KossLab
KOREA OPEN SOURCE SOFTWARE DEVELOPERS LAB
공개SW개발자 Lab

빠르고 실패 없는 물리적 연속 메모리 공간 할당자

GCMA

Guaranteed Contiguous Memory Allocator

공개 SW 개발자 Lab 오픈소스프론티어 3 기 박성재

KossLab

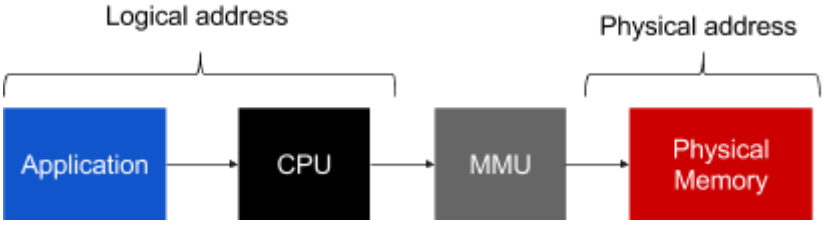
현대의 운영체제는 대부분 가상 메모리 방식의 메모리 관리를 채용하여 각각의 애플리케이션 프로세스가 무한대의 메모리를 모두 독점하고 있는 듯한 가상화를 제공하고 있다. 이에 따라 애플리케이션은 메모리 분절화(memory fragmentation) 문제에 대한 고려 없이 메모리를 쉽게 사용할 수 있게 되었다. 또한, 애플리케이션은 연속된 영역의 메모리가 필요하더라도 가상 메모리 공간에서 연속된 메모리이면 될 뿐 물리적으로 연속된 메모리를 할당받을 필요가 없어졌다. 우리가 많이 사용하는 리눅스 커널에서도 디맨드 페이징(demand paging)과 메모리 리클레임(memory reclaim) 방식을 사용한 가상 메모리 방식을 운용하고 있다. 그러나 DMA(Direct Memory Access)를 사용하지만 IOMMU나 Scatter / Gather DMA 기능이 사용 불가능한 기기들에서는 여전히 물리적으로 연속적인 메모리가 필요하다. 또한, 메모리 사용이 성능에 커다란 영향을 끼치는 경우에는 물리적 메모리 위치까지 고려한 메모리 할당이 필요하며 리눅스 커널의 huge page 같은 기능에서도 물리적으로 연속된 영역의 메모리 할당이 필요하다. 리눅스 커널은 이런 경우를 위해 CMA(Contiguous Memory Allocator)라는 메모리 할당자를 가지고 있다. 하지만 CMA는 그 설계 구조상의 한계로 인해 할당에 걸리는 시간이 길어질 수 있으며, 심지어는 메모리 할당에 실패할 수도 있다. GCMA(Guaranteed Contiguous Memory Allocator)는 CMA의 이러한 단점을 보완하기 위해 새롭게 설계, 개발된 메모리 할당자로 빠른 할당 속도와 할당의 성공을 보장한다. 본 글에서는 GCMA 의 필요성과 구조, 그리고 구현된 GCMA 프로토타입의 성능을 소개한다.

프로젝트명	GCMA (Guaranteed Contiguous Memory Allocator)
개요	리눅스의 CMA의 느린 속도와 메모리 할당 실패 가능성을 해결하기 위한 새로운 리눅스 커널의 물리적 연속 메모리 할당자 서브 시스템 개발
특징	시스템의 전체 메모리 영역 크기를 보존하고, 빠른 물리적 연속 영역 메모리 할당 속도, 물리적 연속 영역 메모리 할당의 성공 가능성을 보장
목표	임베디드 시스템과 같이 물리적 연속 영역 메모리 할당이 필요한 환경에서 고속의, 성공이 보장된 물리적 연속 영역 메모리 할당을 제공하면서 전체 시스템 성능을 유지
기대효과	보다 빠르고 실패 없는 물리적 연속 메모리 공간 할당자 제공
리퍼지토리	https://github.com/sjp38/linux.gcma

[목차]

