공학석사 학위논문

리눅스 기반 모바일 기기에서 사용자 응답성 향상을 위한 프레임워크 지원 선별적 페이지 보호기법

Framework-assisted Selective Page Protection for Improving Interactivity of Linux Based Mobile Devices

2016 년 2 월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템전공

김 승 준

리눅스 기반 모바일 기기에서 사용자 응답성 향상을 위한 프레임워크 지원 선별적 페이지 보호 기법

Framework-assisted Selective Page Protection for Improving Interactivity of Linux Based Mobile Devices

지도 교수 홍 성 수

이 논문을 공학석사 학위논문으로 제출함

2016 년 1 월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템전공

김 승 준

김승준의 공학석사 학위논문을 인준함

2016 년 1 월

위 원 장 박 재 흥 (인)

부위원장 홍 성 수 (인)

위 원 안 정 호 (인)

초 록

스마트폰과 같은 모바일 기기가 널리 보급됨에 따라 사용자들은 모바일 기기 응용들을 사용하면서 빠른 응답성을 제공받기를 바란다. 하지만 모바일 기기 응용들은 종종 사용자가 기대하는 수준의 응답성을 제공하지 못한다. 응답성을 저해하는 주 원인들 중 하나는 과도한 페이지 폴트 발생에 따른 대화형 태스크 수행의 지연이다. 이는 대화형 태스크의 상주 페이지(resident page)들이 비대화형 태스크와의 페이지 캐시 경쟁에 의해 더욱 빈번히 희생될 페이지(victim page)으로 선정되어 스토리지로 쫓겨나기 때문이다. 이 논문은 이러한 문제를 해결하기 위해 프레임워크 지원 선별적 페이지 보호 기법을 제시한다. 제안한 기법은 프레임워크 레벨에서 대화형 태스크를 식별하고 이를 커널에 전달하여 페이지 replacement 시에 대화형 태스크의 페이지를 보호하고, 사용자 입력 처리 중에 발생하는 페이지 폴트를 줄인다. 실험 결과 제안된 기법은 기존 시스템에 비해 페이지 폴트 횟수를 37% 감소시켰고, 응답시간을 11% 단축할 수 있었다.

주요어 : Linux; Interactivity; Mobile devices; Page cache;

학　　번 : 2014-24841

목 차

그림 목차

제 1 장 서 론

스마트폰 보급 됨에 따라 사용자들이 고사양, 고성능 응용들을 요구하고 있다0. 특히 이러한 응용들의 응답성은 사용자 경험에 있어서 중요한 요소들 중 하나이다. 스마트폰 제조사들은 모바일 기기의 사용자 응답성을 개선시키기 위해 많은 노력을 하고 있다. 하지만 이러한 노력에도 불구하고 여전히 양질의 응답성을 제공받지 못하는 경우가 보고되고 있다0.

고사양 응용들은 대용량 컨텐츠들과 라이브러리를 사용하기 때문에 빈번하게 스토리지를 사용한다. Linux 커널에서 스토리지에 대한 접근은 요구 페이징(demand paging)에 의해 처리 된다. 응용이 요청한 페이지가 페이지 캐시에 부재한 경우, 페이지 폴트 핸들러가 페이지 폴트를 발생시켜 해당 페이지가 페이지 캐시에 적재된다. 특히, 메모리 제약이 심한 모바일 기기에서는 페이지 캐시의 크기가 데스크탑 환경에 비해 작기 때문에 페이지 폴트 발생 빈도가 높아진다. 실험을 통해 확인한 결과 사진 갤러리 응용은 USB와 같은 비대화형 응용의 I/O 작업이 있을 때, 최악의 경우 약 10000번 이상의 페이지 폴트가 발생한다. 그림 1은 비대화형 응용의 I/O작업이 수반됨에 따라 증가하는 페이지 폴트 수 실험 결과를 나타낸다. 따라서 사용자 응답성 개선에 있어서 페이지 캐시 관리를 최적화하여 페이지 폴트 발생 횟수를 줄이는 것은 중요하다.

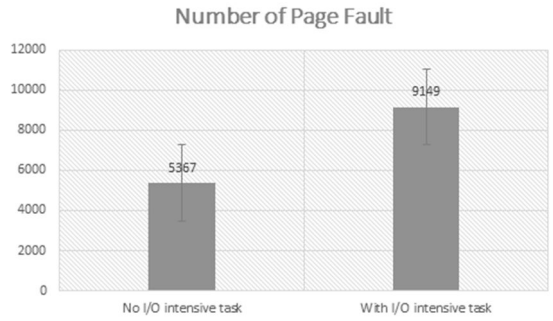

그림 1. 비대화형 응용 I/O 수행에 따른 페이지 폴트 수 측정 결과.

페이지 캐시 적중률을 높이기 위한 페이지 관리 정책은 선페이징(prepaging) 정책과 페이지 교체(page replacement) 정책으로 나뉜다. 리눅스에서는 선페이징 정책으로 readahead 매커니즘을 제공한다. 파일 페이지를 읽을 경우 순차적으로 읽는 패턴이 있음을 가정하고 읽은 파일 페이지 이후의 일부 파일 페이지를 순차적으로 미리 메모리에 적재함으로써 페이지 캐시 적중률을 높인다0. 페이지 교체 정책은 기본적으로 LRU-2(least recently used)에 기반한다. 이를 위해 커널은 모든 프로세스의 작업 세트(working set)를 포함하는 active_list와 물리 메모리 회수 대상인 남은 페이지들을 포함하는 inactive_list를 관리한다. Johnson etl.에 따르면 이 정책은 다른 LRU-k에 비해 5~10% 정도의 높은 페이지 캐시 적중률을 갖는다0. LRU-k는 페이지의 최근 k번의 참조 정보를 토대로 희생될 페이지를 결정하는 페이지교체 정책이다.

