초한 교체 알고리즘 개발을 위한 기본 아이디어를 기술한다. 3장에서는 워킹세트 교체 알고리즘과 비교할 5개의 교체 알고리즘들에 대하여 요약 설명한다. 4장에서는 2,3장에서 기술한 교체 알고리즘들의 성능을 분석하고 평가한다. 마지막으로 5장에서 결론을 맺는다.

2. 워킹세트 모델

워킹세트(working set: WS) 모델[2,3,5]은 먼저 프로세스가 실제로 사용하고 있는 프레임의 수가 몇 개인가를 알아보는 것에서 출발한다. 이 모델은 프로세스 실행의 집약성(locality)을 기반으로 두고 있으며 개념적으로 WS 윈도우(window)를 가지고 있다. 이 윈도우의 크기는 시간의 단위를 가지며 "△ 개의 메모리 참조시간"을 나타낸다. 한 프로세스가 최근 △개의 페이지를 참조했다면 그 안에 들어있는 서로 다른 페이지들의 집합을 워킹세트 또는 작업집합이라 부른다. 워킹세트는 프로그램 집약성의 근사값이 된다. 메모리 요구에 있어 역동적으로 변화에 대한 처리를 고려하는 교체 방법은 이용하면 고정식 메모리 할당 방법보다 더 좋은 수행을 한다. WS 교체 알고리즘은 시간에 따라 이동하는 윈도우를 사용한다. 주어진 시간에 참조되지 않는 페이지는 워킹세트에서 제거된다.

메모리 참조로 측정되는 윈도우 크기 △에 대하여 시간 t에 WS는 시간 간격 (t - △ +1, t)에 참조되어진 페이지들의 집합이다. 페이지가 더 이상 WS에 속해 있지 않으면, 페이지 부재 시 반드시 교체되지는 않지만, 해당 페이지는 교체될 수 있다.

예를 들어, 7개의 가상 페이지 {a, b, …, g}와 4개를 참조하는 윈도우, 참조열 a b a c g a f c g a f d b g를 지닌 프로그램에서 (그림 1)은 다양하게 이동하는 윈도우를 보여준다. WS는 이 윈도우 내에 들어있는 페이지들의 집합이다. <표1>에서는 각 시간대에 따라 참조되는 페이지가 무엇인지를 보여주는 WS의 내용을 나타낸다.



(그림 1) 작업집합 교체 (△ = 4)

1	2	3	4	5	6	7
a	ab	ab	abc	abcg	acg	acgf
8	9	10	11	12	13	14
acgf	acgf	acgf	acgf	agfd	afdb	fdbg

<표 1> 각 시간(1-14)별로 참조되는 작업집합 내용

기본적인 워킹세트 교체 정책은 페이지 부재가 일어날 때에만 페이지를 교체하는 방안에 비하여 본 논문에서 제시하는 수정워킹세트(Modified Working Set: MWS) 정책은 페이지 부재 시에 다음 2가지 경우로 나누어 처리할 수 있다.

[1] 만일 모든 페이지가 WS에 속해 있으면 즉, 페이지 부재 전에 윈도우 W_t 시간 내에 액세스되었다면, 한 페이지만큼 WS를 증가시킨다.

[2] 만일 하나 또는 하나 이상의 페이지가 WS에 속해 있지 않으면 즉, 페이지 부재 전에 윈도우 W_t 시간 내에 참조되지 않았다면, 최근 맨 마지막으로 사용된 페이지를 버림으로써 WS를 감축시킨다.

3. 페이지 교체 알고리즘

3-1. FIFO (First-In First-Out)

FIFO 알고리즘은 가장 간단한 페이지 교체 알고리즘으로 어떤 페이지를 교체해야 할 때, 메모리에 올라온 지가 가장 오래된 페이지를 빼낸다. 페이지가 올라온 시간을 페이지마다 기록해도 되지만 페이지들이 올라온 순서로 FIFO 큐(queue)를 만든다. 큐의 머리(head) 부분 페이지를 교체하고 메모리에 새로 올라온 페이지는 큐의 꼬리(tail)에 삽입한다.