1 물리적 연속 메모리 할당의 필요성

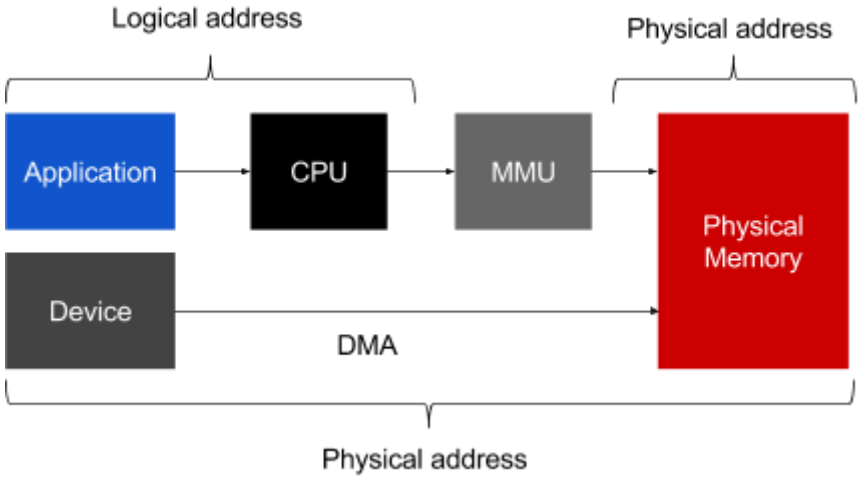
프로세스가 물리 어드레스 공간에 직접 접근하게 될 경우, 특정 프로세스에서 N byte의 메모리가 필요하다고 할 때 시스템의 사용 가능한 메모리 공간의 총합은 N byte 이상이 된다. 하지만 각 사용 가능한 메모리들이 서로 분절되어 있어 N byte가 넘는 크기의 연속된 덩어리가 존재하는 경우 메모리 할당에 실패하는 문제가 발생 가능한데, 이를 메모리 분절화(memory fragmentation) 문제라고 한다. 이 문제를 해결하기 위해 가상 메모리(Virtual Memory) 방식이 개발되었다. 이 방식에서 애플리케이션은 메모리에 논리 주소 영역을 통해 접근하게 되며, 이렇게 접근되는 논리 주소 영역의 각 영역은 실제 물리 메모리 주소 영역과 별개의 매핑 관계를 갖는다. 이로 인해 물리 메모리 주소상으로는 연속되지 않은 영역이라도 논리 주소 영역과의 매핑을 통해 애플리케이션은 해당 영역을 연속된 영역으로 인식하고, 메모리 분절화 문제없이 메모리 할당을 지속할 수 있게 된다. 이러한 논리 메모리 주소 영역과 물리 메모리 주소 영역 사이의 매핑을 CPU가 하게 될 경우 성능상의 오버헤드가 있을 수 있으므로, 오늘날의 컴퓨터는 대부분 MMU(Memory Management Unit)라는 별도의 하드웨어를 추가해 이 작업을 맡긴다.



[그림 1] 가상 메모리 구조

하지만 시스템에는 CPU 말고도 모니터, 프린터, 네트워크 카드, 카메라 등 많은 디바이스들이 연결되며, 이 디바이스들 역시 시스템의 메모리에 접근한다. 모든 디바이스의 메모리 접근을 CPU가 대신하게 될 경우에는 시스템의 성능이 떨어질 수 있다. 따라서 많은 시스템은 디바이스가 CPU의 도움 없이 직접 메모리에 접근할 수 있도록 해주는데, 이를 DMA(Direct Memory Access)라고 한다. 하지만 MMU는 CPU와 메모리 사이에 존재하기 때문에 DMA를 하는 디바이스의 경우 시스템의 메모리 상에 물리 메모리 주소를 사용해 접근할 수밖에 없다. 따라서 물리적으로 연속된 메모리가 있어야 하는 경우가 발생한다. 이외에도 메모리 성능상의 필요로 인해 물리적 메모리 주소에 기반을 둔 메모리 접근을 해야 하는 경우들이 존재한다. 여러 개의 CPU가 존재하며 CPU마다 특정 물리적 메모리

영역으로의 접근 속도와 다른 물리적 메모리 영역으로의 접근 속도가 다른 NUMA(Non-Uniform Memory Access) 구조의 시스템 등이 한 예다. 커널에서는 메모리 사용량이 많고 성능에 메모리 접근 속도가 중요한 경우, TLB 캐시 미스를 최소화하기 위해 Huge Page 기능을 제공하는데, 이 경우 역시 huge page를 확보하기 위해 연속된 물리 주소 영역이 필요하다.



