리눅스 컨테이너를 위한 I/O 성능 Isolation 프레임워크 개발

Team: ★ X 5

201424474 손수호 ngeol564@gmail.com
201424452 박기태 gitae5504@gmail.com
201624409 권민재 kmnj951@gmail.com

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

School of Electrical and Computer Engineering, Computer Engineering Major

Pusan National University

2019년 6월 3일

지도교수: 안 성 용 (인)

목 차

1. 과제 배경 및 목적

1.1 과제 배경

최근 클라우드 서비스가 각광을 받으면서 떠오른 기술로 컨테이너 기술이 있다. 컨테이너는 가상머신(Virtual Machine)과 비교하여 가볍고, 빠르기 때문에 구글, IBM, 마이크로소프트와 같은 IT 업체에서 컨테이너에 많은 투자를 하고 있다. 즉, 많은 클라우드 서비스 제공자들은 컨테이너 형태로 사용자들에게 서비스를 제공하고 있다[1].

SLA(Service Level Agreement)는 서비스 제공자와 사용자와의 협약서이다. 이 협약서에는 서비스에 대한 측정지표와 목표 등이 나열되어 있다. SLA 는 제공되는 서비스와 기대하는 품질에 대한 모든 정보를 하나의 문서를 통해 공유함으로써 서비스 레벨에 대한 오해를 피할 수 있고 만약 서비스를 이용하다 문제가 생겼을 경우 누가 책임을 질 것인가에 관한 내용을 명시할 수 있다.

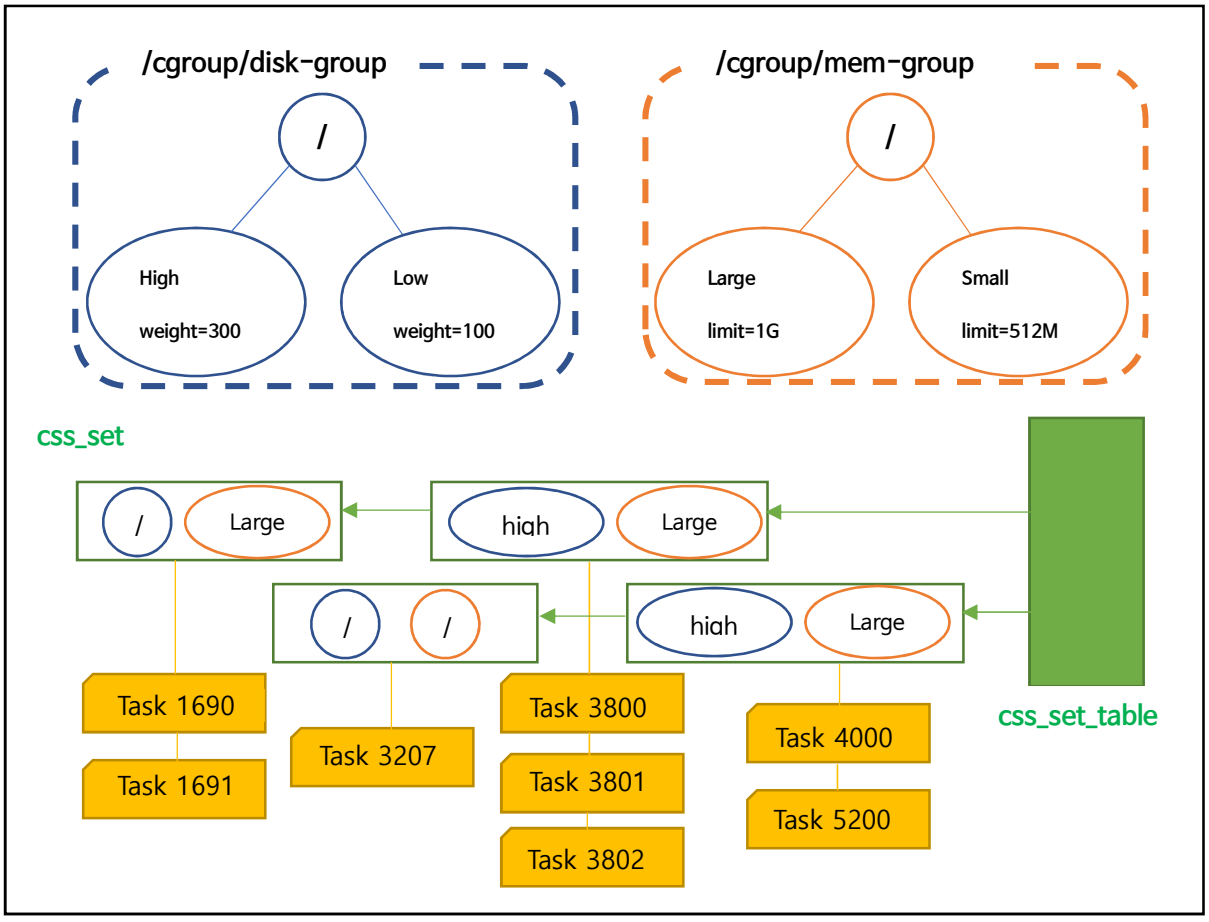
그룹	유형	vCPUs	메모리 (GiB)	네트워크 성능
General purpose	t2.nano	1	0.5	낮음에서 중간
General purpose	t2.micro 프리 티어 사용 가능	1	1	낮음에서 중간
General purpose	t2.small	1	2	낮음에서 중간
General purpose	t2.medium	2	4	낮음에서 중간
General purpose	t2.large	2	8	낮음에서 중간
General purpose	t2.xlarge	4	16	보통
General purpose	t2.2xlarge	8	32	보통

[그림 1. Amazon EC2 인스턴스 유형 선택 화면]

이러한 SLA 를 지키기 위해서는 성능 isolation 이 필요하다. 클라우드 기반 시스템에서는 많은 서비스 사용자가 존재하고, 이 서비스 사용자들이 서버가 제공하는 기능들을 사용하기 위해 많은 요청을 보낸다. Isolation 되지 않은 클라우드 시스템에서는 다른 서비스 사용자들이 서버의 자원을 자유롭게 사용할 수 있기 때문에, 만약 어떤 서비스 사용자가 과도한 자원을 사용하려 한다면 다른 서비스 사용자들의 성능이 나빠질 수 있고, 결과적으로 SLA 를 이행할 수 없다. 따라서, 성능 isolation 을 통해 서비스 사용자들이 SLA 를 통해 부여받은 리소스만을 사용하도록 할 필요가 있다.

[1] 클라우드 가상화 기술의 변화 – 소프트웨어정책연구소(2018)

[그림 1]은 아마존 웹 서비스(AWS)에서 새로운 EC2 인스턴스를 생성하는 그림이다. CPU 와 메모리 그리고 네트워크 성능 그룹별로 나누어 클라우드 서비스 사용자들은 이를 옵션별로 선택할 수 있다.

