반응속도를 고려한 안드로이드 시스템 상에서의 메모리 관리 기법

모홍철○ 김대윤 낭종호
서강대학교 컴퓨터공학과
angel88317@nate.com uniyoun4@naver.com jhnang@sogang.ac.kr

Memory Management method on Android System for User Interaction

Hongchul Mo○ Daeyoun Kim Jongho Nang
Department of Computer Science and Engineering. The Sogang University

요 약

고성능, 고화질의 스마트폰 보급에 따라 어플리케이션 역시 고화질의 이미지를 포함하는 등 점점 중량화 되는 추세를 보이고 있다. 하지만 모바일 단말기의 특성 상 메모리 및 성능 제약 등 다양한 제약이 존재하는데, 따라서 어플리케이션 개발의 기반이 되는 단말기 운영체제 및 시스템에 대한 분석 및 고려가 효율적인 개발에 있어 무엇보다 중요하게 작용된다. 특히 자바를 기반으로 한 안드로이드 시스템의 경우, 비트맵 이미지 등 이미지 관련 자원들에 대한 메모리 사용에 제약이 크기 때문에 이러한 제약 사항 내에서 반응 속도 등 어플리케이션의 질을 높일 수 있는 효과적인 메모리 관리 및 활용 기법이 필요시 된다. 본 논문에서는 이를 위해 안드로이드 시스템 상의 메모리 제약과 관련 된 문제점을 분석하고, 이러한 문제점을 고려하여 어플리케이션 내에서 가장 효과적으로 이미지 관련 자원들을 처리하고 개발할 수 있는 메모리 관리 기법을 제시하였다. 이러한 기법을 통해 제한 된 메모리상에서 고화질의 이미지 자원 등을 효과적으로 다룰 수 있었고, 어플리케이션의 반응 속도 역시 효과적으로 높일 수 있었다.

1. 서 론

고성능, 고화질의 스마트폰 보급 및 이에 기반한 다양한 모바일 어플리케이션이 개발됨에 따라 어플리케이션 역시 고화질의 이미지 및 다양한 시스템 자원을 포함하는 등 점점 중량화 되어 가고 있다. 한편, 모바일 단말기의 경우, 성능 및 메모리 등에 큰 제약이 있기 때문에 이에 따라 모바일 단말기 및 해당 모바일 운영체제, 시스템을 고려한 최적화 된 개발이 필요시 된다. 특히 안드로이드 시스템은 비트맵 관련 이미지 및 자원들의 메모리 할당에 큰 제약이 작용하고 있는데, 이러한 제약은 안드로이드의 힙 메모리 구조에 바탕을 두고 있다. 안드로이드의 메모리 영역을 크게 두 가지로 나눌 수 있는데, 자바의 *allocation heap(VM heap)*과 C에서 사용하는 *native heap*이다. 문제는 안드로이드의 경우, 비트맵 객체들이 다른 자원들과는 달리 자바의 *VM heap*이 아닌 *native heap*에 할당되어 관리된다는 것이다.[1] 따라서 아무리 *VM heap* 메모리 사이즈가 크다 하더라도 native heap 메모리 사이즈가 부족할 경우, 다량의 비트맵 이미지를 사용할 수 없게 되며, 실제로 단말기에 따라 *native heap* 사이즈에 큰 제약이 존재한다.[2] 안드로이드 어플리케이션 개발은 자바 기반으로 이루어지기 때문에 일반적으로 개발자는 VM 기반으로 비트맵 객체들을 할당하여 사용하게 되는데, 실제로 비트맵 객체는 *VM heap*이 아닌 *native heap*에 할당이 되므로 비트맵 객체에 대한 메모리 관리가 추상화 되어 있어 개발자가 다루기 어려운 측면이 있다. 또한 자바의 경우, 가비지 컬렉션(*garbage collection*)을 통해 더 이상 참조되고 있지 않은 객체를 자동으로 메모리에서 해제하는 방식인데, *native heap*에 할당된 비트맵 객체의 경우, 이러한 가비지 컬렉션을 통해 자동으로 관리 및 해제가 되지 못하는 경우가 존재하여[2], 비트맵 객체의 메모리 릭이 발생할 수 있으며 메모리에 할당할 수 있는 이미지의 양에 한계가 있으므로 이 한계를 넘어설 경우, 이로 인한 *Out Of Memory* 문제가 발생하게 된다. (실제로, 가비지 컬렉션 기반임에도 불구하고, 비트맵 객체의 경우, 개발자가 임의로 객체를 해제할 수 있도록 recycle()이란 함수를 제공함) 특히 고화질의 이미지나 자원들을 많이 사용하게 될 경우 빈번히 발생 할 수 있는 문제이기 때문에, 사용할 수 있는 비트맵 관련 메모리에 제약이 생길 수밖에 없고 이에 따라 이미지에 대한 신속한 처리 및 반응 속도에도 문제가 생기게 된다. 따라서 *Out Of Memory* 문제를 야기하지 않고 제한 된 메모리 내에서 최대한 효율적으로 메모리를 최적화시켜 사용할 수 있는 방법을 본 논문에서 제안하고자 한다.