그러나 이러한 리눅스의 페이지 캐시 정책은 사용자 응답성 측면에 있어서 한계가 있다. 입력 처리 과정에서 사용자와 상호작용하는 대화형 태스크의 페이지들이 다른 페이지들과 식별되지 않기 때문에 페이지 캐시 자원을 유리하게 할당해줄 수 없기 때문이다. 결과적으로 사용자 입력을 처리하는 과정 도중에 대화형 프로세스의 페이지들이 다른 페이지들에 의해 페이지 아웃 당하고 페이지 폴트를 발생시키며 긴 응답시간을 초래한다.

본 논문에서는 이 문제를 해결하기 위해 프레임워크 기반 선별적 응용 페이지 보호 기법을 제시한다. 제안된 기법은 대화형 태스크를 프레임워크 레벨에서 식별하고, 커널로 해당 정보를 전달한다. 그리고 커널은 프레임워크로부터 전달받은 대화형 태스크 정보를 이용하여 대화형 태스크의 페이지가 희생될 페이지로 선정되는 것을 방지한다. 이에 따라 대화형 태스크의 페이지 폴트 발생이 줄어들게 된다. 우리는 제안된 기법을 안드로이드 KitKat 4.0 기반과 리눅스 커널 3.4가 탑재된 넥서스5S 스마트폰에 구현하였다. 실험결과 기존 시스템 대비 페이지 폴트 횟수를 37%, 응답 시간 11%를 단축시킨다.

이 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 배경을 설명하며 리눅스 관련 페이지 관리 메커니즘을 분석하고 3장에서는 관련연구를 설명한다. 그리고 4장에서는 사용자 응답성 및 문제를 정의하고, 5장에서는 문제를 해결하기 위한 프레임워크 지원 선별적 페이지 보호기법을 설명한다. 그리고 6장에서 실험 및 결과를 보이고 7장에서 논문의 결론을 맺는다.

제 2 장 배경

제 2 장 배경에서는 리눅스에서 제공하는 페이지 캐시 관리 메커니즘을 설명한다. 페이지 캐시는 디스크 I/O 작업을 통해서 메모리에 적재된 페이지들 중에서 빈번하게 사용될 것으로 선별된 페이지를 유지하는 메모리 공간이다. 이를 통해 같은 페이지에 대한 디스크 I/O 작업 횟수를 줄여 메모리 접근 시간(memory access time)을 줄이는 것이다. 이 장의 나머지 절에서는 본문의 이해를 돕기 위해 리눅스에서의 페이지캐시 구조 및 관리 메커니즘에 관해 개괄적으로 설명하고 페이지 교체 메커니즘과 Lumpy 회수(reclamation) 메커니즘에 대해 상세히 기술한다.

2.1 리눅스에서의 페이지 캐시 관리 메커니즘 개관

이 절에서는 우선 페이지캐시의 데이터 구조를 설명하고 이를 관리하기 위한 리눅스의 동작 메커니즘을 설명한다. 그림 2는 리눅스에서의 페이지 교체 흐름을 나타낸다.

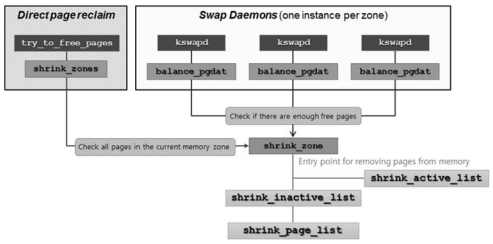

그림 2. 리눅스에서의 페이지 교체 흐름도.

페이지 캐시에 속한 페이지는 active_list와 inactive_list라는 두 개의 LRU 리스트들 중 하나에 속하게 된다. active_list에 속한 페이지들은 최근에 접근된 페이지를 포함하는 LRU리스트이며 페이지 회수 작업 시에 희생될 페이지로 선정되지 않고 메모리에 유지된다. inactive_list는 한동안 접근되지 않은 페이지를 포함하는 LRU리스트로 메모리 회수 작업의 대상 페이지들이 된다. 페이지의 참조 여부를 확인하기 위해 페이지 마다 참조 비트(reference bit)가 존재한다. 이 비트는 리눅스에서 struct page 구조체마다 PG_referenced로 구현되어 있다.

페이지캐시를 관리하기 위한 동작 메커니즘은 페이지 삽입과 제거 동작으로 구분된다. 페이지 삽입 동작은 요구 페이징과 선페이징에 의해 수행된다. 요구 페이징의 경우 mmap() 시스템 콜로 할당된 가상 메모리 공간에서 페이지들이 참조될 때 페이지 폴트 처리기를 통해 해당 페이지들이 페이지캐시에 삽입된다. 선페이징의 경우에는 명시적, 내재적 작업으로 구분된다. 명시적 작업은 유저 레벨 코드가 직접 시스템 콜을 호출하여 페이지를 삽입하는 것을 말한다. 리눅스에서 이러한 작업에 관련된 시스템 콜로는 read()와 MAP_POPULATE플래그를 인자로 받는 mmap()이 있다. 내재적 작업은 리눅스의 readahead 작업을 통해 수행된다.