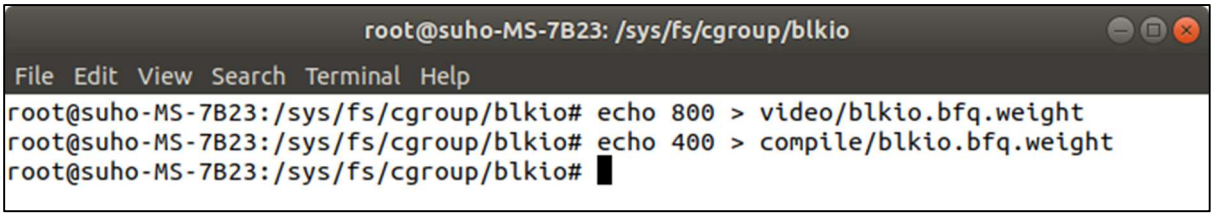


[그림 2. cgroup 파일 시스템 예시]

리눅스 컨테이너에서는 cgroup 을 이용해 컨테이너들에게 자원을 할당하고 관리한다. 커널 v2.6.24 에 처음 merge 된 cgroup(control group)은 시스템 상에서 동작 중인 태스크들을 그룹으로 만들어 제어할 수 있도록 도와주는 기능이다. cgroup 은 계층화 그룹으로 관리하며, 파일시스템으로 마운트 한 뒤 사용한다. cgroup 자체만으로는 동작 중인 태스크의 그룹만 만들어주며, 리소스(CPU, memory, disk, network)의 분배는 net_cls, ns, cpuacct, devices, freezer 와 같은 서브시스템(컨트롤러)이 필요하다.

특히, cgroup 은 'blkio'라는 블록 디바이스를 위한 서브시스템을 지원하고 있고, 싱글-큐 블록 레이어(Single-Queue Block Layer)에서는 CFQ(Completely Fair Queueing) 스케줄러, 멀티-큐 블록 레이어(Multi-Queue Block Layer)에서는 BFQ(Budget Fair Queueing) 스케줄러가 weight 에 따라 I/O 성능을 Isolation 하기 위해 존재한다. 아래 [그림 3]은 blkio 서브시스템의

blkio.bfq.weight 가상 파일을 통해 video cgroup 의 weight 를 800, compile cgroup 의 weight 를 400 으로 할당하는 예시이다.



[그림 3. Blkio 를 통한 weight 변경 예시]

1.2 기존 문제점

[그림 4. BFQ 스케줄러와 카이버 스케줄러 LoC 비교 (Linux 5.2-rc2 기준)]

1.2.1 실험환경

2. CPU	Intel® Core™ i5-8400 CPU @ 2.80GHz (6-Core)
메모리	15G
NVMe SSD	Intel® SSD 760p Series (256 GB)
OS	Ubuntu 18.04, Linux Kernel 5.1.4

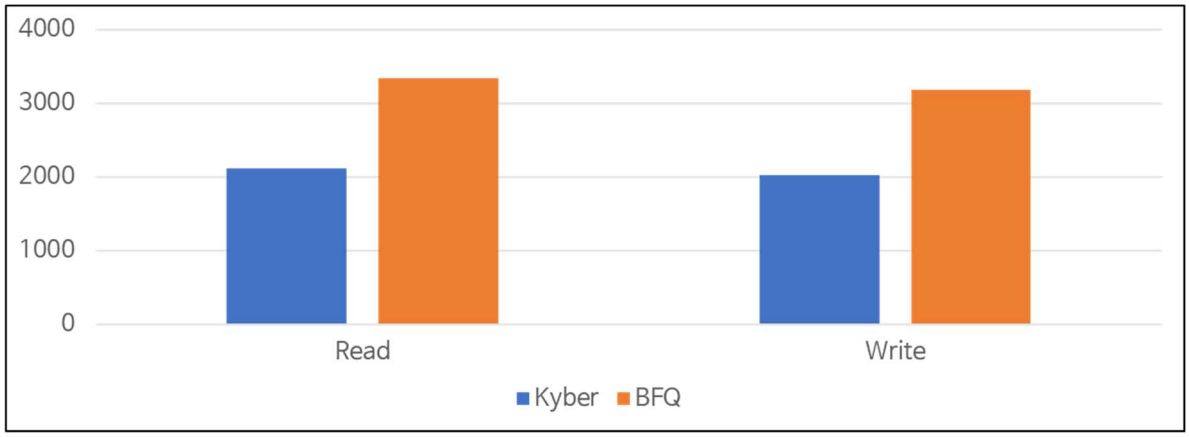
[표 1. 시스템 구성]

section	iodepth	size	rw	bs	cgroup
Compile-group	32	20G	randrw	4k	compile (weight=400)
Video-group					video (weight=800)

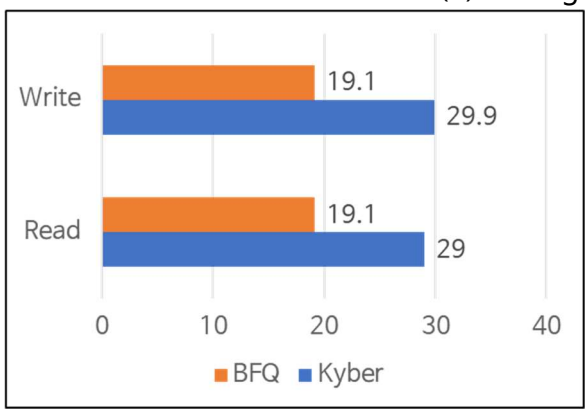
[표 2. fio 및 실험 환경]

실험 환경은 [표 1]과 같으며 fio 의 job 은 [표 2]와 같이 구성하였다.

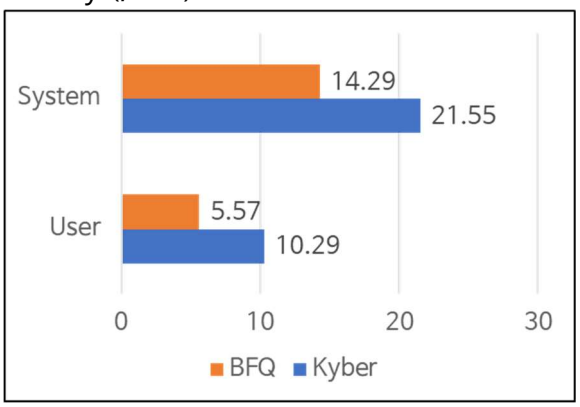
1.2.2 BFQ/카이버 스케줄러 성능평가



(a) Average Latency (μsec)



(a) Bandwidth (MiB/s)



(b) CPU Usage (%)

[그림 5. fio results]

성능 평가를 한 결과, 카이버 스케줄러가 BFQ 스케줄러보다 평균 latency 를 잘 보장하고 있는 것을 관찰할 수 있었다. Throughput 또한 카이버 스케줄러가 BFQ 스케줄러보다 우수하였으나, CPU 사용량 측면에서는 카이버 스케줄러가 BFQ 스케줄러보다 많이 사용하는 것을 확인하였다.

