리눅스 컨테이너를 위한 I/O 성능 Isolation 프레임워크 개발

분과 D (하드웨어, 보안)

Team ★ X 5

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

School of Electrical and Computer Engineering, Computer Engineering Major

Pusan National University

2019 년 9 월 23 일

지도교수: 안 성 용

목차

I. 과제 개요

1. 기존 문제점

1) SLA

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 AWS EC2 인스턴스 유형 선택 화면]

[그림 1]은 SLA 의 대표적인 예로, IaaS 시장을 대표하는 Amazon AWS 의 인스턴스를 선택하는 모습이다. 사용자는 CPU 와 메모리 그리고 네트워크 성능을 자신의 워크로드에 맞게 선택하여 인스턴스를 생성하게 되고, 이 내용을 토대로 서비스 제공자와 사용자는 SLA 를 생성하게 된다. 이러한 서비스는 서버의 성능을 Docker 또는 Container 와 같은 기술들로 Isolation 하여 사용자에게 제공함으로써 가능하다.

최근 인공지능과 빅데이터를 활용하려는 개인 또는 기업들이 증가하면서 많은 워크로드들은 디스크 I/O 작업을 사용하는 비중이 높다. 디스크 I/O 작업은 다른 작업보다 느리기 때문에, 시스템의 성능에 가장 치명적인 영향을 미치는 작업이다. 하지만 [그림 1]에서 볼 수 있듯이, 인스턴스를 선택하는 화면에서 디스크 I/O 성능을 선택하는 옵션은 없다. 그 이유는 Docker 나 Container 에는 디스크 I/O 를 isolation 할 수 있는 기능이 없기 때문이다.

클라우드 기반 시스템에서는 많은 서비스 사용자가 존재하고, 서비스 사용자들은 서버에 수많은 디스크 I/O 작업 요청을 보낸다. 기존의 디스크 I/O isolation 기능을 지원하지 않는 시스템은 디스크 I/O 작업 요청을 구분하지 않고 모두 동일하게 처리한다. 만약 많은 서비스 사용자들이 한정된 디스크 자원을 사용하려 한다면 좋은 성능의 CPU, 메모리를 가진 사용자라 할지라도 디스크 I/O 작업을 기다려야 하기 때문에 결과적으로 사용자는 원하는 성능을 얻을 수 없으며, 공급자는 SLA 를 이행할 수 없다. 따라서 공급자가 SLA 를 제대로 이행할 수 있고, 사용자는 원하는 성능을 얻을 수 있게 하기 위해서는 사용자가 인스턴스를 생성할 때 디스크 I/O 성능을 선택하는 옵션을 제공하는 것 또한 필요하다.

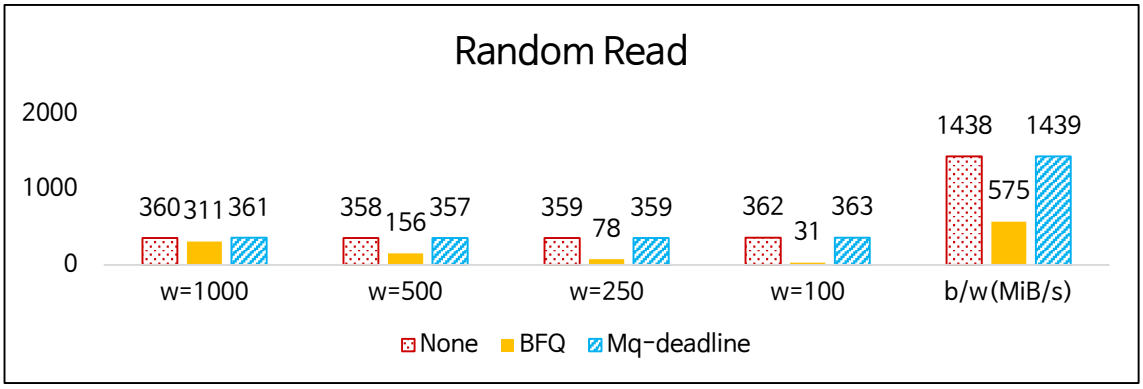
2) BFQ 스케줄러

리눅스 커널의 싱글-큐 블록 레이어에는 CFQ(Completely Fair Queuing) 스케줄러가 디스크 I/O isolation 기능을 수행하고 있으며, 멀티-큐 블록 레이어에서는 BFQ(Budget Fair Queuing) 스케줄러가 디스크 I/O isolation 기능을 수행하고 있다.