페이지 제거 메커니즘은 페이지 교체 메커니즘과 Lumpy 회수 메커니즘의 연계로 수행된다. 페이지 교체 메커니즘은 페이지캐시에서 희생될 페이지를 선정하며, Lumpy 회수

메커니즘은 할당 요청된 페이지의 크기가 2개 이상일 때 그 희생될 페이지를 기준으로 연속된 페이지들을 추가로 희생될 페이지로 선정하여 메모리로 회수한다. 이러한 메커니즘들은 kswapd라는 리눅스 커널 데몬과 커널 메모리 할당자에 의해 수행된다. Kswapd는 주기적 또는 사건 기반(event-driven)으로 low watermark를 체크하여 페이지 교체 메커니즘을 수행한다. 커널 메모리 할당자는 사건 기반으로 min watermark를 체크하여 페이지 교체 메커니즘을 수행한다. 이러한 작업은 리눅스 커뮤니티에서 direct reclaim으로 불린다. Low와 min watermark들의 값은 리눅스에 미리 정의되어 있는 임계값이며 그 수치는 물리 메모리 크기에 따라 달라진다. 2GB의 시스템에서 그 값들은 각각 7,190KB, 5,752KB이다[5].

2.2 페이지 교체 메커니즘

리눅스에서 페이지 교체 메커니즘은 페이지 구분 작업과 희생될 페이지 선택 작업으로 나뉜다.

페이지 구분 작업에서는 페이지캐시에서 메모리에 유지될 페이지들이 active_list에 삽입되고, 그렇지 않은 페이지들은 inactive_list로 삽입된다. 그리고 각 페이지들의 참조 비트 변경 시점에 따라 페이지들이 다른 리스트로 이동된다. 그림 3은 각 리스트간의 교체 메커니즘에 의한 페이지 이동을 나타낸다. 구체적으로, 최초로 삽입된 페이지는 inactive_list에 존재하며 참조

비트는 0이다. 이후에 그 페이지가 참조되어 참조 비트가 1이 된 경우에 kswapd가 주기적으로 해당 페이지를 active_list로 이동함과 동시에 참조 비트를 0으로 설정한다. 그 페이지가 한 번 더 참조되면 1로 설정된다. 해당 비트는 LRU-2메커니즘을 주기적으로 수행하는 kswapd에 의해 0으로 set이 된다. 해당 페이지는 다음 kswapd의 주기에서 여전히 참조 비트가 0이면 inactive_list로 이동한다.

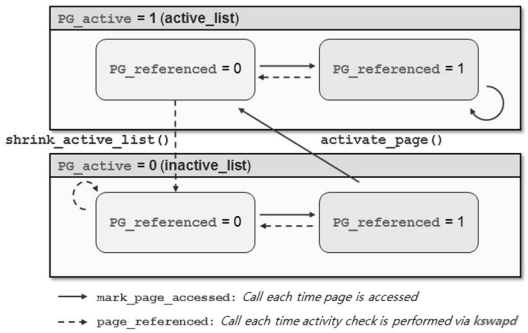

그림 3. LRU리스트 간의 페이지 플래그에 따른 페이지 이동.

희생될 페이지 선정 작업은 inactive_list의 페이지들에서 LRU-2 정책에 따라 여러 개의 희생될 페이지들을 선택한다. 선택된 페이지들의 개수는 시스템의 자유 페이지(free page)의 개수가 high watermark로 도달되기 위해 필요한 개수로 결정된다. high watermark들의 값은 리눅스에 미리 정의되어 있는 임계값이며 그 수치는 물리 메모리 크기에 따라 달라진다. 2GB의 시스템에서 그 값은 8,628KB이다[5].

2.3 Lumpy 회수 메커니즘

Lumpy 회수 메커니즘은 단편화된 물리 메모리에서 2개 이상의 연속된 페이지들을 확보하기 위한 메커니즘이다. 이 메커니즘은 커널 메모리 할당자가 2개 이상의 연속된 페이지 할당을 요청 받았을 때, 물리 메모리 부족으로 수행되는 페이지 교체 메커니즘에 연계되어 수행된다. 구체적으로, 이 메커니즘은 페이지 교체 메커니즘을 위해 inactive_list에서 희생될 페이지를 선택할 때, 해당 페이지를 기준으로 할당 요청된 크기만큼의 연속된 페이지들을 희생될 페이지로 선정하여 함께 페이지캐시에서 제거한다.

그림 4는 LRU리스트와 Lumpy 회수 시 선택되는 주변 페이지 영역을 나타낸다. 태그 페이지(tag page)는 희생될 페이지로 선정된 페이지이다. Lumpy 회수 메커니즘은 태그 페이지를 기준으로 할당 요청된 크기만큼의 주변 페이지들이 희생될 페이지로 선정할 수 있는지 체크한 후 희생될 페이지로 추가한다. 해당 페이지가 희생될 수 있는지 판단하는 기준은 페이지의 타입과 소속된 영역이다. 페이지 타입으로는 이동 불가능한 (unmovable), 회수 가능한 (reclaimable), 이동 가능한 (movable), 예약된(reserve)으로 총 4가지 타입이 있다. 이동 불가능하며 예약된 페이지들은 커널 영역에서 사용되는 페이지로 페이지캐시에서 제거되지 않는 페이지다. 이동 가능하며 회수 가능한 페이지 타입들은 사용자 영역에서 사용되는 페이지로 희생될 페이지에 포함될 수 있다.

