

# New Page Replacement Policy

소프트웨어학과 201521032 한태희  
소프트웨어학과 201720733 신승현

## 1. 개요

리눅스 커널 학습 최종 프로젝트는 두가지 파트를 진행한다. 첫번째 파트는 리눅스 커널 5.15.4 버전에 second chance LRU-approximation algorithm 방식으로 구현되어 있는 Memory Management 소스 코드를 분석하여 LRU 리스트의 변화를 추적하고 출력하는 시스템 콜을 만드는 것이다. 두번째 파트는 기존의 LRU algorithm이 아닌, counter-based clock algorithm으로 페이지 관리 방법을 변경하는 것이다. 유저 프로그램을 통해 두 방식에서의 성능 및 결과를 비교한다.

## 2. 구현 내용

### 2.1 Memory management statistics

현재 LRU list에 존재하는 정보를 제공하는 시스템 콜과 그 정보를 출력하는 유저 레벨 프로그램을 작성한다. Memory management statistics 시스템 콜(memstat\_syscall)에서 제공하는 데이터는 다음과 같다.

1. 현재 LRU 리스트의 active anon / inactive anon / active file / inactive file 페이지의 개수
2. 현재 active / inactive 페이지 중 reference bit가 활성화 된 페이지의 개수
3. 현재까지 promotion / demotion 된 페이지의 개수
4. 현재까지 inactive list에서 eviction 된 페이지의 개수

먼저 1, 2번에 대한 정보를 얻을 수 있도록 현재 LRU 리스트에 접근하기 위해 NODE\_DATA() 매크로를 사용해 현재의 page list를 얻어낸다. 실습환경은 노드가 1개인 상태이므로 contig\_page\_data, 즉 유일한 뱅크에 존재하는 전역변수 page list data로 접근할 수 있다. 얻어낸 page list에서 LRU 리스트를 순회하기 위해 LRU 리스트의 정보를 담고 있는 lruvec 멤버로 접근하여 먼저 spin\_lock\_irq api로 critical section을 생성한다. lruvec 구조체는 active anon / inactive anon / active file / inactive file 페이지에 대한 정보를 리스트 구조로 포함하고 있으므로 각각 list\_for\_each\_safe api를 통해 순회하여 페이지의 개수를 센다. 이 때, 각 페이지에 대해 PageReferenced() 매크로를 통해 해당 페이지의 PG\_referenced 플래그의 값이 활성화 되어있는지 비트값을 비교하여 확인해 reference 된 페이지의 개수를 함께 센다.

3번의 promotion 된 페이지의 누적 개수를 구하기 위해 LRU 리스트의 관리를 담당하는 mm/swap.c 파일의 mark\_page\_accesed 함수에서 promotion이 일어나는 구간에 전역변수 카운터를 위치해 promote 된 횟수를 저장한다. 해당 카운터는 시스템 콜 프로그램에서 접근하기 위해 swap.h 헤더파일에 extern으로 선언한다.

3번의 demotion 된 페이지와 4번의 eviction된 페이지의 개수를 구하기 위해 마찬가지로 extern으로 선언된 카운터를 mm/vmscan.c 파일에서 사용한다. Demotion이 수행되는 함수인 shrink\_active\_list() 에서는 demote 카운터를, eviction이 수행되는 함수인 shrink\_inactive\_list() 에서는 evict 카운터를 위치해 각 동작이 수행되는 페이지의 개수를 더해준다.

최종적으로 구한 1~4번에 대한 데이터를 정수형 포인터에 copy\_to\_user api로 복사하여 유저레벨 프로그램에서 사용할 수 있도록 한다.

## 2.2 Count-based clock page replacement

Reference counter를 동작시키기 위해 타이머를 통해 1초의 주기로 현재 LRU 리스트의 페이지들의 카운터를 증가시키거나 감소시키는 커널 모듈을 구현한다. PageReferenced() 매크로를 통해 각 페이지가 최근에 reference 되었다면 카운터를 증가시키고, 아니라면 카운터를 감소시켜 주기별로 카운터의 값을 변화시킨다. 그리고 try\_to\_free\_pages() 가 호출된 경우, LRU 리스트들에 대하여 각각 가장 reference counter 값이 적은 페이지를 리스트의 마지막으로 이동시킨다.