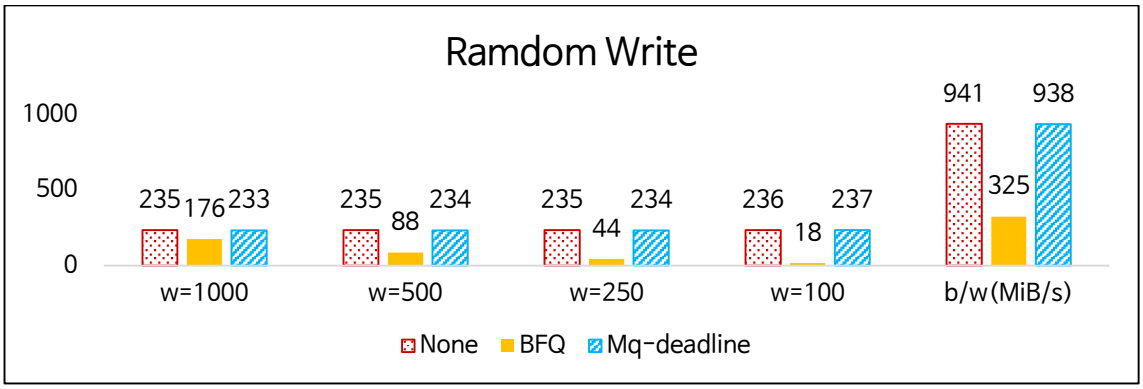
```
oslab4@oslab-server4:~$ ls /sys/fs/cgroup/blkio/highest
blkio.bfq.io_service_bytes                 blkio.throttle.io_serviced_recursive
blkio.bfq.io_service_bytes_recursive       blkio.throttle.read_bps_device
blkio.bfq.io_serviced                      blkio.throttle.read_iops_device
blkio.bfq.io_serviced_recursive            blkio.throttle.write_bps_device
blkio.bfq.weight                           blkio.throttle.write_iops_device
blkio.kyber.weight                         cgroup.clone_children
blkio.reset_stats                          cgroup.procs
blkio.throttle.io_service_bytes            notify_on_release
blkio.throttle.io_service_bytes_recursive  tasks
blkio.throttle.io_serviced
```

[그림 2. BFQ 스케줄러의 weight 설정 파일]

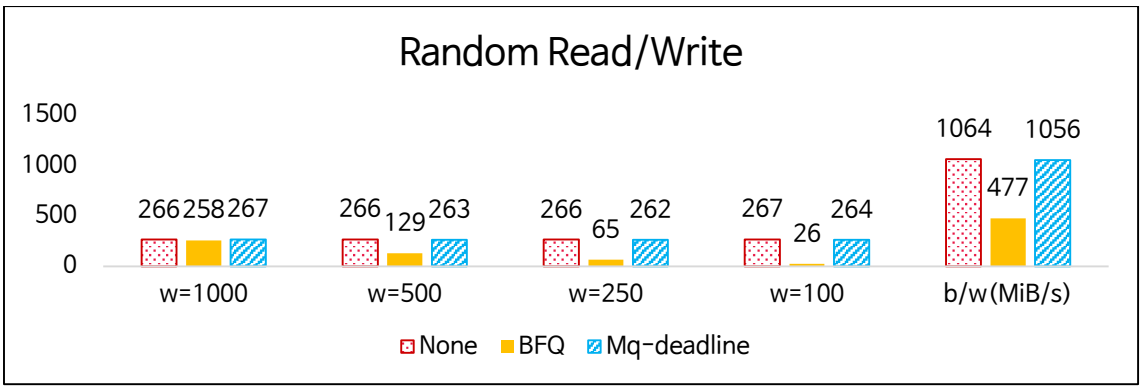
위 스케줄러들은 [그림 2]와 같이 cgroup 들에게 weight 를 부여할 수 있는 파일을 사용자에게 제공함으로써 cgroup 사이에 상대적인 bandwidth 비율을 설정할 수 있게 한다. 스케줄러는 파일에서 설정된 weight 값을 통해 특정 cgroup 의 디스크 I/O 요청들을 throttling 하고, 다른 cgroup 의 디스크 I/O 요청을 처리하는 방식으로 디스크 I/O isolation 이 가능케 한다.