Lumpy 회수 메커니즘은 태그 페이지를 중심으로 정해진 영역의 페이지가 메모리 영역을 벗어나지 않는지 확인하고, 특정 블록에 포함된 페이지는 단편화를 방지하기 위해 Lumpy 회수 대상에서 제외된다.

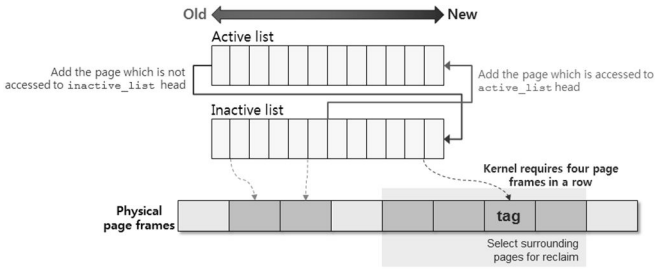

그림 4. LRU리스트와 Lumpy reclamation.

제 3 장 관련연구

제 3장에서는 사용자 응답성을 높이기 위한 관련 연구를 설명한다. 모바일 환경에서의 사용자 응답성을 높이기 위한 선행연구가 활발히 진행되어 왔다. 이러한 기존의 접근 방법들은 커널 레벨과 프레임워크 레벨로 나눌 수 있다.

3.1 커널 레벨

커널 레벨에서는 태스크들이 CPU, 메모리, I/O와 같은 한정된 시스템 자원을 사용하기 위해 경쟁하면서 지연이 발생하게 된다. 리눅스 커널에서는 이러한 지연을 줄이기 위해 각 시스템 자원마다 시스템 자원 할당을 위한 다양한 메커니즘을 채택하고 있다. 하지만 커널에서는 대화형 태스크를 고려하지 않기 때문에 사용자 응답성 측면에서 만족스럽지 못한 성능을 제공한다. 따라서 커널에서 대화형 태스크를 구분하고, 해당 태스크에게 우선적으로 시스템 자원을 할당하는 기법과 관련한 연구가 진행되었다.

CPU 자원 경쟁에 의한 지연의 경우 사용자 응답성을 높이기 위해 스케쥴링 알고리즘을 개선시키는 연구가 진행되어 왔다. 리눅스 기본 스케쥴링 알고리즘으로 채택하고 있는 CFS(Completely Fair Scheduler)는 각 태스크가 공정하게 CPU 자원을 할당 받을 수 있도록 각 태스크에게 가중치(weight)를 두어

그 가중치에 비례하는 CPU 시간을 할당한다. 그리고 해당 시간 동안 CPU 자원을 할당 받은 태스크는 다른 태스크에 의해서 선점 당하지 않도록 보장한다. 이를 통해 리눅스는 CPU 자원을 최대한 태스크들이 공정하게 할당 받을 수 있도록 한다. 하지만 CFS는 대화형 태스크를 식별하지 않기 때문에 사용자 입력 처리 중 할당 받은 CPU 자원을 모두 소진했을 경우 다른 태스크에 의해 CPU자원을 선점 당하여 긴 응답시간이 초래될 수 있다. 따라서 이를 해결하기 위해 프레임워크 지원을 받아 커널에서 대화형 태스크를 구별하고, 사용자 응답 처리 시 우선순위를 높여주어 CPU 자원을 빠르게 할당 받을 수 있도록 하는 기법이 제안되었다[6].

메모리 자원에서 페이지 폴트로 인한 I/O발생 횟수를 줄이기 위해서 리눅스는 readahead 메커니즘을 제공한다. 이 메커니즘을 이용하여 대화형 태스크의 빠른 수행을 위한 readahead 기법이 제안되었다[7]. 이 기법은 대화형 태스크를 커널에서 런타임으로 구분하여 해당 태스크가 페이지를 메모리로 적재 시, 다른 태스크보다 더 많은 페이지를 선페이징하여 대화형 태스크의 페이지 적중률을 높인다. 하지만 이 기법은 대화형 태스크를 찾기 위한 런타임 오버헤드가 발행하며, 페이지 교체 정책에서 대화형 태스크를 구분하지 않아 해당 페이지가 페이지 캐시에서 제거될 수 있다.

I/O자원의 경우 I/O스케쥴러에 의해 해당 태스크가 발생시키는 I/O의 특성에 따라 우선순위를 두어 요청을 처리한다. I/O의 특성은

태스크의 수행에 영향을 주는 동기적(Synchronous) I/O와 태스크 수행에 영향을 주지 않는 비동기적(Asynchronous) I/O로 구분된다. 따라서 I/O 수행 시 태스크 수행에 영향을 주는 동기적 I/O를 우선적으로 처리되어야 하기 때문에 비동기적 I/O보다 높은 우선순위가 부여된다. 하지만 커널에서 모바일 기기에서 발생시키는 I/O의 특성을 잘못 분류하여 동기적 I/O를 비동기적 I/O로 분류하는 경우가 발생한다. 이를 해결하기 위하여 동기적 I/O임에도 비동기적 I/O로 분류되는 경우를 파악하고 해당 비동기적 I/O로 분류된 I/O의 우선순위를 높여주어 응답 시간이 지연되는 문제를 해결하는 기법이 제안되었다[8]. 하지만 이 기법은 지연될 수 있는 상황을 방지하지만 대화형 태스크를 커널이 식별하지 않아 사용자 응답성을 보장하기에는 한계가 있다.