[그림 2] 물리적 연속 메모리가 필요한 디바이스의 구조

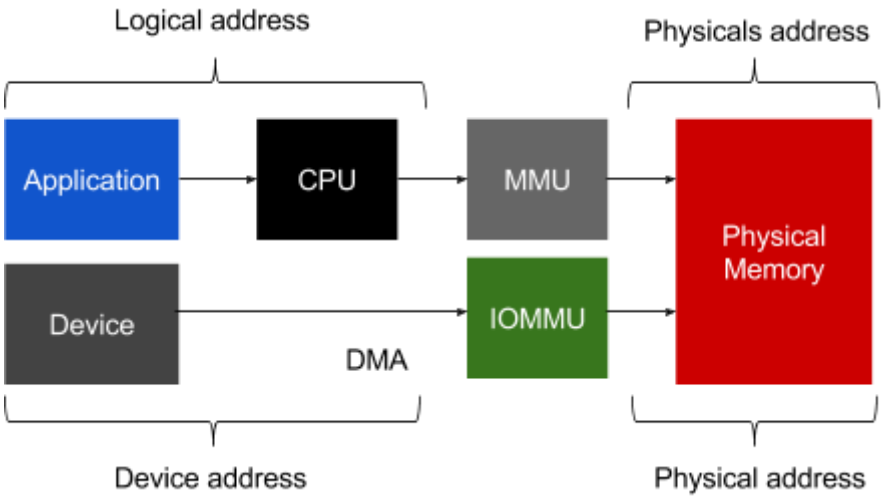
2 기존의 물리적 연속 메모리 할당 방법

이러한 경우를 위해 물리적으로 연속적인 메모리 영역을 할당하려는 방법이 여럿 존재하는데, 크게 하드웨어를 사용하는 방법과 소프트웨어적인 방법으로 나눌 수 있다.

2.1 하드웨어적 방법을 통한 물리적 연속 메모리 할당

앞서 살펴본 것과 같이 CPU를 사용하는 프로세스들이 물리적 연속 메모리 할당 문제를 겪지 않는 이유는 CPU와 메모리 사이에 MMU가 존재해서 CPU가 사용하는 논리적 메모리 주소 영역과 물리적 메모리 주소 영역 사이의 매핑을 대신해주기 때문이었다. DMA를 사용하는 디바이스들의 물리 메모리 영역 접근 문제를 해결해주는 첫 번째 방법은 MMU와 비슷한 하드웨어 장치 또는 기능을 제공하는 것이다. IOMMU는 DMA를 사용하는 디바이스와 메모리 사이에 위치해서 그러한 기능을 제공하는 추가적인 하드웨어다. IOMMU는 MMU가 CPU와 메모리 사이에서 해주는 역할을 DMA 사용 디바이스와 메모리 사이에서 해주며, 이로 인해 IOMMU를 갖춘 시스템에서는 디바이스에서의 물리적 연속 메모리 할당 문제가 존재하지 않는다. 또 다른 유사한 방식은 Scatter-Gather DMA 기능이다. 이 기능 역시 하드웨어적으로 제공되는 기능이며, 물리 메모리상에서 분절된 영역을 모아서 접근할 수 있게 해주는 방식으로 물리적 연속 메모리 할당 문제를 해결한다. 이런 하드웨어적 방식들은 별도의 하드웨어를 사용하기 때문에 소프트웨어적으로 수정이 필요 없고 성능 저하 문제가 거의 없다는 장점이 있다. 하지만 하드웨어적인 기능의 추가는 트랜지스터 추가가 필연적이며 소프트웨어나

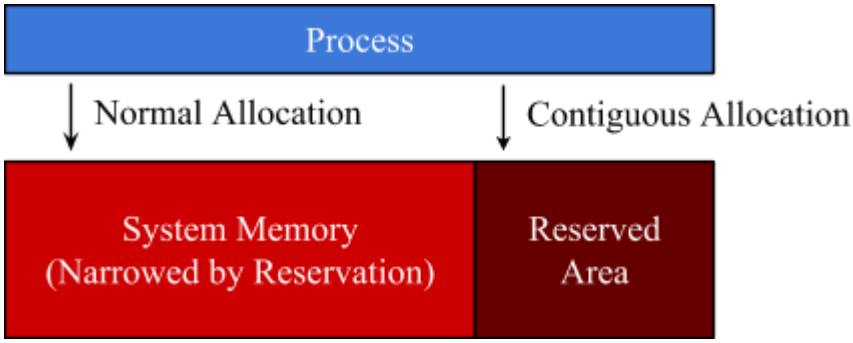
성능상의 비용이 필요하지 않은 대신 금전적 비용의 추가를 해야 한다. 고성능을 목표로 만들어진 비싼 시스템에서 이 정도의 추가적인 비용은 별것 아닐 수 있지만, 카메라를 장착한 임베디드 시스템과 같은 저비용의 시스템에서 이런 추가적 비용은 무시할 수 없을 만큼 크다. 또한, 이 방법은 DMA 사용의 경우에는 잘 동작하겠지만, 여전히 물리 메모리 영역의 접근이 있어야 하는 경우에 대해서는 해결이 되지 않는다.