Count-based clock page replacement는 기존의 LRU에서 사용하는 4개의 리스트 자료구조를 수정하지 않고, 그대로 이용하는 방식으로 추가적인 구현을 하였다. 우선, 기존 리눅스의 struct page에 있는 reference count는 현재 해당 페이지를 참조중인 객체의 개수만을 보여주기에, Count-based 교체 방식을 위해 곧바로 사용하기엔 부적합하다. 따라서, 누적된 참조 횟수를 표시하기 위해 struct page에 my\_ref\_c라는 int 형 변수를 하나 추가하였다. my\_ref\_c는 \_\_init\_single\_page 함수에서 페이지를 생성할 때, 0으로 초기화 한다.

my\_ref\_c 는 주기적으로 값이 증가하거나, 감소해야 한다. 그런데, 페이지 reclaim이나 evict 때 해당 변수를 더하거나 빼면 주기적인 감시를 할 수가 없다. 이를 해결하기 위해서, 별개의 IruCount라는 모듈을 만들었다. IruCount 모듈은 리눅스 커널 타이머 인터럽트를 이용해서 1초마다 ref\_counter\_watchdog 함수를 실행시킨다.

ref\_counter\_watchdog 함수는 Iruvec 객체를 구하고, 스판락을 통해 리스트 접근을 보호하고, 각 LRU 리스트에 대해서 count\_ref\_c\_in\_lru 함수를 호출한다.

count\_ref\_c\_in\_lru 함수는 LRU 리스트를 순차적으로 탐색한다. 각 페이지에 대하여 페이지가 PageReferenced 함수를 통해 어딘가에서 참조되어 있는 것이 확인되면 my\_ref\_c 값을 1 더하고, 참조되어 있지 않다면 my\_ref\_c 값을 1 뺀다. my\_ref\_c는 최대 10000, 최소 -10000의 범위만을 가지며, 이를 벗어나는 범위로는 더하거나 빼지 않는다. my\_ref\_c는 중간 값인 0으로 초기화 되며, 자주 접근되는 페이지일수록 높은 점수를, 자주 접근되지 않는 페이지 일수록 낮은 점수를 받게 된다.

탐색 작업의 주기를 1초로 잡은 이유는, 이것보다 더 짧은 주기로 ref\_counter\_watchdog 함수를 실행시켰을 경우, 나중에 자세히 설명할 어플리케이션 레벨에서의 측정이 더 안좋게 나왔기 때문이다.

가상머신 실행 후, 위에서 설명한 IruCount 모듈을 insmod 하게 되면, my\_ref\_c가 변화하게 된다. 해당 모듈을 가상머신 부팅시에 자동으로 실행하게 했으면 더 편리했겠지만, 아쉽게도 해당 기능은 구현하지 못했다. 따라서 IruCount 모듈은 수동으로 실행해야 한다.

IruCount 모듈이 작동하여 my\_ref\_c 값이 변화하면, 이 값을 이용해 페이지 관리 방식을 바꿀 수 있다. 이때 적용되는 논리는 LFU (Least Frequently Used)이다. 이것은 “페이지는 모두 동일한 확률로 호출되지 않는다. 대부분의 경우, 자주 사용되는 페이지만 계속해서 사용하고, 자주 사용되지 않는 페이지는 극히 드물게 사용된다.” 라는 전제를 기반으로 작동한다. my\_ref\_c가

높은 페이지는 지속적으로 참조되어 있는 것이 확인되어 높은 점수를 받고 있는 것이고, `my_ref_c`가 낮은 페이지는 참조가 매우 드물게 되어 앞으로도 참조가 발생할 가능성이 낮다고 간주한다.

위 결론에 따라서 demotion(ACTIVE에서 INACTIVE로 강등)이나 eviction(INACTIVE에서 퇴출)이 발생할 경우, 기존 LRU의 tail에 존재하는 페이지 (Least Recent Page)가 아닌, `my_ref_c`가 가장 낮은 페이지 (Least Frequent Page)를 이동시키도록 하면 된다. 다만, 위 구현을 짧은 프로젝트 기간에 작성하기는 힘들기 때문에, Least Frequent Page를 리스트에서 빼낸 뒤, 다시 리스트의 맨 끝 (tail)에 집어 넣는 편법을 사용하였다. 이렇게 구현하면, 기존에 구현되어 있는 리눅스의 Memory Management는 기존의 second chance LRU-approximation과 동일한 작업을 수행하지만, 실제로 리스트에서 빠져 나가는 것은 Least Recent Page가 아닌 Least Frequent Page가 되어 Count-based clock page replacement가 작동하게 된다.