3.2 프레임워크 레벨

프레임워크 레벨에서도 빠른 사용자 응답성을 제공하기 위해 다양한 기법을 사용하였다. 대표적으로 안드로이드는 응용태스크를 빠르게 실행시켜 사용자 응답성을 개선시켰다0. Zygote 프로세스라는 프로세스를 생성하여 응용이 자주 사용하는 라이브러리 클래스들을 미리 초기화 한다. 그리고 각 응용들이 생성될 때 미리 초기화된 Zygote 프로세스를 통해 생성됨으로써 응용이 생성되는 시간을 단축한다.

안드로이드에서는 터치화면에서의 드로잉시간 단축을 위해 분리된 surface라는 분리된 구역으로 나누어 렌더링하는 작업을 수행한다[10]. 여러 개의 surface구역으로 나누어 갱신된 surface만 새롭게 랜더링한다. 안드로이드에서 제공하는 서비스인 SurfaceFlinger는 여러 개의 surface들을 조합하여 프레임버퍼장치로 보냄으로써 전체 랜더링 시간이 줄어들어 사용자에게 향상된 응답성을 제공한다.

프레임워크의 지원을 받아 대화형 태스크를 커널에서 구분하여 대화형 태스크의 우선순위를 조정함으로써 사용자 응답성을 개선하는 기법도 제안되었다[11]. 사용자 응답 처리 과정에서 태스크 경쟁에 의해 대화형 태스크의 수행이 지연될 수 있다. 따라서 대화형 태스크의 우선순위를 높여주어 사용자 응답 처리 중 CPU자원을 우선적으로 할당해줌으로써 사용자 응답성을 개선한다. 하지만 메모리에서 응답성 관련 태스크의 페이지를 보호하지 못하기 때문에 사용자 응답 처리 중 페이지폴트에 의해 지연되는 문제를 해결하지 못한다.

제 4 장 사용자 응답성 및 문제 정의

제 4장에서는 사용자 응답성을 정의하고 문제 상황에 대해서 분석하고 설명한다.

본 논문에서는 사용자 응답 처리 과정에서 발생하는 페이지 폴트로 인한 응답 시간 지연을 해결하고자 한다. 사용자 응답성은 사용자 입력 이벤트가 모바일 기기에 발생한 시점부터 화면에 결과가 표시되기까지 걸리는 총 소요시간으로 정의한다0. 사용자 입력을 처리하기 위해 이벤트가 응용 태스크에 전달되고 응용 태스크가 해당 이벤트에 따라 작업을 수행한다. 그리고 그 결과를 화면을 출력하는 태스크에 전달하여 사용자에게 표시한다. 임의의 사용자 입력 관련 응답시간 RT는 다음과 같다.

$$RT = T_E + n * T_{pf}$$

이 때 T_E 는 사용자 응답성 관련 태스크들의 수행 시간을 나타내고 n, T_{pf} 은 각각 응답 처리 과정 중 발생한 페이지 폴트 횟수와 페이지 폴트 발생시 소요되는 시간을 나타낸다.

사용자 입력 처리 페이지 폴트가 발생하면 사용자 응답 시간이 지연된다. 그림 5는 사용자 입력 이벤트가 모바일 기기에 발생한 시점부터 결과가 출력되기까지의 태스크 체인과 그 과정에서 발생하는 페이지 폴트를 나타낸다. 사용자 응답성 관련 태스크들은 각자의 작업을 수행 하면서 페이지 사용을 위한 메모리공간을

요구하게 된다. 이 때 요청한 페이지가 메모리에 부재한 경우 페이지 폴트가 발생하면서 해당 페이지를 메모리에 적재하기 위한 디스크 I/O가 발생하면서 응답 시간이 지연된다.

페이지 폴트 발생의 주된 원인은 기존 페이지 교체 기법이 대화형 태스크의 페이지와 다른 태스크의 페이지를 식별하지 않고 LRU-2 정책에 의해 희생될 페이지를 선정하기 때문이다. 따라서 사용자 응답 관련 태스크들의 사용자 입력 처리 중 다른 태스크의 페이지에 의해 페이지 폴트가 발생하며 사용자 응답 시간을 지연시킨다.

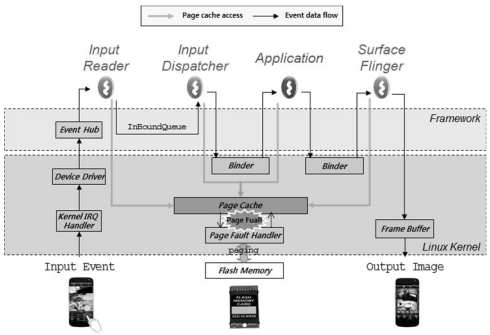

그림 5. 이벤트 전달 태스크 체인과 페이지 폴트 발생

제 5 장 프레임워크 지원 선별적 페이지 보호 기법

본 장에서는 앞서 언급했던 대화형 태스크의 사용자 입력 처리 중 페이지 폴트 발생 횟수를 줄이기 위한 프레임워크 지원 페이지 보호 메커니즘을 설명한다. 이 메커니즘은 프레임워크의 지원을 받아 커널 레벨에서 대화형 태스크들을 식별하고 그들의 페이지들을 페이지 교체 정책 및 Lumpy 회수 메커니즘에서 희생될 페이지로 선정되지 않도록 보호한다. 이를 위해 이 메커니즘은 대화형 응용 태스크 식별을 위한current interactive task pid identifier (CIPI), framework-assisted task identifier (FTI)와 페이지 식별 및 보호를 위한 interactive page protector (IPP), page identifier로 구성된다.