[그림 3] IOMMU를 사용하는 디바이스의 DMA 구조

2.2 Reserved Area Technique 을 통한 물리적 연속 메모리 할당

소프트웨어적으로 이 문제를 해결하는 가장 간단하고 널리 사용되는 방법은 연속적 메모리 할당을 위한 전용 메모리 영역의 운영이다. 운영체제는 최초 시스템 시작 시에 물리 메모리의 일정 연속된 영역을 별도의 영역으로 표시해 놓고, 물리적으로 연속된 영역의 메모리 할당 요청이 들어오면 이 영역에서만 메모리를 할당해준다. 만약 물리적으로 연속된 영역의 메모리 할당이 아니라 평범한 메모리 할당 요청이 들어올 경우에는 해당 영역을 제외한 물리 메모리 영역에서 메모리를 할당해준다. 이를 Reserved Area Technique이라고 한다. 본 글에서는 이후 RAT로 부르도록 하겠다. 이 방식에서 연속된 영역 메모리 할당을 위해 전용되는 영역(Reserved Area)의 크기는 시스템 관리자에 의해 시스템의 시작 시 운영체제에 전달되는데, 이를 실제 시스템에 필요한 연속된 물리 영역의 크기에 맞춰 올바르게 설정할 필요가 있다. 해당 영역의 크기를 너무 크게 잡으면 평범한 메모리 할당을 처리하는 영역이 작아져 평범한 프로세스가 사용할 수 있는 물리적 메모리 영역이 좁아지고, 이로 인해 해당 프로세스의 성능이 떨어질 수 있기 때문이다. 해당 영역을 적당하게 잡는다 하더라도 시스템의 다른 프로세스들이 사용할 수 있는 메모리 영역이 줄어들기 때문에 기본적으로 시스템 전체 성능이 떨어지는 것은 피할 수 없다.



[그림 4] Reserved Area Technique을 사용한 연속 메모리 할당

2.3 CMA: Contiguous Memory Allocator

리눅스 커널에서는 이 문제를 해결하기 위해 CMA(Contiguous Memory Allocator)라고 하는 메모리 할당자 서브 시스템을 갖추고 있다. 이 방식은 RAT 방식에 기반을 두고 추가적인 방법으로 그 한계를 최소화시킨다. CMA는 RAT와 같이 시스템의 시작 시에 메모리의 특정한 영역을 표시해 두고 해당 영역에서 물리적으로 연속적인 메모리 할당에 사용하는 식으로 물리적 연속 메모리 할당을 처리한다. RAT와의 차이점은 해당 영역에 평범한(물리적 연속성을 신경 쓰지 않는) 메모리 할당을 가능하게 한다는 점이다. 대신 CMA는 Demanded Paging 방식과 비슷하게 추후 물리적 연속 메모리 영역의 할당을 하려 하지만 물리적 연속 메모리 할당을 위해 이번에 사용해야 하는 영역에 물리적 연속성을 신경 쓰지 않는 메모리 할당이 이루어져 있다면, 해당 영역의 데이터를 Reserved Area 바깥의 메모리로 옮겨주고 논리적 메모리 주소와 물리적 메모리 주소 사이의 매핑을 수정해 해당 데이터를 가리키는 논리적 메모리 주소가 여전히 유효하게 만들어 준다. 이제 해당 물리적 메모리 영역은 다른 용도로 할당하는 것이 가능하므로 해당 영역을 물리적 연속 메모리 할당 요청한 프로세스에 넘겨주게 된다. 한마디로 요약하면, CMA는 RAT에 기반을 두고 Reserved Area 상에 물리적 연속 메모리 할당 요청과 물리적 연속성과 관계없는 메모리 할당 요청이라는 서로 다른 우선순위의 메모리 할당을 가능하게 하며, 우선순위가 떨어지는(물리적 연속성과 관계없는) 메모리 할당으로 인한 메모리를 필요하면 Reserved Area 바깥으로 쫓아내는 방식이다. 이러한 CMA의 설계는 단순하고 깔끔하게 동작할 것 같지만, 실제 운영상에서는 여러 가지 문제가 발생할 수 있다.