Video cgroup (weight=800)	Compile cgroup (weight=400)
READ: bw=19.1MiB/s (20.0MB/s) Write: bw=19.1MiB/s (19.0MB/s)	READ: bw=19.1MiB/s (20.0MB/s) Write: bw=19.1MiB/s (19.0MB/s)

[표 3. BFQ 스케줄러의 isolation 실험 결과]

그리고 BFQ 스케줄러의 문서에서는 CFQ 의 policy 를 이어 받았다고 서술하고 있기 때문에 weight 에 따른 bandwidth isolation 이 가능한 것으로 생각하였으나, 실제 확인한 결과 bandwidth isolation 이 일어나고 있지 않았다.

멀티-큐 블록 레이어에서 유일하게 I/O 성능 isolation 을 지원하려고 존재하는 BFQ 스케줄러도 아직까지 I/O 성능 isolation 이 지원되지 않고 있었다. 또한, BFQ 스케줄러는 카이버

스케줄러보다 latency 나 bandwidth 측면에서 좋지 않은 모습을 보여주고 있기 때문에, I/O 성능 isolation 이 지원되더라도 메인 스케줄러로 사용하기에는 무리가 있다.

1.3 과제 목표

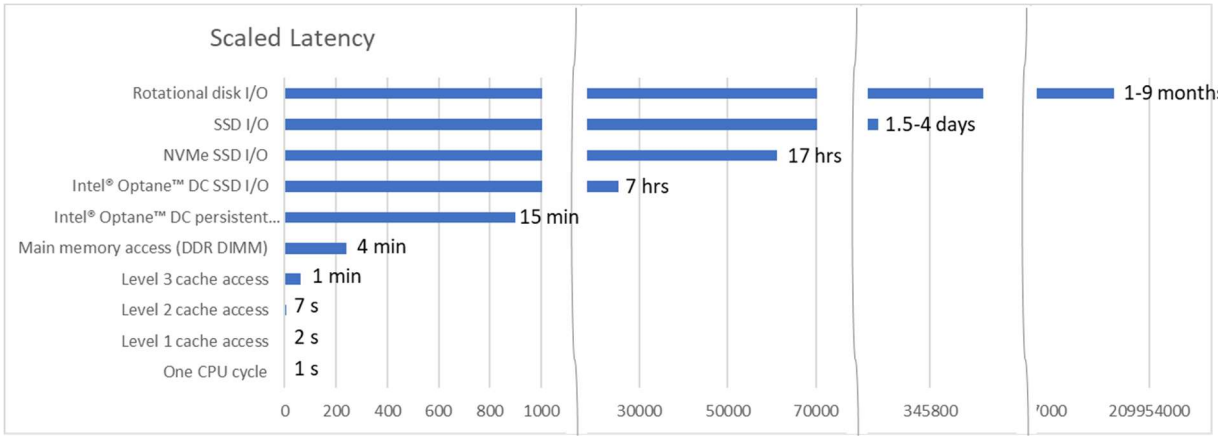
최근에는 고성능의 블록 디바이스들이 출시됨에 따라 싱글-큐 블록 레이어를 사용하는 것보다는 멀티-큐 블록 레이어를 많이 사용하고 있다. 하지만, 멀티-큐 블록 레이어에서 I/O 성능 isolation 을 지원하고자 하는 스케줄러는 BFQ 스케줄러뿐이며, 이 또한 아직 isolation 을 제대로 지원하지 못하고 있다. 또한, BFQ 스케줄러에 I/O 성능 isolation 이 구현된다 하더라도 I/O 성능이 좋지 않기 때문에, 클라우드 시스템은 쉽게 BFQ 스케줄러를 선택하지 못할 것이다. 이러한 상황에 클라우드 시스템 제공자들은 I/O 작업에 대한 SLA 를 이행하는데 문제가 발생하며, 클라우드 시스템 사용자들은 제대로 서비스를 제공받지 못한다.

따라서 본 팀은 멀티-큐 블록 레이어에 BFQ 스케줄러와 같이 proportional-share policy 를 가지며, 카이버 스케줄러처럼 성능도 고려할 수 있는 스케줄러 구현을 본 과제의 목표로 한다.

2 리눅스 커널 분석

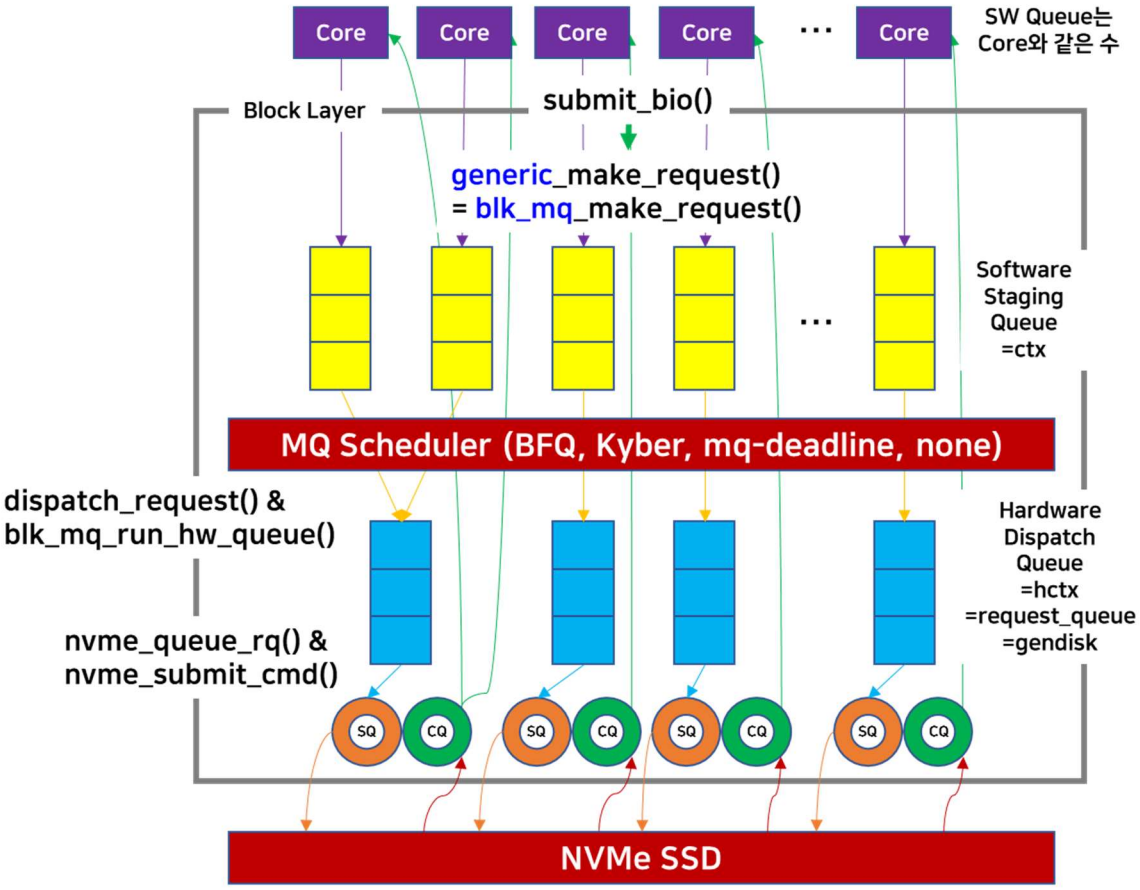
블록 I/O 스케줄러를 구현하기 위해선 멀티-큐 블록 레이어 분석이 필수적이기 때문에, 멀티-큐 블록 레이어에 대한 전체적인 분석을 실시하였다. 또한, 기존에 존재하는 멀티-큐 블록 스케줄러들 중 본 팀의 주제와 연관이 있는 BFQ 스케줄러와 카이버(Kyber) 스케줄러를 분석하여 과제 진행에 도움을 얻고자 하였다. 마지막으로, 최적의 스케줄러를 구현하기 위해 멀티-큐를 가지고 있는 NVMe SSD 가 사용하는 NVMe 리눅스 디바이스 드라이버를 분석해 보았다.