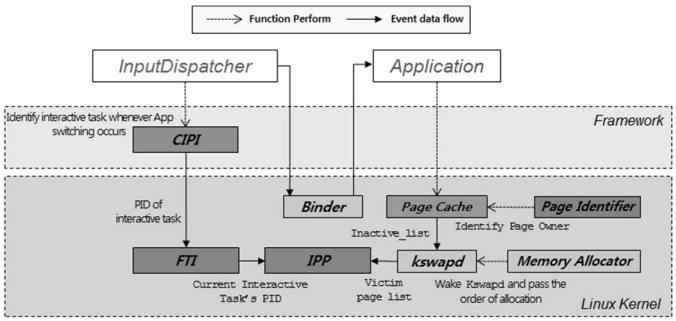

그림 6. 프레임워크 지원 선별적 페이지 보호 기법

그림 6은 각 모듈의 구성과 모듈간의 데이터 전달을 나타낸다. CIPI 모듈은 프레임워크레벨에서 대화형 태스크 PID를 FTI에게

전달하고, FTI는 전달받은 ID를 통해 커널에서 대화형 태스크를 식별한다. Page identifier는 페이지가 메모리에 적재될 때 이를 요청한 태스크의 ID를 해당 페이지 구조체에 기록한다. 마지막으로 IPP는 페이지 교체 에서 희생될 페이지를 선정할 때, 대화형 태스크의 페이지인지 판별하여 보호한다.

CIPI는 동적으로 변하는 대화형 태스크ID를 기록하고 커널에 해당 ID값을 커널의 FTI에게 전달한다. 구체적으로 CIPI는 InputDispatcher 로부터 사용자 입력을 처리하는 응용 태스크 ID를 전달받아 기록한다. InputDispatcher는 입력 이벤트(input event)가 발생할 때마다 CIPI에게 해당 대화형 태스크 ID를 전달한다. 그리고 CIPI는 InputDispatcher로부터 전달된 값과 기록되어있는 값을 비교하여 대화형 태스크 변경 여부를 파악하고 변경 시 전달받은 값을 기록한다. 그리고 변경된 값은 커널의 FTI에게 전달한다. 전달 방법은 Linux 커널에 시스템 콜을 추가하고 프레임워크레벨에서 호출하여 이것의 인자로 대화형 태스크 ID를 전달한다.

FTI는 해당 CIPI의 시스템 콜 호출로 전달된 대화형 태스크 ID를 기록하여, 커널에서 대화형 태스크를 식별한다. 커널에서 추가된 시스템 콜이 프레임워크 레벨에서 호출될 때마다 FTI는 대화형 태스크ID를 인자로 전달받는다. 그리고 FTI는 현재 기록된 대화형 태스크 ID와 전달받은 태스크 ID를 기록한다. 그리고 커널에 추가된 대화형 태스크 ID 요청 함수가 호출되면 현재 기록되어 있는 대화형 태스크 ID를 반환한다.

Page identifier는 메모리에 적재된 각각의 페이지들의 구조체에 이를 요청한 태스크의 ID를 기록한다. 리눅스에서는 메모리에 적재된 페이지를 관리하기 위해 페이지프레임마다 struct page 라는 페이지 구조체를 유지한다. 하지만 이 구조체로는 해당 페이지를 요청한 태스크를 구별할 수 없다. 따라서 이 구조체에 태스크 ID 멤버 변수를 추가하고 여기에 관련 태스크 ID를 기록한다.

IPP는 현재 FTI에 기록된 태스크 ID의 페이지가 희생될 페이지로 선정되지 않도록 보호한다. 기본적으로 IPP는 페이지 교체 메커니즘 수행 시 동작되며 추가로 Lumpy 회수 메커니즘이 수반되는 경우 추가 작업을 수행한다. IPP는 페이지 교체 메커니즘에서 inactive_list 내 희생될 페이지 선정 시, 해당 페이지의 구조체에 기록된 ID와 FTI에 기록된 ID를 비교하여 해당 페이지가 대화형 태스크의 페이지이면 희생될 페이지 선정에서 제외시킨다. 그림 1에서 inactive_list에서 희생될 페이지로 선정된 페이지들은 shrink_list()에서 페이지 타입 및 참조 비트를 검사하여 최종 희생될 페이지들을 선정한 후 해당 페이지들을 페이지캐시에서 제거한다. 따라서 대화형 태스크의 페이지를 보호하기 위해 구현된 함수를 shrink_list()에서 호출한다. 해당 함수는 FTI에서 제공하는 대화형 태스크 ID 제공함수를 호출하여 대화형 태스크 ID를 획득하여 해당 페이지를 판별한다. 그리고 shrink_list()에서 대화형 태스크의 페이지일 경우의 case문을 추가하여 해당 페이지가 메모리에 유지될 수 있도록 한다. 또,

대화형 태스크 페이지의 참조 비트를 1로 설정하여 active_list로 포함될 수 있도록 한다.

Lumpy 회수 메커니즘으로 선정된 희생될 페이지들에 관해서 IPP는 비대화형 태스크의 메모리 할당 요청 시 reclamation영역의 대화형 페이지의 존재 여부를 판단한다. 대화형 페이지가 존재할 경우 이 페이지를 제외한 연속된 페이지들을 희생될 페이지들로 선정한다. Lumpy 회수 메커니즘의 경우 태그 페이지를 중심으로 페이지를 검사하는데, 참조 비트를 고려하지 않는다. 따라서 이 메커니즘에서 페이지 검사를 하는 과정에서 대화형 페이지를 보호하기 위해 구현된 함수를 호출하여 대화형 태스크의 페이지를 식별한다. 그리고 Lumpy 회수 메커니즘에서 대화형 태스크의 페이지가 희생될 페이지로 식별된 경우의 희생될 페이지에서 제외된다.