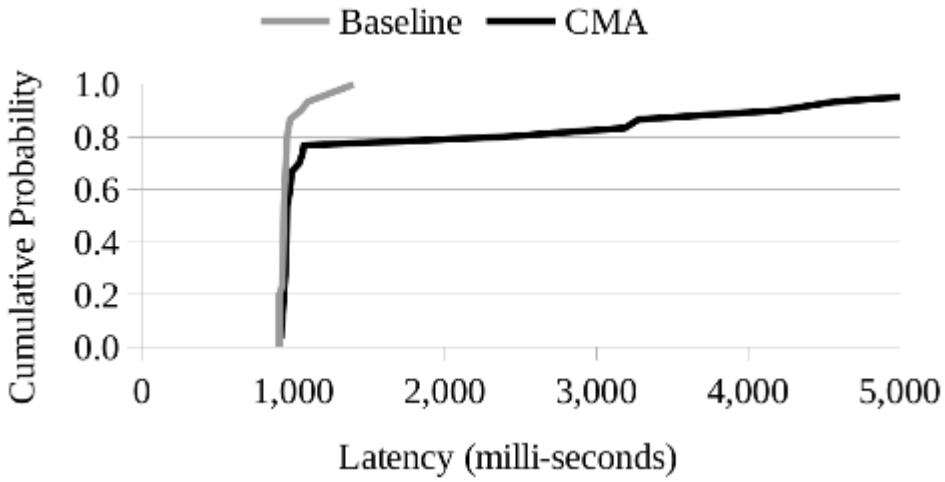
2.4 CMA의 한계

CMA에는 몇 가지 한계가 존재하는데, 메모리 할당에 걸리는 시간이 길 수 있다는 점이 첫 번째 한계이고, 심지어 메모리 할당이 실패할 수도 있다는 점이 그 두 번째 한계다.

연속적 메모리 할당의 과정에서 필요한 메모리 영역이 물리적 메모리 연속성과 관계없는 메모리 할당으로 사용되고 있다면 해당 영역의 데이터를 Reserved Area 바깥으로 옮기고, 해당 영역으로의 논리주소와 물리주소 사이의 매핑을 뒤바꿔서 기존의 논리주소가 데이터가 옮겨간 영역을 가리키게 한다

고 설명했다. 설명에서 유추할 수 있겠지만, 이는 매우 복잡한 동작이다. 데이터를 옮길 Reserved Area 바깥의 사용 가능한 메모리 영역을 찾아내야 하는데 이 과정에서 많은 시간이 필요할 수 있다. 데이터를 Reserved Area에서 바깥의, 간신히 찾은 사용 가능한 영역에 복사해야 하는데 복사해야 할 데이터의 크기에 따라서 이 복사 작업 역시 긴 시간이 필요할 수 있다. 논리 주소와 물리 주소 사이의 매핑은 리눅스 가상 메모리 시스템의 핵심적인 부분이고, 이 부분을 버그 없이 완벽하게 수행하기 위해서는 많은 자료 구조를 수정하고 많은 일을 행해야 하는데 이 역시 많은 시간을 소모하게 될 수 있다. 그뿐만 아니라 Reserved Area에서 바깥으로 몰아내야 하는 데이터를 현재 누군가가 사용하고 있다면 그 사용이 마무리될 때까지 기다려야 한다. 이는 데이터를 사용하는 코드의 구성에 따라 그 시간이 무한정 길어질 수 있으며, 이로 인해 메모리 할당이 아예 실패할 수조차 있다.

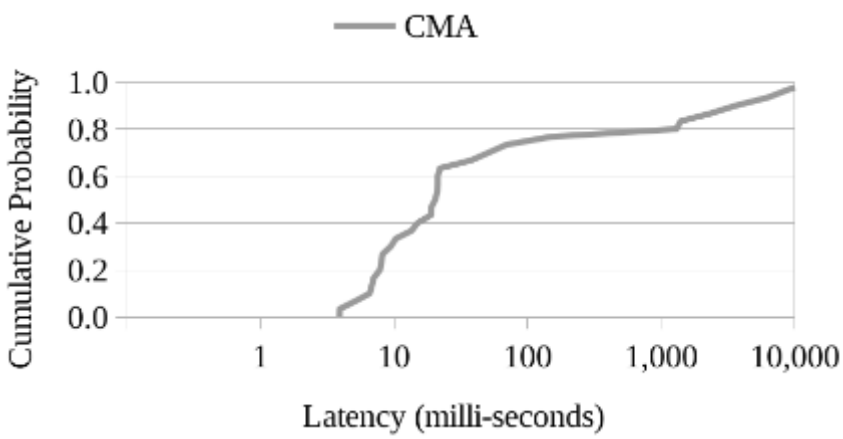
아래의 그림에서는 라즈베리파이2에서 blogbench라고 하는 워크로드가 돌아가는 가운데, 라즈베리파이의 기본 카메라 애플리케이션을 사용해 사진을 한 장 찍을 때 걸리는 시간의 분포를 누적 분포를 통해 표현하고 있다. 라즈베리파이2의 기본 설정은 RAT를 사용하게 되어 있는데 Baseline으로 표시된 선이 이 구성을 의미한다. 라즈베리파이2의 설정을 수정하면 RAT 대신 CMA를 사용하도록 할 수 있는데, 이 구성 하의 결과가 CMA로 표시된 선이다. 그림에서 볼 수 있듯 RAT를 사용하는 경우 모든 사진 촬영이 약 1.5초 이내에 마무리된 반면에, CMA를 사용하면 약 10%의 사진 촬영은 5초 이상의 시간이 걸린 것을 알 수 있다. 당신이 산책하러 나갔는데 귀여운 새끼 고양이를 마주쳐서 사진을 찍으려 했다고 가정해보자. 5초는 고양이가 도망가기에 충분한 시간이다.