2. 안드로이드 메모리 관련 문제 및 처리 기법

위에서 언급했듯이 안드로이드의 경우, 비트맵 객체가 *native heap* 메모리에 할당되기 때문에 이로 인한 메모리 제약 및 메모리 릭 문제가 존재하게 되는데 이에 대한 세부적인 문제 및 처리 기법은 다음과 같다.

2.1 메모리 릭 문제

메모리 릭의 경우, 사용자 인터페이스를 구성하는 기본 단위인 액티비티(*Activity*)의 라이프사이클과 비트맵 객체의 라이프사이클이 다르기 때문에 비트맵 객체를 가지고 있던 해당 액티비티가 해제되었다 하더라도 객체가 해제되지 못하고 참조가 남아있어 가비지 컬렉션이 되지 못해 발생할 수 있다. 안드로이드의 액티비티는 <그림 1>의 1과 같은 구조의 참조를 하고 있는데, 액티비티의

*context*를 액티비티를 구성하는 뷰 그룹이 참조하고 있으며, 뷰 그룹에 속하는 뷰가 비트맵 객체를 가지고 있을 경우 뷰는 비트맵 객체를 참조하게 된다. 그런데, 이 비트맵 객체 역시 setCallback()이란 함수를 통해 뷰를 참조하게 되어 뷰와 비트맵 객체 사이에 쌍방 참조가 이루어지게 되고 이로 인해 해당 액티비티를 해제해도 가비지 컬렉터가 동작하지 않아 메모리 해제가 정상적으로 이루어지지 않고 메모리 릭이 발생하게 된다.
또한 <그림 1>의 2와 같이 액티비티 외부의 static 변수가 해당 액티비티를 참조하고 있는 상황이 생길 수 있는데, 참조하고 있는 변수가 살아있는 동안 액티비티 해제가 이루어질 수 없으므로 메모리 릭이 발생하게 된다. 따라서 위와 같은 문제를 해결하기 위해선 액티비티를 해제할 때 액티비티 내부의 모든 뷰들의 참조를 끊어주고 명시적으로 비트맵 객체를 해제해주어야 하며 또한 *context*가 자신의 범위 외에서는 사용되지 않도록 해야 한다. 이를 위해 액티비티의 *context* 대신 어플리케이션과 동일한 라이프사이클을 가지는 *application context*를 사용하는 방법이 존재한다.[2]

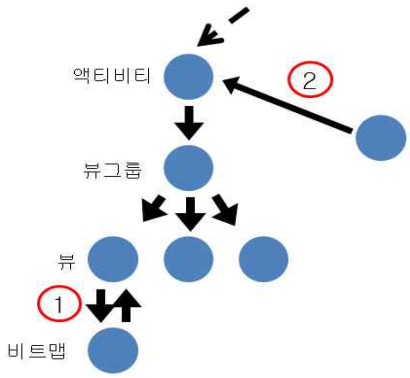

그림 1 안드로이드 액티비티 참조 형태

2.2 안드로이드 액티비티 및 뷰 구조

메모리 릭 외에도 액티비티 및 뷰 구조에 대한 고려가 필요한데, 안드로이드의 경우 하나의 액티비티에서 새로운 액티비티를 화면에 띄우는 경우 기존의 액티비티가 스택에 저장되는 루프구조를 가진다. 이 때 native heap에 할당되어 있는 비트맵 객체는 해제 될 수 없는 상태이다. 따라서 액티비티가 계속해서 스택에 쌓이게 될 경우, Out Of Memory 문제가 발생할 수 있으므로 액티비티를 계속해서 쌓이게 되는 경로가 존재해서는 안 된다. 일반적으로 상용 프로그램들을 이를 고려해 액티비티를 쌓을 수 있는 최대 깊이를 3~4 정도만 가능하도록 설계를 하고 있다.

3. 반응 속도를 고려한 메모리 관리 기법

2장에서와 같은 문제 및 처리 사항들을 고려하여 *Out Of Memory* 문제를 야기하지 않을 뿐 아니라 제한 된 메모리 내에서 효과적으로 비트맵 객체를 관리, 활용하고 어플리케이션의 반응 속도를 극대화 시킬 수 있는 방법을 제안하고자 한다.