(a) Random Read



(b) Random Write



(c) Random Read/Write (50%/50%)

[그림 3. 멀티-큐 스케줄러별 fio 실험 결과 (w=weight)]

하지만 BFQ 스케줄러는 random read/write request 들에 대해서는 디스크의 bandwidth 를 완전히 사용하지 못한다는 단점이 존재한다. [그림 3]은 cgroup 의 weight 를 각각 1000, 500, 250, 100 으로 설정하고 fio [2]를 통해 random request 에 대한 워크로드를 실행한 결과이다. BFQ 스케줄러는 weight 에 맞게 fairness 를 보장하고 있지만, 전체 bandwidth 사용량이 다른 멀티-큐 블록 I/O 스케줄러인 None 스케줄러, Mq-deadline 스케줄러에 비해 3 배나 낮은 모습을 볼 수 있고, 심지어

가장 높은 weight 를 가진 cgroup 의 throughput 도 다른 스케줄러를 사용했을 때의 throughput 보다 낮게 측정되고 있다. BFQ 스케줄러가 random request 들에 대하여 성능 저하의 원인은 이후 **II. 과제 배경 지식**에서 다루도록 하겠다.

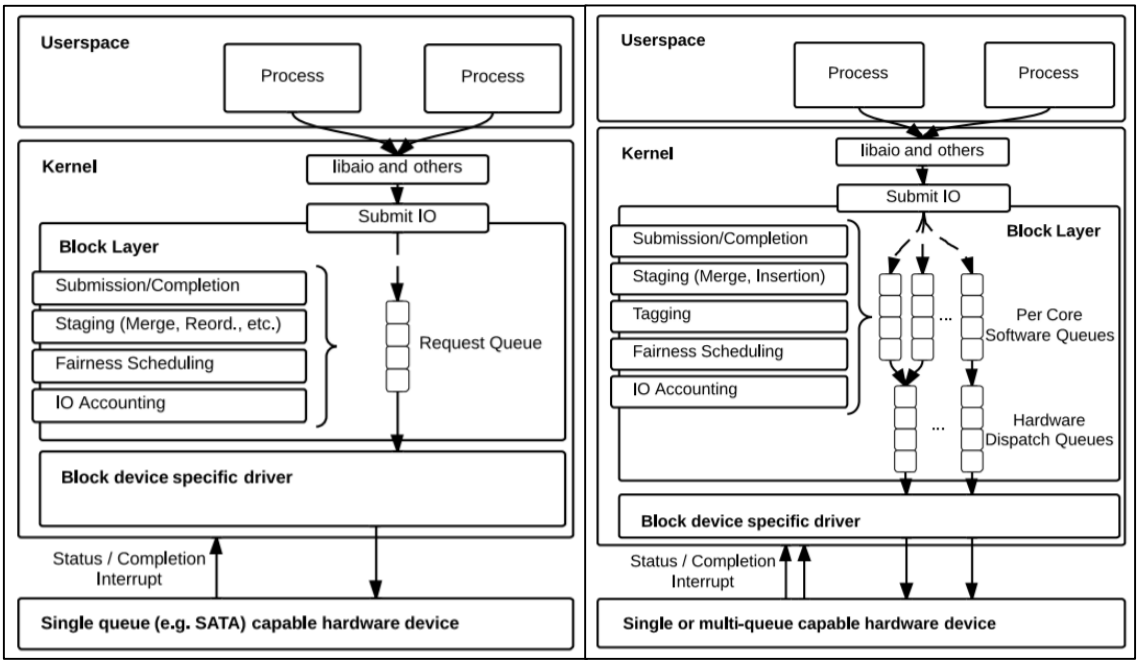
2. 과제 목표

최근에는 NVMe SSD 와 같은 저 지연, 고 성능의 블록 디바이스들이 출시되고 있다. 이러한 디바이스를 지원하는 멀티-큐 블록 레이어에서 I/O 성능 fairness 를 보장하는 스케줄러는 BFQ 스케줄러가 유일하며, 이 마저도 random request 를 처리할 때엔 디스크의 자원 낭비가 심각하다. 여러 사용자가 동시에 이용하는 클라우드 시스템의 스토리지는 연속적인 공간을 접근하는 sequential request 의 형태보다는 random request 가 더 많이 존재할 것이다. 하지만 BFQ 스케줄러는 random request 에서 좋지 못한 성능을 보이고 있다. 이러한 상황에서 클라우드 공급자들은 BFQ 스케줄러를 자신들의 시스템에 도입하는 것은 무리라고 판단했을 것이다.