3-2. SC (Second Chance)

2차 기회 교체 알고리즘은 FIFO 알고리즘을 약간 수정한 것이며 원형 큐(circular queue)로 구현된다. 이 큐에는 포인터가 있어 다음에 교체될 페이지를 가리킨다. 어떤 프레임이 필요하면 포인터는 참조비트의 값이 0인 페이지를 발견할 때까지 큐를 뒤진다. 포인터가 돌아가면서 참조비트 값이 1인 것은 0으로 바꾼다. 희생될 페이지(0인 페이지)가 발견되면 그 페이지는 교체되고 새로운 페이지를 순환 큐의 해당 위치에 삽입한다. 최악의 경우, 모든 비트 값이 1이었다면 포인터는 큐를 완전히 한 바퀴 돌게 되면서 0으로 설정한다. 그 다음 모든 페이지는 2차의 기회가 주어진다. 두 번째 돌 때는 FIFO와 같다.

3-3 LFU (Least Frequently Used)

LFU는 참조 회수가 가장 작은 페이지를 교체하는 방법이다. 이러한 선택의 이유는 활발하게 사용되는 페이지는 큰 참조 회수 값을 가져야 한다는 논리이다. 이 알고리즘은 어떤 프로세스가 초기 단계에서 한 페이지를 집중적으로 많이 사용하지만 후에 다시 그 페이지를 사용하지 않을 경우에는 성능이 떨어질 수 있다. 이를 위한 해결책으로 참조회수를 일정한 시간마다 하나씩 오른쪽으로 쉬프트해서 지수적으로 값을 감소시키는 방법을 사용한다.

3-4. LRU (Least Recently Used)

LRU 알고리즘은 가장 오래 동안에 사용되지 않은 페이지를 교체한다. LRU 정책은 페이지 교체 알고리즘으로 자주 이용된다. 프레임들을 최근 사용된 시간 순서로 파악할 수 있어야 한다. 구현된 LRU 기법에서는 페이지 번호의 스택(stack)을 유지하는 방법으로 페이지가 참조될 때마다 그 페이지의 번호를 스택에서 제거하여 꼭대기(top)에 놓는 방식이다. 스택의 꼭대기는 항상 가장 최근에 사용된 페이지이고 밑바닥(bottom)은 가장 오랫동안 이용되지 않은 페이지이다.

LRU 교체는 FIFO에서 처럼 프레임을 프로세스에 더 주었는데 오히려 페이지 부재율이 더 증가하는 벨러디의 모순(Belady's anomaly) 현상을 야기하지 않는 '스택 알고리즘'이다. 스택 알고리즘은 n개의 프레임에 수록되는 페이지의 집합이 항상 n+1개의 프레임에 수록되는 페이지의 집합의 부분집합이 되는 알고리즘이다.

3-5. Random Replacement

랜덤 교체 정책에서는 주 메모리에 현재 있는 모든 페이지들에서 교체할 페이지를 무작위로 선택한다. 일반적으로 랜덤 교체 알고리즘은 좋지 않으나 다른 교체 정책과 비교 목적으로 유지한다.

4. 성능 평가

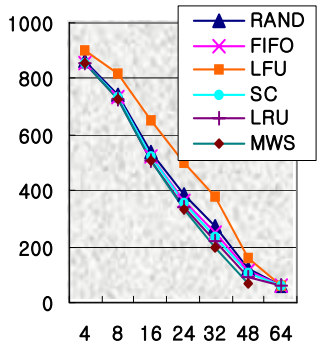
제안 MWS를 포함하여 6개의 페이지 교체 알고리즘 시뮬레이션을 개발하여 발생 페이지 부재 수를 비교 평가한 성능실험 결과를 보고한다. 구현 및 실험에서는 Intel Celeron, 1GHz, 256MB RAM 사양의 PC/Windows XP에서 Visual C++ 6.0 언어를 사용하였다. 페이지 교체 알고리즘의 성능은 CPU가 생성해내는 메모리 주소의 열(sequence)을 가지고 분석하고 평가한다. 이 열을 참조 스트링(reference string)이라 부른다. 실제로는 메모리 주소 대신에 페이지 번호로 구성된 페이지 참조 스트링을 가지고 페이지 교체 알고리즘을 비교한다. 이는 페이지 단위로 페이지 부재율을 계산하기 때문이다.