3.1 UI 스레드와 비트맵 객체 할당 스레드의 기존 결합 형태

일반적으로 다량의 비트맵 객체를 어플리케이션에서 처리하기 위해선 화면 출력에 관련 된 UI 스레드와 화면에 출력할 실제 내용을 다운받거나 읽고 처리하는 비트맵 객체 할당 스레드가 존재해야 한다. 이 두 스레드를 어떻게 결합하여 사용하느냐에 따라 사용되는 메모리양 및 반응 속도가 달라지게 되는데 이에 대한 시나리오는 다음과 같다.

3.1.1 완전 비동기적 결합 형태

<그림 2>의 (a)와 같은 형태로, UI 스레드 및 비트맵 객체 할당 스레드를 완전히 비동기적으로 결합시킨 일반적인 구현 방법이다. 반응속도가 가장 좋지만 할당 된 비트맵 객체에 대한 참조를 유지시켜야 하므로 그만큼 필요한 메모리양도 커지게 되어 *Out Of Memory*가 발생할 수 있는 문제점이 있다.

3.1.2 동기적 결합 형태

<그림 2>의 (b)와 같은 형태로, UI 스레드 및 비트맵 객체 할당 스레드를 interleaved 방식으로 동기적으로 결합시킨 방법으로, on demand 형태로 비트맵 객체를 처리하며 사실 상 하나의 스레드가 UI 및 비트맵 객체 할당을 수행하는 것과 같은 형태이다. 필요시에만 비트맵 객체를 할당한 후 바로 해제하므로 메모리양이 적지만, 동기적으로 처리하기 때문에 반응속도가 크게 좋지 않은 문제점이 있다.

3.2 하이브리드 형태의 메모리 관리 기법

위의 두 가지 형태의 경우, 메모리의 양 또는 반응속도 측면에서 큰 문제점을 가지게 되는데, 따라서 이 두 가지 문제를 모두 최소화 시킬 수 있도록 위 두 가지 형태의 방법을 하이브리드 형태로 결합시킨 방법이 <그림 2>의 (c)와 같은 형태이다. 어플리케이션의 반응속도를 최적화 시킬 수 있도록 UI 스레드를 비트맵 객체 할당 스레드와 비동기적으로 동작시키도록 하되, 사용자가 실제 비트맵 이미지를 포함한 뷰를 보고자 플리킹을 멈춰 현재 뷰에서 머무를 때 단말기의 화면 가로 사이즈만큼을 출력시키는 데 필요한 비트맵 객체만을 비트맵 객체 할당 스레드가 일괄 처리 방식으로 다운 받고, 이 동안 UI 스레드에서 프로그레스 바를 화면에 띄워줌으로서 명시적으로 현재 상태를 알려주는 방식이다. 현재 보고 있던 뷰를 벗어날 경우, 더 이상 필요치 않은 비트맵 객체를 해제시키기 때문에 3.1.1에서 발생했던 메모리양에 대한 문제를 해결할 수 있고, UI 스레드가 비동기적으로 동작하는 상태에서 플리킹이 정지되었을 경우, 현재 단말기 사이즈만큼의 뷰를 출력하는 데 필요한 비트맵 객체만 on demand 방식으로 한꺼번에 다운 받아 처리하므로 3.1.2에서 발생한 반응속도 문제 역시 크게 줄였음을 알 수 있다.

3.3 지역성(locality)을 이용한 캐싱 기법

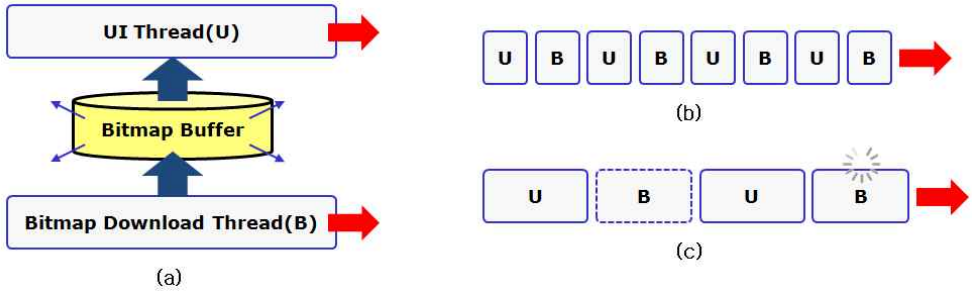

그림 2 스레드 간 결합 형태에 따른 비트맵 데이터 출력 방법

안드로이드의 경우, 비트맵 객체에 대한 메모리 제약이 있기 때문에 비트맵 객체를 계속해서 참조하여 가지고 있는 것이 문제가 될 수 있지만, 캐시와 같이 고정된 사이즈 형태로 일정 개수의 비트맵 객체만을 참조하는 방식이라면 캐시 사이즈 이상으로 비트맵 객체를 메모리에 할당하지 않기 때문에 <그림 3>와 같이 3.2의 방법과 지역성을 이용한 캐싱 기법을 결합하여 제한 된 메모리 내에서 반응속도를 더욱 향상 시킬 수 있다. 특히 지역성이 크게 작용하는 어플리케이션에서 큰 효과를 볼 수 있을 것이다.