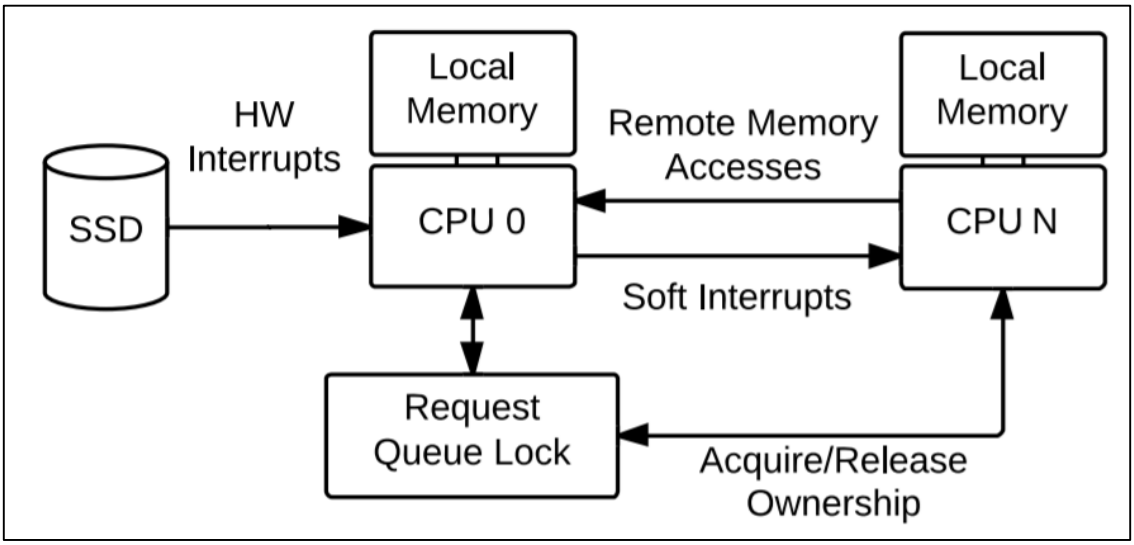
따라서 본 팀의 목표는 work-conserving (디바이스의 유휴자원을 모두 활용) 하고 유지보수가 쉬운 멀티-큐 I/O 스케줄러 개발이다. 본 팀이 제안하는 Kyber Fairness 스케줄러의 목표는 BFQ 스케줄러가 random request 에서 보이는 성능보다 나은 성능을 가지고, 10,397 LOC 의 BFQ 스케줄러보다 가벼운 스케줄러를 구현하여 유지보수를 쉽게 할 수 있도록 만드는 것이다.

II. 과제 배경 지식

1. 멀티-큐 블록 레이어



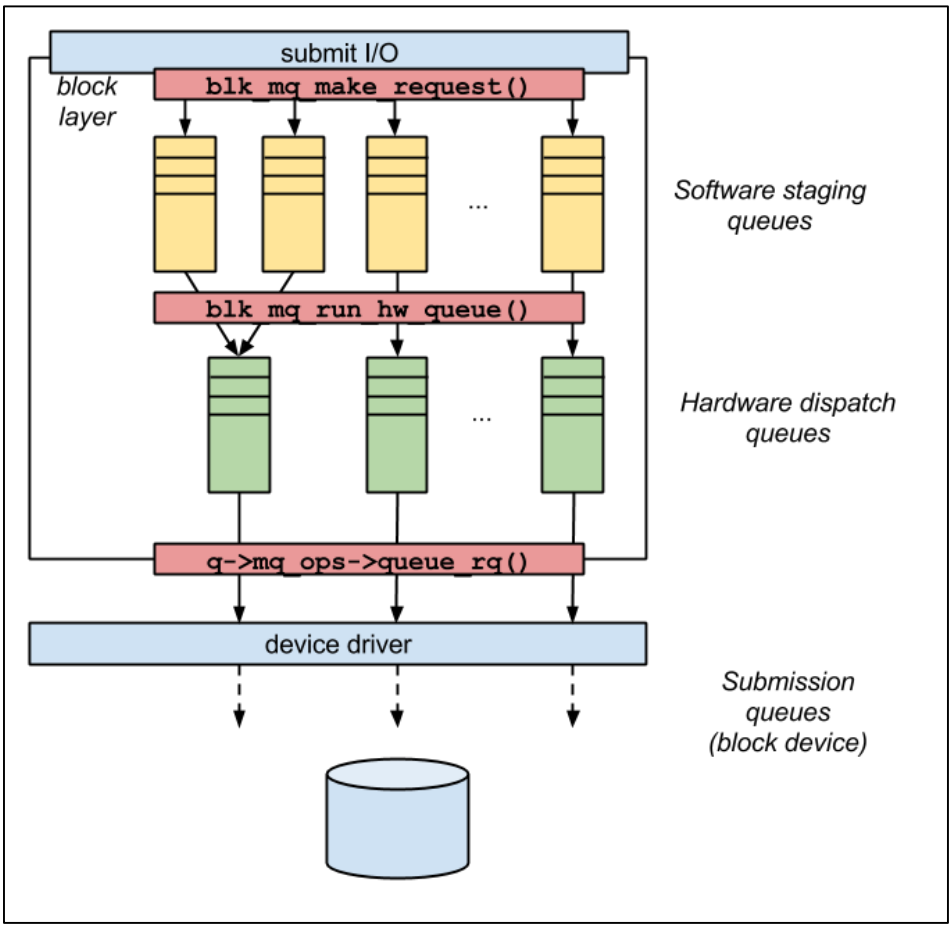
[그림 4. 싱글-큐 블록 레이어와 멀티-큐 블록 레이어]