이를 위해서 현재 남은 페이지가 부족할 경우 다른 페이지를 free 하도록 호출하는 `try_to_free()` 함수에 replacement 기능을 구현했다. 기존의 `try_to_free` 함수의 기능이 작동하기 이전에, `IcuCount` 모듈에서 사용했던 방법과 마찬가지로 4가지 LRU 리스트를 순차적으로 탐색한다. 그리고 가장 `my_ref_c`가 낮은 페이지를 리스트의 맨 끝으로 옮겨 넣는다. (동일한 크기인 페이지가 있을 경우, tail에 더 가까운 것을 선택)

### 3. 실행

지금까지 수정한 커널의 동작을 확인하기 위해서, c 코드를 이용해서 간단한 테스트 프로그램을 만들고 테스트를 진행하였다. 테스트의 자세한 작동과 결과는 4번 항목에서 기술한다. 아래의 스크린샷은 파트1과 파트2가 모두 구현된 리눅스 커널이 작동하는 가상 머신에서 테스트 프로그램을 실행시킨 결과이다.

```
root@sce394_vml:~/project/user# ./memstat_user
Active File page      : 4565
Inactive File page    : 9986
Active Anon page      : 49
Inactive Anon page    : 1788

Referenced Active page : 2510           테스트 이전 LRU 리스트 통계
Referenced Inactive page : 6705

Cumulated promoted page : 3402
Cumulated demoted page : 0

Cumulated evicted page : 0           시스템 콜에 소요된 시간
Total time = 1234 microseconds

INIT: Id "S1" respawning too fast: disabled for 5 minutes      테스트에 소요된 시간
TEST TIME is 34030779 microseconds

Active File page      : 2193
Inactive File page    : 14
Active Anon page      : 28
Inactive Anon page    : 545

Referenced Active page : 1058           테스트 이후 LRU 리스트 통계
Referenced Inactive page : 23

Cumulated promoted page : 3416
Cumulated demoted page : 1

Cumulated evicted page : 3614
Total time = 139 microseconds
```

## 4. 성능 측정 및 결과 비교

테스트 애플리케이션은 한가지 작동 방식에서 4가지 종류의 변형을 만들었다. 우선, 기본적인 골자는 가상 머신 테스트 환경의 램 크기보다 더 큰 메모리 공간을 `malloc`을 통해 할당 받는다. 구체적으로는 1MB 크기의 메모리 공간을 7만번 반복하여 할당받는다. (약 70GB) 그 뒤, 각 메모리 구간에 접근하여 쓰기를 수행하는 것을 999번 반복한다. (총 쓰기 횟수 7만\*999 = 6993만회) 쓰기는 배열의 첫번째 지점에 정수 하나만을 새로 덮어 쓰는 작업을 한다.

여기서 각 메모리 구간에 접근하는 방법에 따라 4가지 종류의 변형을 만든다. **Seq**는 7만개의 메모리 공간을 순차적으로 접근한다. **Half Seq**는 7만개의 메모리 공간 중 앞의 절반만을 순차적으로 접근하는 것을 2회 반복한다. (자주 사용되는 메모리와 그렇지 않은 메모리 유사 구현) **Rand**는 7만개의 메모리 공간을 임의로 7만번 접근한다. **Half Rand**는 7만개의 메모리 공간 중 절반의 공간을 임의로 7만번 접근한다.

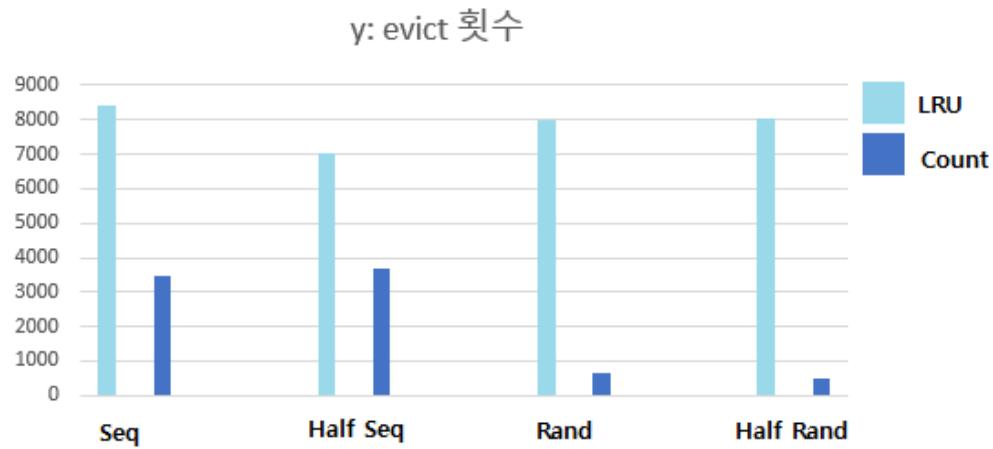
Seq	promot	demot	evict	time(ms)		Half Seq	promot	demot	evict	time(ms)
Iru	59	2342	8415	47873		Iru	60	1530	7042	44177
counter	15	0	3451	42625		counter	15	21	3667	45447
Rand	promot	demot	evict	time		Half Rand	promot	demot	evict	time
Iru	54	2145	8005	33621		Iru	59	2103	8041	34568
counter	15	0	639	34236		counter	15	0	506	34190

이를 통해서 위 표와 같은 테스트 결과를 얻을 수 있었다. 각 결과는 테스트 이전과 이후의 값을 빼서 구한 값이다.