[그림 5] 라즈베리파이 카메라의 기본 설정과 CMA 설정 사이의 속도 차이

위의 구성 하에서 카메라 애플리케이션이 요청한 물리적 연속 메모리 할당을 처리하기 위해 CMA가 사용한 시간의 누적 분포가 아래 그림에 있다. CMA의 메모리 할당 누적 분포는 위의 카메라 성능과 비슷한 그림을 보이는데, 이를 통해 CMA 구성 하에서 카메라 애플리케이션의 속도가 떨어졌던 원인은 CMA의 메모리 할당 속도 때문이었음을 알 수 있다. 고양이 사진을 못 찍게 된 원인을 찾은 것이다. 실

제로 라즈베리파이는 이러한 한계로 인해 기본 설정으로 RAT를 사용하며, CMA는 공식적으로는 지원하지 않고 있다.



[그림 6] CMA의 메모리 할당 속도

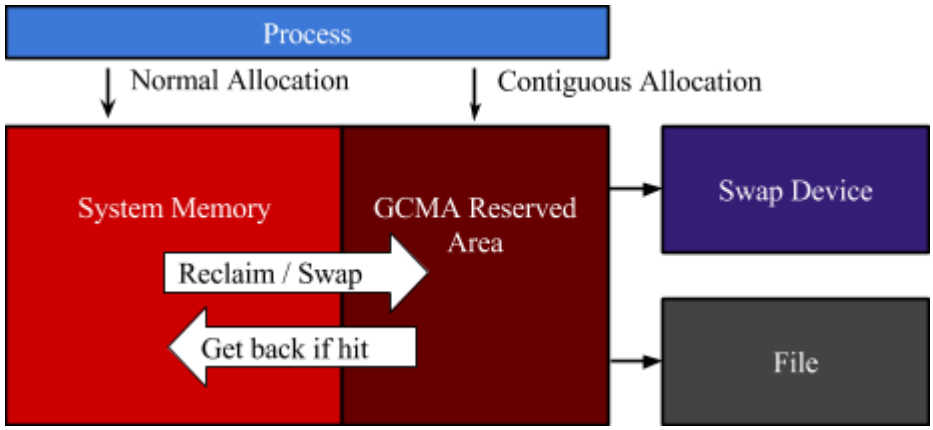
3 GCMA: Guaranteed Contiguous Memory Allocator

GCMA 는 앞서 설명한 물리적 연속 영역 할당 문제를 해결하면서 CMA의 한계점들도 해결하기 위해 만들어진, 리눅스를 위한 새로운 물리적 연속 메모리 할당자 서브 시스템이다. GCMA는 RAT의 물리적 연속성과 관계없는 메모리 할당 영역을 줄임으로써 시스템 전체 성능을 떨어뜨리는 문제가 없게 하면서도 CMA처럼 메모리 할당의 속도가 지나치게 길어질 수 있거나 심지어는 실패할 수도 있는 문제를 해결하는 방식으로 설계되었다. 이름의 Guaranteed는 합리적인 시스템 성능, 빠른 연속적 메모리 영역 할당 속도, 그리고 연속적 메모리 영역 할당의 성공을 보장한다는 뜻이다.

GCMA의 설계를 이해하기 위해서는 CMA가 가진 문제의 원인을 다시 한 번 짚어볼 필요가 있다. CMA의 패인을 한마디로 짚어보자면 Reserved Area의 낮은 우선순위 고객(물리적 연속성 관계없이 할당된 메모리의 데이터)이 생각보다 친절하지 못하다는 점이었다. 자세하게 말하자면 Reserved Area의 낮은 우선순위 고객은 높은 우선순위 고객인 '물리적 연속성이 있어야 하는 메모리 할당'이 필요로 할 때 해당 영역을 가능한 빨리 내놓아 주어야 하는데, 이 작업에 지나치게 많은 시간을 사용했다. 또한, 해당 데이터를 다른 코드가 사용 중이라면 해당 코드가 해당 영역을 놓아주기 전까지는 높은 우선순위 고객이 기다려야만 했다. 즉, CMA가 활용한 낮은 우선순위 고객(물리적 연속성과 관계없는 메모리 할당)은 충분히 친절하지 못한 고객이었다.