2.1 멀티-큐 블록 레이어



[그림 6. Computer Latency at a Human Scale]

블록 I/O 레이어는 블록 디바이스(하드디스크, SSD 등)의 입출력 작업들을 관리하는 리눅스 커널의 서브시스템이다. 캐릭터 디바이스(키보드 등)보다 복잡한 특성을 가지는 블록 디바이스는 관리하기가 쉽지 않다. 블록 I/O 레이어는 기본적인 I/O 작업 관리뿐만 아니라, 블록 디바이스에 접근하는 오버헤드가 크기 때문에 이를 줄이기 위한 여러 작업들이 필요하다. 따라서, 이런 복잡하고 다양한 작업들을 처리하기 위해서 블록 I/O 레이어는 크고 복잡할 수밖에 없다.



[그림 6. 리눅스 커널의 멀티-큐 블록 I/O hierarchy]

먼저, 블록 I/O 레이어는 리눅스 커널 내에서 매핑 레이어와 디바이스 드라이버 사이에 위치한다. 매핑 레이어는 파일시스템에서 요청된 논리적 주소를 파일 descriptor 등을 통해 물리적 주소로 바꿔주는 역할을 하며, 디바이스 드라이버는 실제 하드웨어 장치(여기서는 블록 디바이스)와 커널을 연결해주는 역할을 한다. 따라서, 블록 I/O 레이어는 매핑 레이어에서 받은 물리적 주소를 디바이스 드라이버에 전달하는 역할을 하게 되며 그 중간 과정에서 성능을 위한 추가적인 작업을 실시하게 된다.

매핑 레이어에서 submit_bio() 함수를 통해 블록 I/O 작업이 블록 I/O 레이어로 넘어오게 되고, 여기서부터 실제 블록 I/O 가 시작되게 된다. submit_bio() 함수를 통해 I/O 연산의 종류, 해당 연산에 대한 디스크 영역의 정보와 I/O 를 수행할 데이터를 저장하기 위한 메모리 영역의 정보 등을 포함하는 bio 구조체들을 전달받는다. 이때, 하나의 I/O 요청이 디스크 상에서 연속적이지 않은 여러 부분을 접근해야 한다면 bio 또한 여러 개가 생성될 것이다. bio 들은 linked-list 형태로 구현되어 있다.

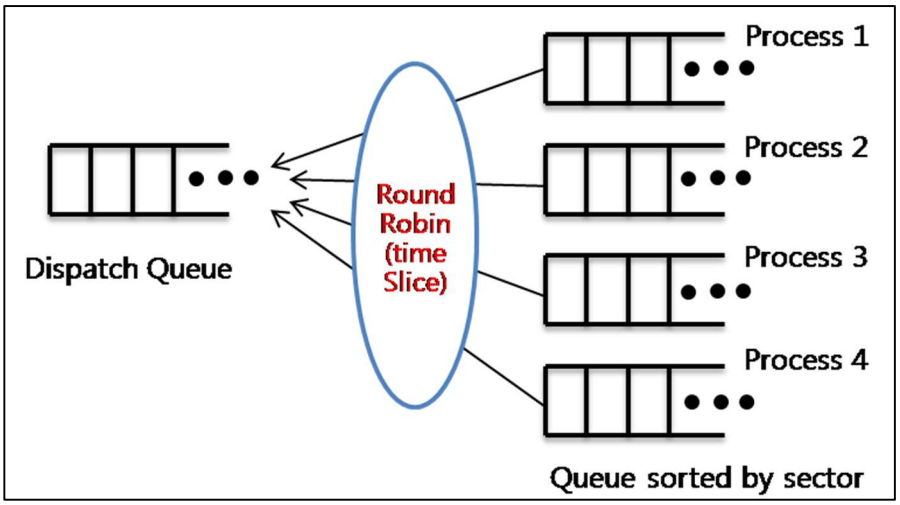
submit_bio()를 통해 전달받은 bio 들은 generic_make_request() 함수에서 디바이스 드라이버에서 제공하는 make_request_fn()(본 과제에서는 nvme_request_rq()가 해당됨) 함수로 디바이스 드라이버에서 요구하는 request 구조체로 만들어 디바이스 드라이버에 전달한다.

Request 구조체는 1 개 이상의 bio 들로 구성되어 있으며, 하나의 request 가 여러 개의 bio 를 관리함으로써 bio 구조체마다 처리해야 했던 작업들을 한 번에 할 수 있으므로 작업의 효율성을 높일 수 있다.