페이지 참조 작업은 다음 3가지 중에 하나로 귀착된다. 1) 요청된 페이지가 메모리에 이미 있으면 적중(hit)이 된다. 2) 그 페이지가 없으면 부적중(miss)이다. 할당하기 위한 메모리 페이지를 사용할 수 있으면 요청된 페이지를 사용한다. 3) 페이지 부재가 발생하여 모든 페이지들을 현재 사용하고 있다면 페이지 교체 알고리즘을 선택하여 페이지를 교체해야 한다.

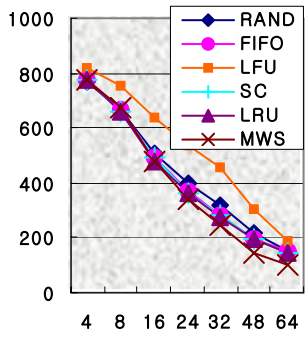
시뮬레이션에서는 랜덤 페이지 참조 생성방식뿐 아니라 샘플 참조 스트링방식으로 페이지 요청열에 존재하는 집약성을 고려한 MWS 정책을 사용하여 다른 페이지 교체 정책들과의 성능차이를 확인할 수 있었다. 또한 실험에서는 높은 집약성 파악을 위해 페이지 범위를 확장하고 페이지 번호가 나올 확률을 스트링 패턴별로 계산하여 WS 변화를 추적하여 시스템 성능을 측정한다.

그림 2, 3은 6개 페이지 교체 알고리즘을 사용하여 프레임 수에 따른 페이지 부재수를 비교 측정한 것이다. 그림에서 가로축은 프레임의 수를 나타내고, 세로축은 페이지 부재수를 나타낸다. 페이지 참조 방식은 그림 2는 참조집약적인 스트링을 사용한 결과이고, 그림 3은 이러한 참조집약 스트링에 작업집합 성질을 포함시킨 스트링을 사용한 결과이다.

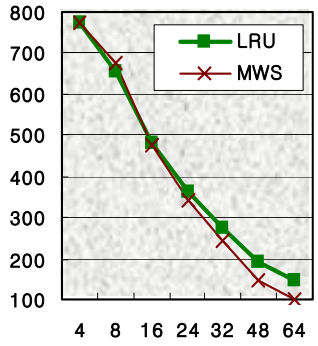
실험 결과를 살펴보면 LFU 알고리즘이 다른 알고리즘보다 부재수가 유난히 높은 것을 볼 수 있는데 이는 스트링의 참조집약적인 성질이 LFU에서는 오히려 역효과를 가져오기 때문이다. 즉, 스트링이 한 지역을 집중 참조하다가 다른 지역으로 넘어가는



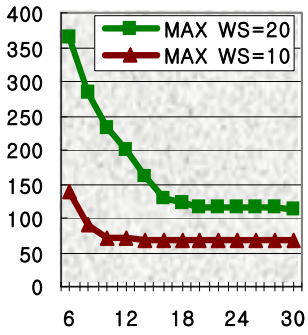
(그림 2)



(그림 3)



(그림 4)



(그림 5)

경우 앞으로의 참조스트링은 새로운 지역의 스트링에 집중될 것임에도 불구하고 LFU는 현재까지 참조된 수가 작다는 이유로 이 지역의 페이지를 일정기간동안은 희생 페이지로 선택할 것이기 때문이다.