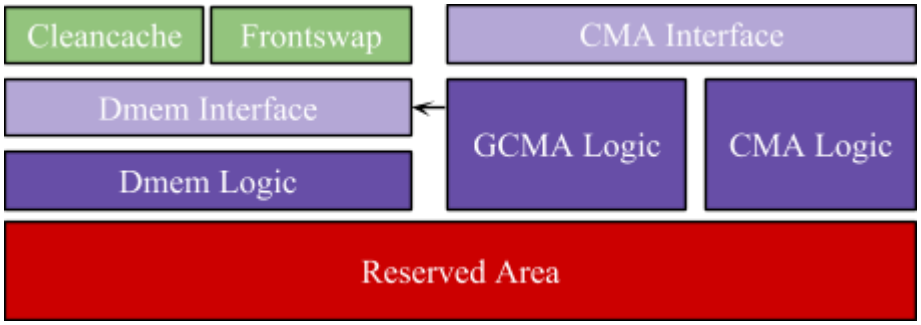
CMA가 RAT 방식에서 약간의 수정으로 이루어졌듯, GCMA는 CMA와 기본적인 구조는 공유하되 하나의 작지만 중요한 차이를 갖는다. GCMA는 기본적으로 CMA와 같이 Reserved Area를 사용하되 해당 영역을 서로 다른 우선순위의 고객에게 제공한다. 이로써 낮은 우선순위 고객에게도 넓은 메모리를 제공해 시스템의 성능을 유지하고, 높은 우선순위 고객인 물리적 연속 영역 메모리 할당을 가능하게 한다. 유일한 차이점은 낮은 우선순위 고객의 선정이다. CMA에서는 물리적 연속성과 관계 없는 메

모리 할당이 그 대상이었다. GCMA에서는 1) Reserved Area에서 내쫓는 게 간단하고, 2) 다른 곳에서 해당 영역에 대한 소유권을 주장할 수 없는 상대를 낮은 우선순위 고객으로 선정한다. Frontswap과 Clean-cache이다. 지면상의 한계로 인해 Frontswap과 Clean-cache에 대해서는 별도의 문서를 참고 바란다. Frontswap과 Clean-cache의 데이터는 이미 운영체제 커널의 관점에서 메모리로부터 추방되었지만, 향후의 기회를 위해 남겨져 있는 데이터이므로 언제고 다른 곳으로 옮겨지고 매핑되는 복잡한 과정을 가질 필요 없이 곧바로 삭제될 수 있다. 또한, 커널의 관점 상으로 메모리의 바깥에 존재하기 때문에 별다른 곳에서 해당 데이터를 동시에 사용하고 있는 경우가 존재할 수 없다.



[그림 7] GCMA의 구조

GCMA 구현상의 구조는 가능한 기존의 인터페이스를 재활용하는 방향으로 설계되었다. GCMA는 CMA와 같은 인터페이스를 사용하며, 시스템 관리자의 설정에 따라서 실제로 사용되는 할당자가 CMA와 GCMA 사이에서 선택될 수 있도록 구성되어 있다. GCMA의 Reserved Area는 Cleacache와 Frontswap에 의해 사용되는데, GCMA는 낮은 우선순위 고객을 Cleancache와 Frontswap으로 고정하기보다는 그와 같은 성격의 다른 고객도 지원할 수 있도록 DMEM(Discardable Memory)이라는 내부 서브시스템으로 Reserved Area를 외부에 노출한다. 따라서 어떤 서브시스템이든 DMEM 인터페이스를 사용해 GCMA의 Reserved Area를 활용할 수 있다.



[그림 8] GCMA의 모듈 구조

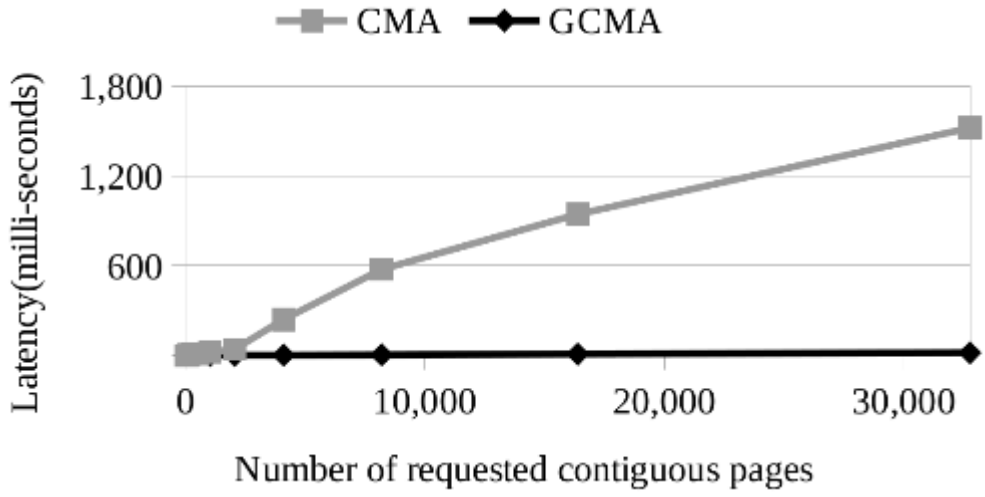
4 성능 평가

이렇게 개발된 GCMA 프로토타입의 성능 평가 결과를 공유한다.