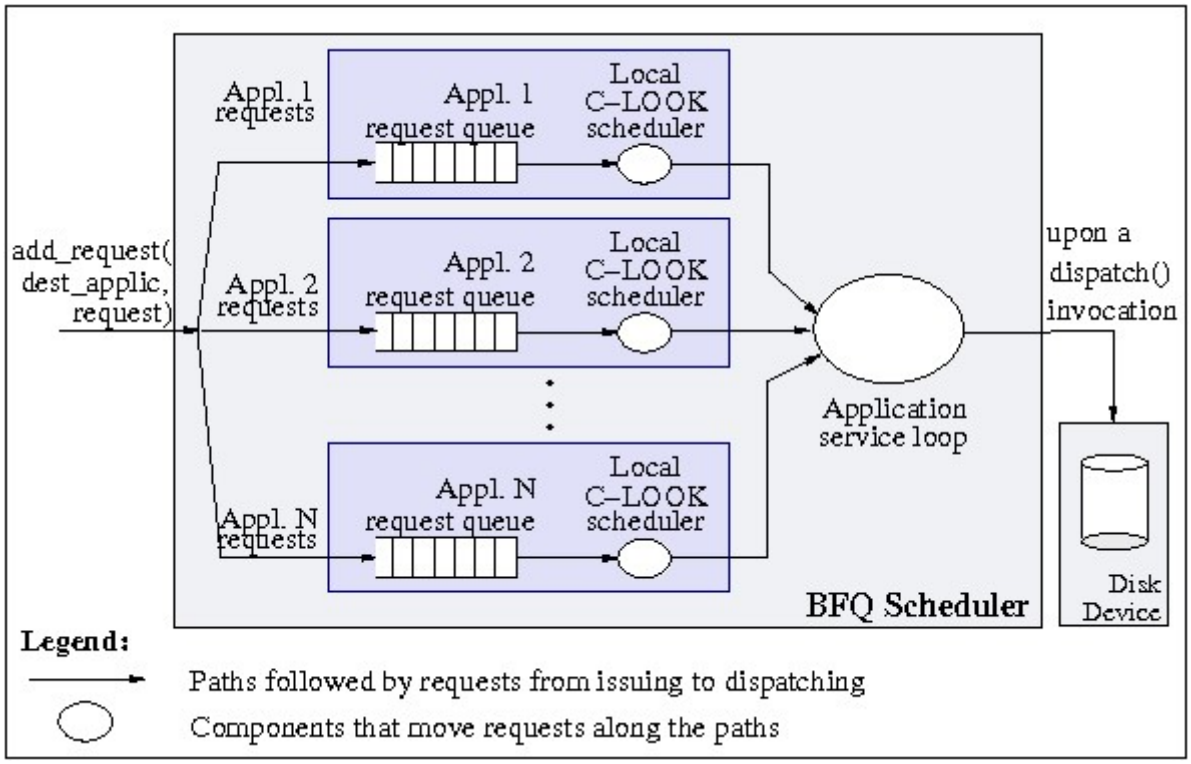
기존의 I/O 블록 레이어는 싱글-큐 형태였기 때문에 멀티-코어 프로세서가 하나의 큐에 접근하였고 전체 스토리지 시스템에 병목현상을 야기하였다. 특히 NUMA 기반의 프로세서 시스템에서 그 병목현상이 더욱 가중되었기 때문에 2-level 의 여러 큐, 즉 멀티-큐를 가지는 I/O 블록 레이어의 형태로 변형하였고, 멀티-코어 프로세서가 각 큐에 독립적으로 접근할 수 있도록 하여 병목현상을 해결한 것이 멀티-큐 블록 레이어이다.

하지만 앞에서 언급했듯이, 블록 I/O 작업은 높은 latency 를 가지는 작업으로써 즉시 처리하면 시스템 전체의 성능이 느려질 수 있다. 따라서 request 는 스케줄러에 의해 merge, reorder 등의 작업을 거쳐 효율적으로 처리될 수 있도록 조정된 후 request_queue 로 전달된다. 멀티-큐 블록 레이어에는 여러 스케줄러들이 존재하지만, 본 팀은 isolation 과 latency 에 관심이 있으므로 BFQ 스케줄러와 카이버 스케줄러를 분석하도록 하겠다.

2.2 BFQ 스케줄러



[그림 7. CFQ 스케줄러 개요]

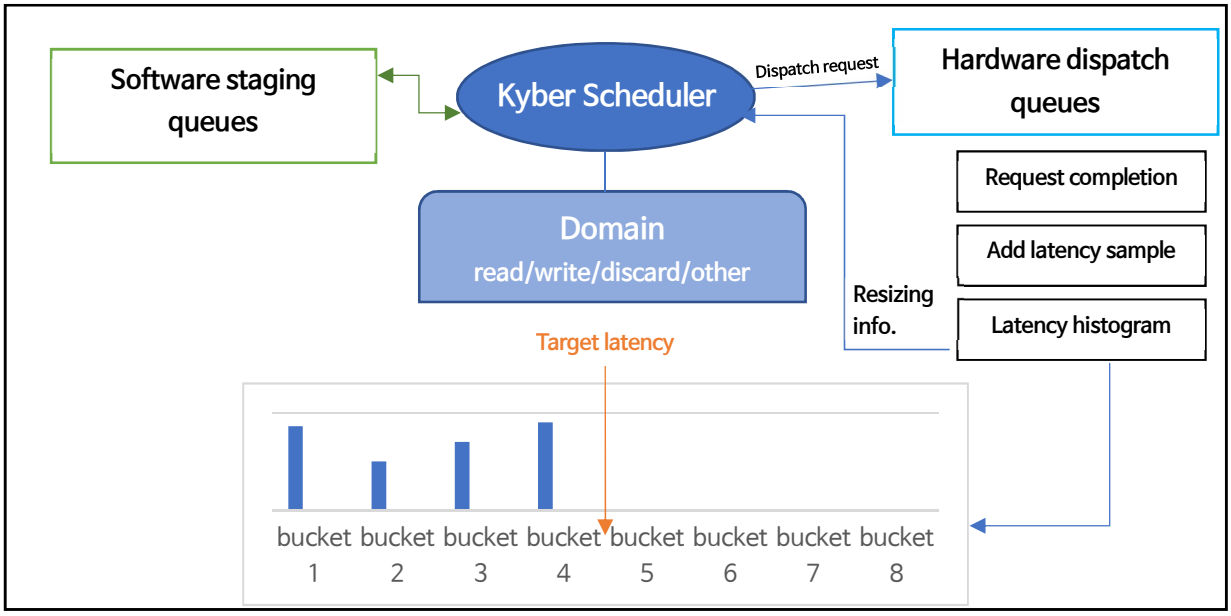


[그림 8. BFQ 스케줄러 개요]

BFQ 스케줄러는 싱글-큐 블록 레이어에 있던 CFQ(Completely Fair Queueing) 스케줄러를 멀티-큐 블록 레이어에 구현한 스케줄러다. 두 스케줄러 목적은 프로세스/스레드(process/thread) 사이에 weight 를 이용해 비례 배분(Proportional Share) 하여 디스크의 throughput 을 나눠 가진다는 데에 있으며, cgroup 의 interface 를 사용한다는 공통점을 가지고 있다. 두 스케줄러의 차이점은 CFQ 는 시간개념을 사용해 순차적으로 라운드로빈(Round-Robin)을 이용해 로드밸런싱(Load-Balancing) 하지만, BFQ 는 시간이 아닌 프로세스/스레드마다 처리할 수 있는 섹터(sector)의 수를 나타내는 "I/O Budget"이라고 불리는 것을 각 프로세스/스레드마다 할당하여 request 마다 섹터의 수량을 측정한 뒤, 프로세스/스레드에게 디스크의 throughput 을 배분하는 스케줄러이다.