시간이 지남에 따라 대화형 태스크와 관련된 페이지들이 IPP의 보호로 대부분의 페이지캐시 공간을 차지한다. 하지만 페이지 보호로 페이지캐시에 대화형 태스크의 페이지만 존재하게 된다면, 페이지캐시의 여유공간이 부족함에도 불구하고 페이지 회수를 할 수 없어 비대화형 태스크를 비롯한 대화형 태스크가 요청하는 페이지가 메모리에 적재되지 못하는 상황이 발생한다. 따라서 IPP는 비대화형 태스크의 페이지가 페이지캐시에서 모두 제거되고, watermark를 체크하여 여유 메모리공간이 부족하다고 판단될 경우 LRU-2 정책으로 대화형 태스크의 페이지를 제거한다.

제 6 장 실험 및 검증

본 장에서는 제안된 기법을 검증하기 위한 실험과 그 결과를 보인다. 우리는 제안된 기법을 안드로이드4.4 KitKat과 커널3.4가 탑재된 구글 넥서스 5 스마트폰 상에서 구현하고 I/O작업을 발생시키는 응용인 갤러리 응용의 응답시간과 페이지 폴트 횟수를 측정하였다. 응답시간 측정은 커널 레벨 프로파일링 도구인 KernelShark[12]를 사용하였고, 페이지 폴트 횟수 측정은 proc 파일 시스템[13]을 이용하여 측정하였다. 그리고 제안된 기법이 적용된 시스템의 전체적인 성능을 측정하기 위해 Antutu[14], PassMark[15] 벤치마크 응용을 사용하였다. 각 벤치마크는 CPU, RAM, 스토리지, I/O 등과 같은 성능을 측정하여 수치로 제공한다.

본 실험에서는 안드로이드에서 제공하는 기본 응용인 갤러리 응용과I/O 집약적 태스크를 동시에 실행하였다. I/O집약적 태스크는 지속적으로 I/O를 발생시키는 태스크로서 본 실험에서는 USB를 통해 스마트폰으로 대용량 파일을 전송하도록 요청하며 I/O를 발생시킨다.

그림 7은 기존 시스템과 제안된 기법인 framework-assisted selective memory protection(FMP)를 적용한 시스템에서 I/O 집약적 태스크를 수행시키기 전과 후의 페이지폴트 발생 횟수를 측정한 결과이다. 기존 시스템에서는 I/O집약적 태스크가 있는 경우 페이지폴트 수가 약 41% 증가한다. FMP기법을 적용시킨 시스템에서는 기존 시스템에서 I/O 집약적 태스크가 있는 경우에

발생하는 페이지 폴트 수보다 약 37% 감소하였다. 그림 8은 기존 시스템과 FMP가 적용된 시스템에서의 응답시간 측정 결과를 나타낸다. I/O집약적인 태스크가 수행중인 경우 기존 시스템에 비해 FMP가 적용된 시스템에서는 응답시간이 약 11% 단축되었다.

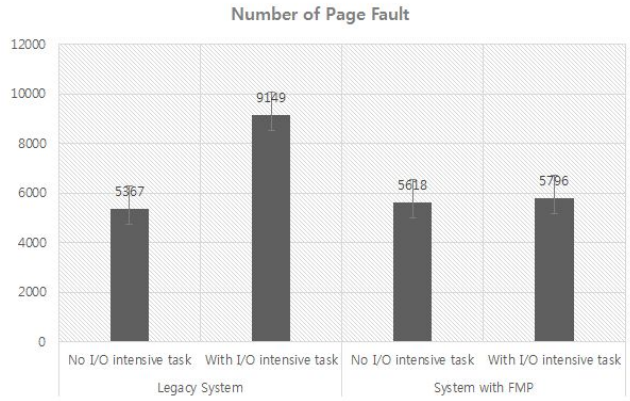

그림 7. 기존 시스템과 FMP기법을 적용한 페이지 폴트 수 측정 결과

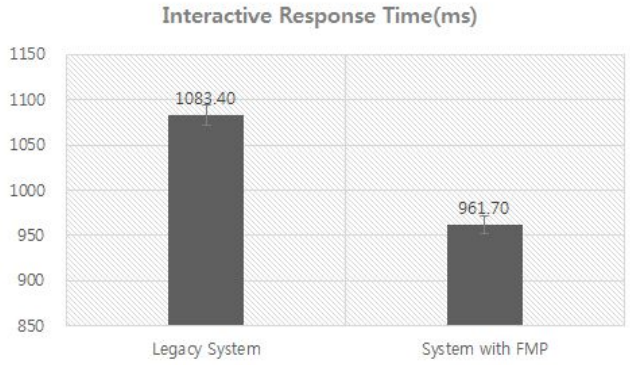

그림 8. 기존 시스템과 FMP기법을 적용한 응답시간 측정 결과

그림 9는 Antutu, PassMark 벤치마크를 통해 기존 시스템과 제안된 기법이 적용된 시스템의 성능을 측정한 결과값이다.