성능 평가에 사용된 시스템은 라즈베리파이2다. 라즈베리파이2는 4개의 코어를 갖는 ARM Cortex-A7 프로세서인데, 이 프로세서의 속도는 900MHz다. 메모리는 1GiB의 LPDDR SDRAM을 사용한다. 저장 장치로는 Class 10 SanDisk 16 GiB micro SD card를 사용했다.

[그림 9] 성능 평가에 사용된 라즈베리파이2

아래 그림은 요청된 연속된 메모리 영역의 크기에 따른 메모리 할당에 걸리는 시간이다. 크기별로 30번씩 메모리 할당을 하고, 그 평균을 낸 값이다. 메모리 할당을 방해하기 위한 별다른 워크로드 없이 idle한 라즈베리파이2 위에서 메모리 할당 작업만을 수행했음에도, GCMA는 CMA보다 약 15배에서 130배까지 빠른 할당 시간을 보인다. 또한, 그림에는 표현되지 않았지만, CMA는 32,768개의 연속된 페이지로 구성된 메모리 영역의 할당을 요청받았을 때, 시스템의 메모리가 충분하고 방해하는 워크로드가 아예 존재하지 않음에도 30회의 중 한 번은 할당에 실패하기까지 했다.

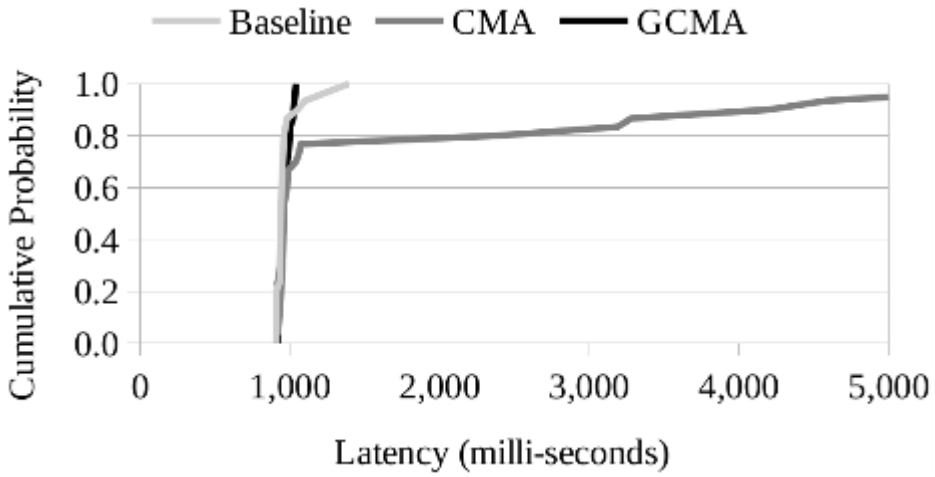


[그림 10] CMA와 GCMA의 메모리 할당 속도 비교

실제 현실적인 사용 시나리오에서의 효과를 알아보기 위해 라즈베리파이2 위에 blogbench 워크로드

를 돌리는 상태에서 동시에 라즈베리파이 기본 카메라 애플리케이션을 사용해 사진을 한 장 찍는데 걸리는 시간의 누적분포를 서로 다른 환경 구성상에서 비교해 본 결과를 아래 그림으로 표시했다. Baseline은 라즈베리파이2의 기본 구성으로, RAT를 사용한다. CMA는 라즈베리파이2의 구성을 변경, 리눅스에서 제공하는 CMA를 사용한다. 마지막으로 GCMA는 개발된 GCMA를 사용하도록 구성되었다. 앞에서 설명했듯 GCMA는 CMA와 같은 인터페이스를 사용하므로 애플리케이션의 코드는 수정되지 않았다.

그림에서 볼 수 있듯 CMA는 약 80%의 사진 촬영이 약 1초 만에 완료되었지만 약 10%는 1~5초, 나머지 약 10%는 5초가 넘는 시간이 걸렸다. 반면 RAT에 기반을 둔 Baseline과 GCMA의 경우, 약 1초 만에 모든 사진 촬영을 완료했으며 tail latency에서는 GCMA가 Baseline보다도 좋은 결과를 보였다. 이를 통해 GCMA는 CMA 대비 빠른 속도의 메모리 할당을 제공하며 그 영향은 실제와 가까운 상황에서 카메라 애플리케이션처럼 많은 사람이 사용하는 경우에 큰 차이를 가져올 수 있음을 알 수 있었다.



[그림 11] 백그라운드 워크로드 존재 하의 라즈베리파이 카메라 애플리케이션 속도 비교