어떤 요청을 처리해야 할지 결정해야 할 때, BFQ 스케줄러는 idle(유휴) 디스크에서 가장 먼저 I/O Budget 을 소진할 수 있는 프로세스/스레드를 먼저 선택한다. 따라서, 일반적으로 작은 I/O Budget 을 가진 프로세스/스레드는 큰 I/O Budget 을 가진 프로세스/스레드보다 빨리 처리된다. I/O Budget 은 I/O weight 에 기반하여 계산되고, 또한 관찰된 프로세스/스레드의 이전 행동들에 기반하여 값이 조정된다. 즉, I/O weight 는 우선순위(priority) 값이라고 이해할 수 있다. I/O weight 값은 4 페이지, 1.1 절의 [그림 2]에서처럼 설정할 수 있다.

2.3 카이버 스케줄러



[그림 9. 카이버 스케줄러 구성도]

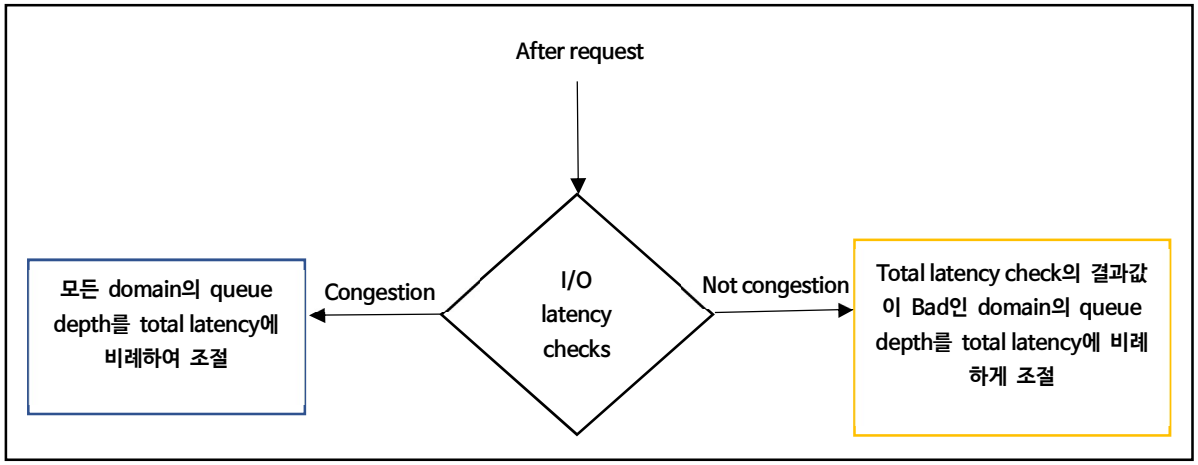
카이버 스케줄러는 큐의 깊이(queue depth)를 조절하는 방식으로 쓰로틀링(throttling)을 걸어 latency 를 조절하는 멀티-큐 블록 레이어의 스케줄러이다.

카이버 스케줄러는 총 4가지의 도메인(Domain)을 가지고 있으나, 메인 도메인은 Read 도메인과 Write 도메인이다. Read 도메인은 동기 요청(Synchronous request)들을 위한 것이고, Write 도메인은 비동기 요청(Asynchronous request)들을 위한 것이라고 말할 수도 있다. Write 요청이 많아진다는 뜻은 비동기 요청이 많아진다는 뜻이고, 이는 동기 요청인 Read 요청의 굶주림(Starvation) 문제가 발생할 수 있다. 따라서, 비동기 요청에 대한 최대 비율을 75%로 설정하여 25%의 큐 공간은 동기 요청이 사용할 수 있도록 하여 이러한 문제를 해결하였다.

사용자는 “/sys/block/*/queue/iosched” 폴더 아래에 존재하는 “read_lat_nsec”, “write_lat_nsec”의 파일을 수정하여 target read latency 와 target write latency 의 초깃값을 설정하고, 카이버 스케줄러는 runtime 에 자체적으로 latency 목표를 조절한다. Latency 를 조절하기 위하여 hardware dispatch queue 에 전송(dispatch) 되는 request 수를 조절하고, 이를 조절하기 위해 각각의 도메인들이 가진 토큰(Token)들의 수를 조절한다. 각 도메인의 토큰 사용 여부는 sbitmap(scalable bitmap)을 통해 관리된다. 이러한 결과로, Hardware dispatch queue 의 depth 를 줄이면 request 당 latency 를 줄일 수 있어 우선순위가 높은 요청에 대한 빠른 처리가 보장되지만, latency 목표를 너무 낮게 설정하게 되면 의도한 대로 latency 는 줄어드는 대신에 merge 할 수 있는 기회가 줄어들어 throughput 이 낮아진다.

카이버 스케줄러의 runtime 에서의 과정을 보면, 매 request 마다 Hardware dispatch queue 로의 request completion 후 latency 샘플링을 실시한다. 샘플링 된 데이터를 이용해 총

8개의 bucket을 두어 히스토그램을 그린 뒤, 중간지점을 목표 latency로 설정하여 도메인 토큰의 개수를 동적으로 조절한다.



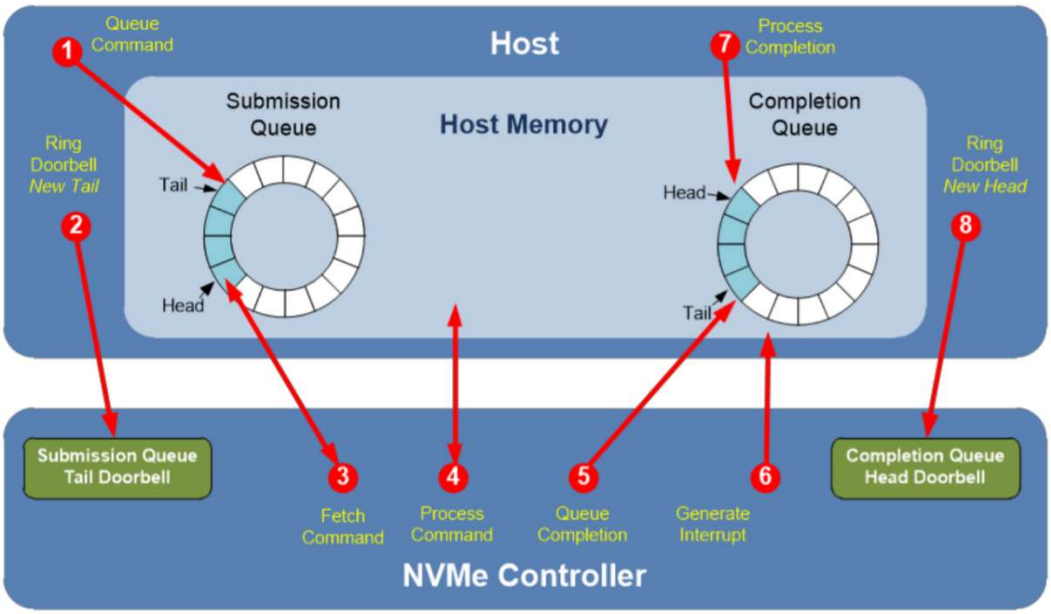
[그림 10. Kyber 스케줄러 도메인 토큰 개수 조절 알고리즘]