I/O집약적 태스크가 없는 경우 성능 오버헤드는 각 시스템이 비슷한 성능을 나타낸다. I/O집약적 태스크가 수행될 경우에는 FMP기법이 적용된 시스템이 기존 시스템보다 3% 정도의 오버헤드를 가진다.

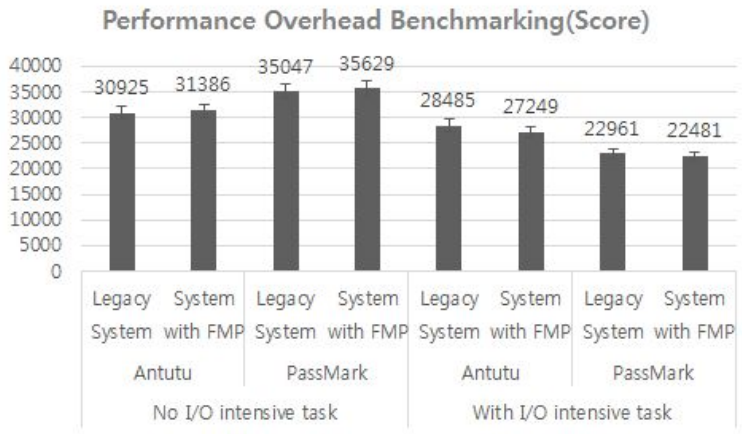

그림 9. 벤치마크를 통한 성능 오버헤드 측정결과

제 7 장 결론

이 논문은 모바일 환경에서의 사용자 응답성이 저조해지는 상황을 파악하고 이를 해결하기 위해 프레임워크 지원 선별적 페이지 보호 기법을 제안하였다. 이를 위해 먼저 사용자 응답성과 태스크 체인을 정의하고, 이 과정에서 페이지 폴트가 응답시간을 지연시킨다는 것을 밝혔다. 제안된 기법은 프레임워크에서 사용자 입력 처리 태스크를 식별하고 이를 커널로 전달하여 해당 대화형 페이지를 보호함으로써 사용자 입력 처리 중 페이지 폴트 발생을 최소화한다.

선별적 페이지 보호는 리눅스의 페이지캐시에서 채택하는 페이지 교체정책과 Lumpy 회수를 모두 고려하여 보호한다. 각 상황에서 회수 되기 위해 희생될 페이지로 선정되는 페이지 중에서 대화형 태스크의 페이지를 식별하고 희생될 페이지 선정에서 제외시켜 해당 태스크의 페이지를 보호한다. 실험 결과 제안된 기법이 기존 시스템 대비 페이지 폴트 횟수는 약 37%, 응답시간은 11% 감소하였다.

참고 문헌

[1] Klick Health,
“https://www.klick.com/health/news/blog/mhealth/mobile-apps-what-consumers-really-want”.

[2] BGR, “http://bgr.com/2013/09/20/iphone-android-touch-screen-responsiveness” (2013).

[3] W. Mauerer, “Professional Linux Kernel Architecture.” Wolfgang-Mauerer. (2008): 970.

[4] T. Johnson and D. Shasha, “2Q: A low overhead high performance buffer management replacement algorithm.” Proc. VLDB Conf. (1994): 297-306.

[5] LWN, https://lwn.net/Articles/422291/” (2011).

[6] S. Huh, J. Yoo, and S. Hong, "Cross-layer resource control and scheduling for improving interactivity in Android." Softw. Pract. Exper. (2014): 1549-1570.

[7] S. Bae, H. Song, C. Min, J. Kim, and Y. Eom, "EIMOS: enhancing interactivity in mobile operating systems." Proceeding of ICCSA. (2012): 238-247.

[8] D. Jeong, Y. Lee, and J. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," Proceedings of the 13th USENIX Conference. (2015): 191-202.

[9] D. Bornstein, “https://sites.google.com/site/io/dalvik-vm-

internals” (2008)

[10] Android Open Source

Project, “https://source.android.com/devices/graphics/”

[11] Y. Son, S. Huh, J. Yoo and S. Hong, "Framework-assisted Priority boosting for Improving Interactivity of Android Smartphones." Journal of KIISE. (2012): 380-386.
[12] eLinux,

“http://elinux.org/images/6/64/Elc2011_rostedt.pdf” (2011)

[13] Wikipedia, http://en.wikipedia.org/wiki/Procfs” (2010)

[14] Antutu, “http://www.antutu.com/en/index.shtml" (2015)

[15] PassMark software, “http://www.androidbenchmark.net/”

(2015)

Abstract

Framework-assisted Selective Page Protection for Improving Interactivity of Linux Based Mobile Devices

Seungjune Kim
Graduate School of Convergence Science and Technology
Seoul National University

While Linux-based mobile devices such as smartphones are increasingly used, they often exhibit poor response time. One of the factors that influence the user-perceived interactivity is the high page fault rate of interactive tasks. Pages owned by interactive tasks can be removed from the main memory due to the memory contention between interactive and background tasks. Since this increases the page fault rate of the interactive tasks, their executions tend to suffer from increased delays. This paper proposes a framework-assisted selective page protection mechanism for improving interactivity of Linux-based mobile devices. The framework-assisted selective page protection enables the run-time system to identify interactive tasks at the framework level and to deliver their IDs to the

kernel. As a result, the kernel can maintain the pages owned by the identified interactive tasks and avoid the occurrences of page faults. The experimental results demonstrate the selective page protection technique reduces response time up to 11% by reducing the page fault rate by 37%.

Keywords: Linux; Interactivity; Mobile devices; Page cache;

Student Number: 2014-24841