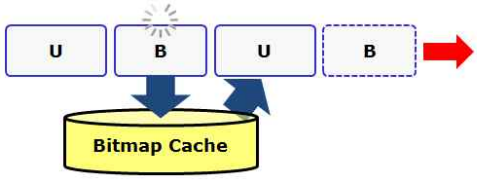

그림 3 지역성을 이용한 캐싱 기법

4. 실험 결과

위의 실험은 2.1에서 제시한 메모리 누수에 관한 실험이다. 아래의 <그림 4, 5>는 0.4초 간격으로 10개의 이미지 뷰를 가진 액티비티를 생성하고 해제를 반복하였을 때의 힙메모리의 사용량을 보여준다. <그림 4>에서는 명시적으로 뷰와 비트맵 객체의 참조를 끊어주지 않았으며, static 변수가 액티비티의 *context*를 참조하도록 설정하였다. <그림 5>에서는 명시적으로 쌍방참조를 끊어주었고, 외부에서 액티비티를 참조하지 않도록 하였다. 두 실험 모두 메모리 사용량이 증가하다가 가비지 컬렉션이 일어나면 메모리 사용량이 줄어드는 것을 확인할 수 있다. 하지만 <그림 4>에서는 조금씩 메모리 누수가 발생하는 현상을 살펴볼 수 있었다.

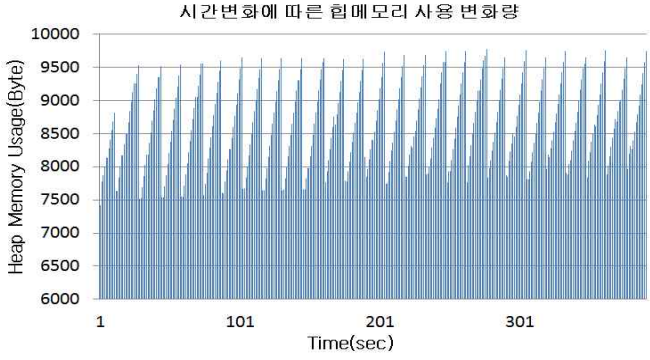

그림 4 참조가 유지된 경우의 힙메모리 사용 변화량

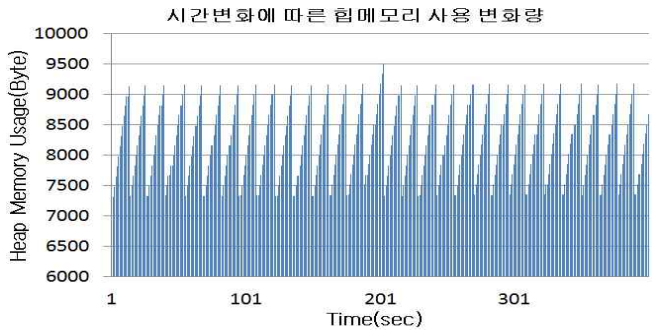

그림 5 참조를 끊었을 경우의 힙메모리 사용 변화량

5. 결론

위와 같이 기존 안드로이드 시스템의 경우, 비트맵 객체에 대한 힙 메모리 제약 및 가비지 컬렉션 정책에 일부 문제가 있기 때문에 *Out Of Memory* 문제 등 비트맵 객체 사용에 대한 제약이 크게 작용한다. 따라서 본 논문에서는 이러한 제약 내에서 메모리를 효율적으로 관리하고 반응속도를 높이기 위한 어플리케이션 개발 방법에 대해 제안하였다. 실제로 안드로이드 3.0 허니콤부터는 이러한 문제점을 인지하여 가비지 컬렉션 정책에 변화가 있었고, *native heap*에 할당하던 비트맵 객체를 *VM heap*에 할당되도록 수정하는 등 비트맵 객체에 대한 메모리 관리 정책에 큰 변화가 있었다.[1] 따라서 본 논문의 방법과 같이 안드로이드 시스템에 대한 분석 및 고려사항들을 바탕으로 메모리 릭 등의 문제가 야기되지 않도록 체계적이고 효율적인 설계 및 관리 기법이 앞으로도 필요시 될 것이다.

참고문헌

[1] P. Dubroy, "Memory Management for Android Apps," *Google I/O Development Conference*, 2011.
[2] R. Guy, "Avoiding Memory leaks," *Android Developers Blog*, http://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html, 2009.
[3] S. Brahler, "Analysis of the Android Architecture," *Karlsruher Institut fur Technologie*, 2010.
[4] J. Lessard, G. C. Kessler, "Android Forensics: Simplifying Cell Phone Examinations," *Small Scale Digital Device Forensics Journal*, Vol. 4, no. 1, 2010.