도메인 토큰 개수 조절 알고리즘은 I/O latency 를 확인할 때, 90%의 I/O latency 의 분포를 확인하여 congestion 을 판단하고 모든 domain 의 큐 depth 를 total latency 에 비례하게 조절한다. Congestion 이 아닌 경우, 99%의 total latency 분포를 확인하여 결과가 좋지 않은 도메인의 큐 depth 만 total latency 의 99% 분포에 비례하게 조절한다.

2.4 NVMe 리눅스 드라이버

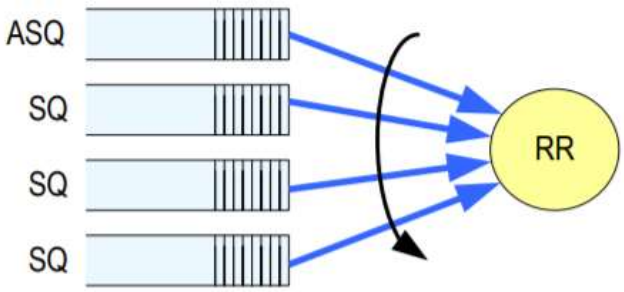
NVMe 는 'NVM Express'의 줄임말로서, PCIe(PCI 익스프레스) 버스에 부착된 비 휘발성 기억 매체 접근을 위한 논리 장치 인터페이스 사양이다. 설계상, NVM 익스프레스는 여러 수준의 병렬화를 허용하여 길이가 긴 다중 명령 큐(multiple long command queue)들을 제공하고, 이는 I/O 오버헤드를 줄이고, latency 를 줄여 성능의 향상을 가져온다.

NVMe 는 NVM Express Workgroup 에서 표준을 정하여 사용하고 있으며, 모바일부터 데이터 센터까지 리눅스를 포함하여 다양한 컴퓨팅 환경에서 사용 가능하다.

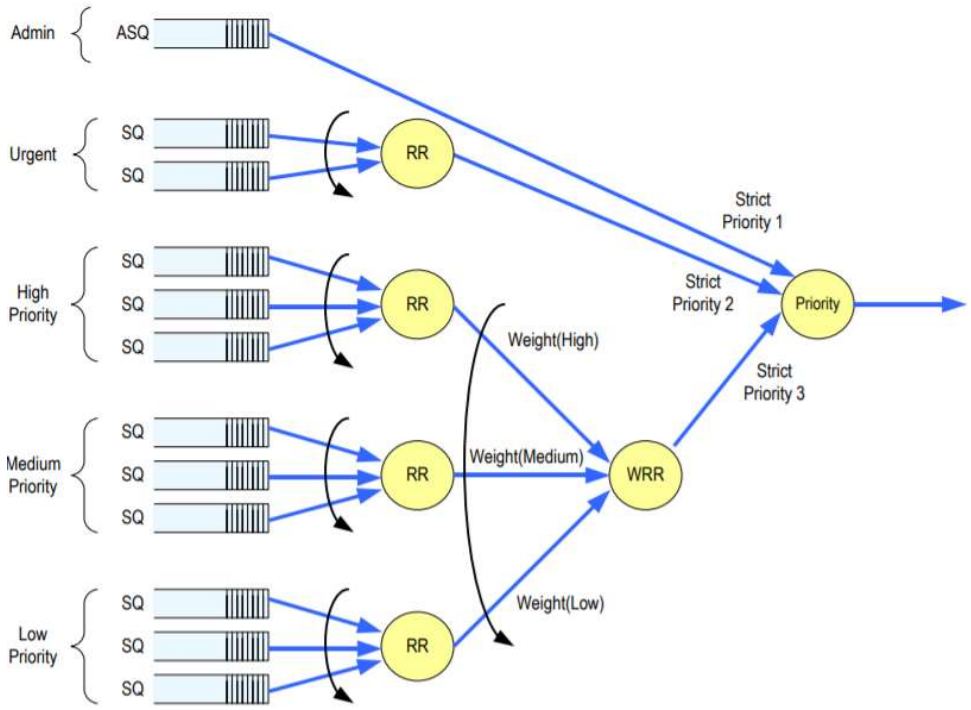


[그림 12. NVMe Ring Doorbell 동작 과정]

NVMe 드라이버는 크게 호스트와 컨트롤러로 나누어져 있다. Host 는 코어마다 I/O Submission Queue 와 I/O Completion Queue 를 가지고 있다. 동작 과정을 살펴보면, NVMe 디바이스에 요청을 하고자 할 때, 블록 레이어에서 전달받은 request 를 NVMe 커맨드로 변경하여 Submission Queue 에 넣게 되고, Ring Doorbell operation 을 통해 NVMe 컨트롤러에게 새로운 커맨드가 Submission Queue 에 추가되었음을 알린다. 이를 확인한 컨트롤러는 호스트의 Submission Queue 에서 커맨드를 가져와 처리하고 호스트의 Completion Queue 에 결과를 전달한 다음 인터럽트를 발생시켜 커맨드의 수행이 끝났음을 알림으로써, NVMe 디바이스의 request 가 처리된다.



[그림 13. Round Robin Arbitration]



[그림 14. Weighted Round Robin Arbitration]

NVMe 컨트롤러는 호스트의 Submission Queue 를 확인하기 위해 기본적으로 Round Robin Arbitration 을 사용한다. 하지만, request 들을 우선순위화(prioritization)가 필요한 경우에는 Weighted Round Robin(WRR)을 사용하여 우선순위화할 수 있다.

3 과제 세부 요구사항

3.1 과제 세부 목표

3.1.1 Isolation

본 팀이 제안하는 스케줄러의 주 목적은 I/O 성능 Isolation 이다. 즉, 멀티-큐 블록 레이어에 cgroup 의 weight 에 기반한 I/O 성능 Isolation 기능을 가진 스케줄러를 구현하여, 클라우드 환경에서 SLA 에 따른 정확한 서비스를 보장할 수 있도록 한다.