시뮬레이션 결과에서는 프레임수가 증가할수록 페이지 부재수가 감소하는 것을 확인할 수 있고, MWS의 부재수가 다른 알고리즘에 비하여 더 낮은 것을 확인할 수 있었다. 또한, 그림 3의 결과는 참조집약 스트링에 비해 작업집합 성질이 포함된 스트링을 이용하는 MWS가 다른 알고리즘에 비하여 더 나은 성능을 보여준다.

그림 4는 그림 3에서의 LRU와 MWS만을 따로 그린 것이다. 그림을 보면 프레임수가 적은 부분에서는 LRU와 MWS가 비슷한 성능을 보이다가 프레임수가 일정 수 이상으로 커지면서 MWS가 LRU보다 좋은 성능을 나타냄을 확인 할 수 있다. 이는 MWS의 WS 윈도우 크기에 관련된 것으로 윈도우 크기가 작업집합을 수용할 만한 크기가 되어야만 MWS의 성능이 나타남을 보여주는 것이다. 그림 5는 작업집합의 최대 크기를 20, 10 으로 설정하여 WS 윈도우 크기에 따른 페이지 부재수를 그린 것이다. 그래프를 보면 윈도우 크기가 작업집합의 크기보다 작은 곳에서는 페이지 부재수가 현저하게 많은 것을 볼 수 있는데, 이것은 윈도우 크기가 작업집합을 충분히 수용하지 못하여 발생하는 현상이다. 반면 윈도우 크기가 작업집합 보다 큰 곳에서는 페이지 부재수가 크게 차이 나지 않는 것을 볼 수 있다. 이것은 윈도우 크기가 무조건 크다고 해서 좋은 것이 아니라 작업집합을 수용할 만큼의 크기가 되었을 때 MWS가 가장 최적화된 성능을 나타냄을 보여주는 것이다. 이러한 작업집합 기법을 이용하면 가능한 최대의 다중 프로그래밍 정도를 유지하면서 스라싱(thrashing)을 방지하고 CPU의 이용률도 최적화해주는 장점이 있다. 그러나 기억장소 참조 집약성을 이용하여 적정한 WS 크기를 결정하는 것이 중요한데 프로그램의 효율적 실행을 위해서 WS는 주 기억장치에 상주해야 하고 WS를 추적하는 일이 어려운 점이다. 매 메모리 참조마다 상주 페이지 세트를 재조정해야 하므로 오버헤드가 높다.

5. 결론

본 논문에서는 요구 페이징 방식의 가상메모리 시스템을 논의했고 성능 문제에 대한 복잡성과 페이지 교체 알고리즘의 비용을 측정하는 시뮬레이션을 개발하였다. 워킹세트 모델을 수정한 페이지 교체 알고리즘 MWS 정책을 제안하였고 다른 교체 알고리즘(FIFO, SC, LFU, LRU, Rand)과의 성능을 비교 평가하였다.

참고문헌

[1] Avi Silberschatz, Peter Galvin, and Greg Gagne, “Operating System Concepts,”, 7th Edition, John Wiley and Sons, 2005.

[2] Peter J. Denning, “The Working Set Model for Program Behavior,” Communications of the ACM, 19(5), 1976, pp. 285- 294.

[3] Peter J. Denning, “Working Set Today,” IEEE Computer Software and Applications Conference, 1978 (COMPSAC'78), Nov. 13-16, 1978, pp. 71-77.

[4] Peter J. Denning, “A Short Theory of Multiprogramming,” Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Jan. 18-20, 1995, pp. 2-7.

[5] Peter J. Denning, “Virtual Memory,” ACM Computing Surveys, 2(3), Sep. 1970, pp. 153-189.

[6] Peter J. Denning, “Virtual Machines,” IEEE, Volume: 23, Issue: 3, July-Sept. 2001.

[7] Komino Tomoo, “Programming Methods Virtual Memory System Environment,” IPSJ Magazine, 21(4), March 2001.