**promotion**은 패치 이후가 더 적게 일어나는 경향이 있었지만, 그렇다고 해서 패치 이전이 매우 많이 발생하는 것은 아니기 때문에 주목할만한 사항은 아니라고 볼 수 있다. 테스트에 사용한 어플리케이션은 매우 많은 메모리를 할당받고 호출하는 방식이라서 대부분의 페이지들은 **INACTIVE**에서 **ACTIVE**로 올라가는 것이 아닌, **EVICTED**에서 **ACTIVE**로 바로 활성화 되기 때문인 것이 이유로 생각된다.

테스트에 진행된 시간은 서로간에 크게 차이가 나지 않아서 어느 한쪽이 더 우수한 처리 속도를 가지고 있다고 결론 내리기 힘든 결과가 나왔다. 구현 과정에서 진행했던 별도의 간단한 테스트로는 **Clock** 주기를 1초보다 더 짧게 하면 오히려 처리 속도가 나빠졌고, **Clock** 주기를 1초보다 더 길게 한다고 해서 처리 속도가 개선되지는 않았다. 기존의 리눅스 MM 구조도 나쁜 성능의 알고리즘은 아니기 때문에, **Least Frequent** 계산을 따로 한다고 해서 처리 속도 향상을 얻기는 힘든 것으로 생각된다.

위 두개 요소는 유의미한 향상을 얻지 못했지만, **Counter-based clock page replacement**가 기존 **second chance LRU-approximation algorithm**에 비해서 **demotion** 횟수와 **eviction** 횟수가 매우 적게 일어난 것을 확인할 수 있었다.



특히 주목할만한 사항은 **evict 횟수**의 차이이다. **evict**는 가상의 논리 공간이 바뀌는 것이 아니라, 실제 물리적 저장 위치를 바꾸는 행위이기 때문이다. 기존의 **second chance LRU-approximation** 방식은 모든 경우에 7000~8000 회 정도의 **eviction**이 발생했지만, **Counter-based clock** 방식은 Seq 쓰기 테스트에서 3000~4000, Rand 쓰기 테스트에서 1000 이하의 **eviction 횟수**로 훨씬 적은 페이지 **eviction**이 발생함을 확인할 수 있었다.

## 5. 결론

**Counter-based clock** 방식을 구현하더라도, 기존 리눅스의 **second chance LRU-approximation** 방식에 비해서 처리 속도로는 유의미한 차이를 얻을 수 없었다. 하지만, **eviction 횟수**는 매우 적게 줄일 수 있었다.

**page eviction**은 스토리지를 무한히 사용 가능한 자원으로 간주하고 지금 사용하지 않는 페이지를 스토리지에 임시로 내려 놓는다는 개념이지만, 실제로는 스토리지도 유한하고 한계가 있는 자원이며, **page eviction**을 진행하면 당연히 오버헤드가 발생하게 된다.

**Counter-based clock** 방식을 사용한다면, **eviction 횟수**를 줄여서 스토리지에 가해지는 부담을 줄일 수 있을 것이다.

예를들어, 임베디드 시스템이나 저가형 넷북, 스마트폰 같이 스토리지의 용량이 매우 한정적인 경우 **swap** 공간을 위해서 스토리지의 일부를 할애하는 것도 매우 부담스러운 일이 된다. 이 경우, **Counter-based clock** 방식의 페이지 교체를 하게 되면, 상대적으로 더 적은 양의 **swap** 공간 만 있어도 되므로, 스토리지 공간에 대한 압박이 적어지게 된다.

스토리지의 수명이 무한하지 않다는 점도 **Counter-based clock** 방식의 페이지 교체를 사용하기에 좋은 이유가 될 수 있다. 최근 고성능 스토리지로 널리 사용되는 **SSD**의 경우, 구조적인 문제로 인해 읽기 쓰기를 반복할 경우 플래시 메모리의 수명이 짧아진다는 문제가 있다. **SSD** 파일 시스템에서 쓰는 공간을 주기적으로 바꾸는 방식으로 이러한 문제를 보완하긴 하지만, 읽기와 쓰기가 빈번하게 발생되어 스토리지의 수명을 줄이는 것을 막을 수 없다. **Counter-based clock** 방식을 사용한다면 원천적으로 **swap** 공간 쓰기로 인한 횟수를 크게 줄여서 **SSD**의 수명을 늘릴 수 있을 것이다.