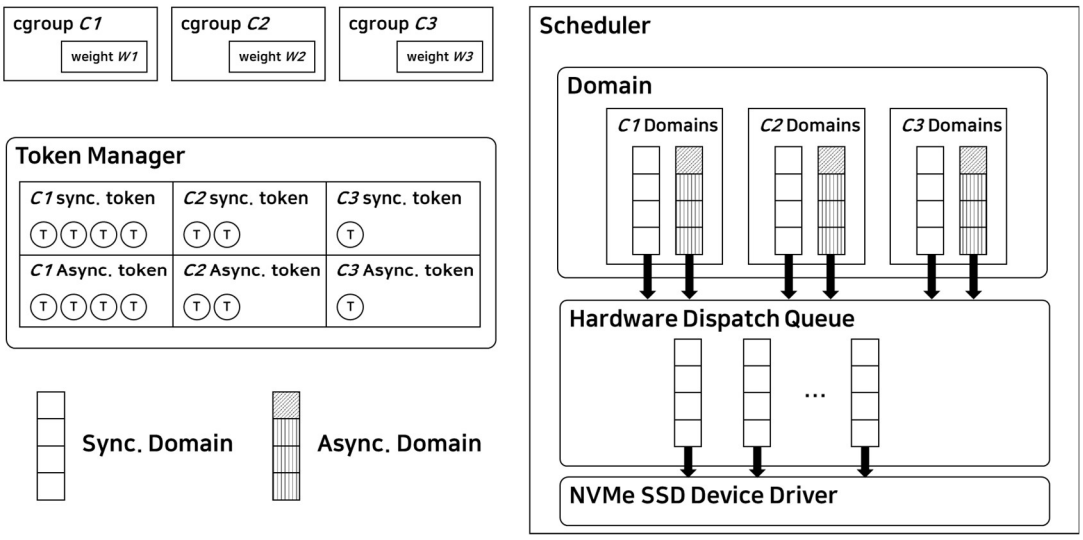
3.1.2 성능

I/O 성능 Isolation 기능을 구현하면서, BFQ 스케줄러 보다 낮은 성능이 관측되지 않도록 한다.

3.1.3 유지 보수

스케줄러를 구현할 때 구조를 복잡하게 하지 않고 간단하게 하여 차후에도 본 스케줄러를 쉽게 유지 보수 할 수 있도록 구현한다.

3.2 과제 세부 설계



[그림 11. 새로운 Scheduler Overview]

본 팀은 스케줄러에서 I/O 성능 Isolation 서비스를 제공하기 위해 “Token Manager”의 개념을 제안하고자 한다. Token Manager 는 카이버 스케줄러에서 토큰을 이용해 성능을 조절하는 점에 착안하여 cgroup 마다 도메인 set 를 할당하고, 해당 도메인마다 토큰을 할당하는 방식을 이용하여 I/O 성능 Isolation 을 구현하고자 한다. 이와 같은 방식으로 구현함으로써 I/O 성능 Isolation 은 물론, I/O 성능 측면에서 Throughput 과 Latency 의 컨트롤도 가능할 것으로 생각된다.

4 개발 일정 및 역할분담

4.1 추진 체계 및 일정

6월				7월					8월					9월				
2주	3주	4주	5주	1주	2주	3주	4주	5주	1주	2주	3주	4주	5주	1주	2주	3주	4주	5주
리눅스 커널 스터디																		
				스케줄러 인터페이스 작성														
				blkio 와의 연동을 위한 스케줄링 구조체 생성														
				적정 Latency 측정 알고리즘 설계														
							중간보고서 작성											
							구조체, 알고리즘 스케줄러에 통합											
								blkio 와 연동										
									안정성, 성능 평가									
										디버깅 및 설계문서 작성								
													안정성, 성능 평가					
														디버깅 및 설계문서 수정				
																최종보고서 작성, 발표 심사 준비		

4.2 구성원 역할 분담

이름	역할 분담
손수호	- Elevator Interface 를 이용해 멀티-큐 블록 레이어에 스케줄러 구현 - 스케줄러의 새로운 그룹 구조체와 알고리즘을 적용해 I/O request 들을 스케줄링하여 request_queue 로 보내는 작업 구현
박기태	- 적정 Latency 를 구하는 알고리즘 설계 - Throughput 과의 trade-off 를 고려하여 Latency 를 보장할 수 있는 알고리즘 설계
권민재	- cgroup 의 서브시스템인 blkio 와 연동 - blkio 에서 받아온 정보와 스케줄러의 연동을 위해서 새로운 스케줄링 구조체 생성
공동	- 리눅스 멀티-큐 블록 레이어 이해 - 시스템 테스트 - 성능 평가 - 설계문서 작성 - 발표 심사 및 시연 준비

[References]

[1] Bedir Tekinerdogan, AlpOral, “Software Architecture for Big Data and the Cloud”.

[2] “computer latency at a human scale” (https://www.prowesscorp.com/computer-latency-at-a-human-scale/)

[3] Linux Kernel Git Repository (https://github.com/torvalds/linux)

[4] cat /var/log/ava 블로그 (http://ari-ava.blogspot.com/)

[5] 작은 서랍 :: 소프트웨어와 일상 블로그 (https://ji007.tistory.com/entry/IO-Schedulers)

[6] LWN.net (https://lwn.net/)

[7] F/OSS study 블로그 (http://studyfoss.egloos.com/)

[8] Red hat 자원 관리 가이드 Documentation (https://access.redhat.com/documentation/ko-kr/red_hat_enterprise_linux/6/html/resource_management_guide/index)

[9] NVM Express 공식사이트 (https://nvmexpress.org/)

[10] NVMe Specification (https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3d-2019.03.20-Ratified.pdf)

[11] NVM Express - 위키피디아 (https://en.wikipedia.org/wiki/NVM_Express)

[12] AWS 컴퓨팅 세부 정보 페이지 (https://aws.amazon.com/ko/compute/sla/)

[13] Sungyoung Ahn 등, “Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems”, USENIX Hotstorage, 2016.

[14] BJØRLING 등, “Linux Block IO: Introducing MultiQueue SSD Access on Multi-Core Systems”, SYSTOR, 2013.

[15] Jie Zhang 등, “FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs”, USENIX OSDI, 2018.

[16] Kanchan Joshi 등, “Enabling NVMe WRR support in Linux Block Layer”, USENIX HotStorage, 2018.

[17] Paolo Valente, Fabio Checconi, “High Throughput Disk Scheduling with Fair Bandwidth Distribution”, IEEE Transactions on Computers, 2010.

[18] Technical Report, “New version of BFQ, benchmark suite and experimental results”, 2014.

[19] Amber Huffman, “NVM Express Overview & Ecosystem Update”, Flash Memory Summit, 2013.

[20] R.Love, “Linux Kernel Development”, 3rd Edition.