[그림 5. 싱글-큐 블록 레이어에서의 bottle-neck]

멀티-큐 블록 레이어 [3]는 NVMe SSD 와 같이 큐가 여러 개인 블록 디바이스가 등장하면서 싱글-큐 블록 레이어에서 생긴 bottle-neck 을 해결하기 위해 등장한 레이어다. 기존의 싱글-큐 블록 레이어는

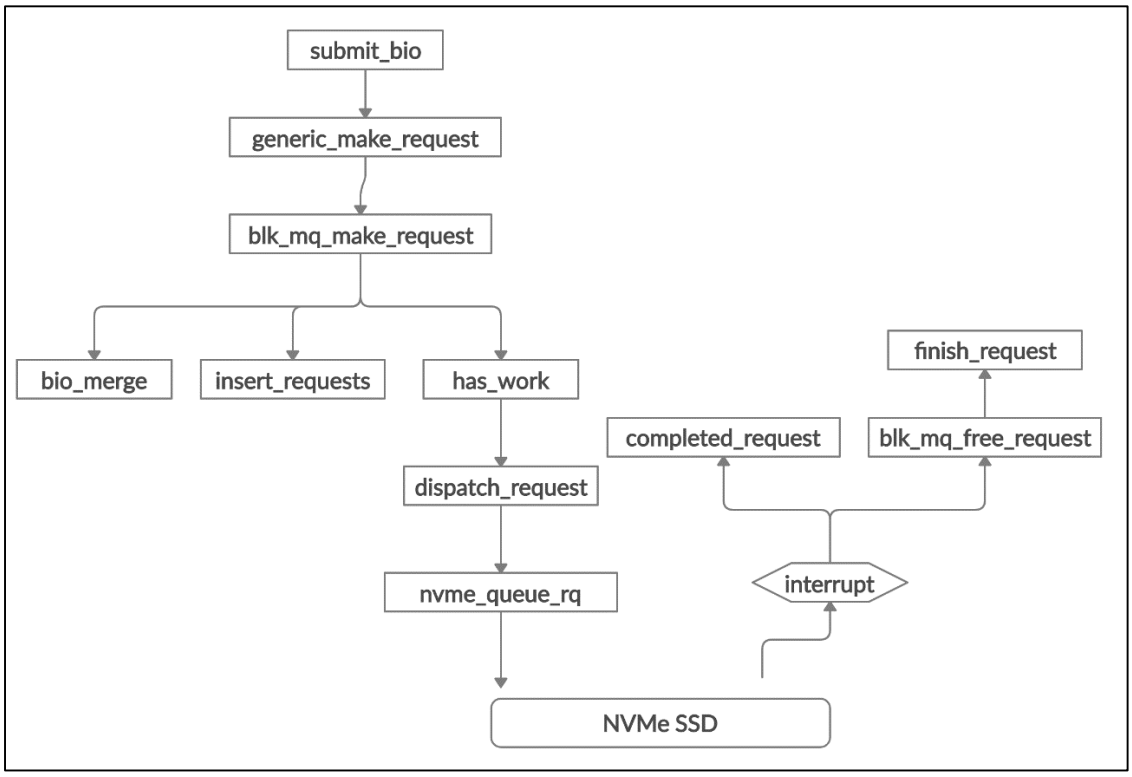
모든 request 가 하나의 큐로 전달되면서 Request Queue Locking, Hardware Interrupts, Remote Memory Access 와 같은 문제로 bottle-neck 이 발생해 멀티-큐 블록 디바이스를 제대로 활용하지 못하였다. 하지만 멀티-큐 블록 레이어는 프로세서의 개수에 맞춰 S/W 큐를 생성하고 블록 디바이스의 큐 개수에 맞춰 H/W 큐를 생성한다. 그리고 S/W 큐와 H/W 큐를 일대일 또는 다대일로 연결함으로써 멀티-큐 블록 디바이스의 큐를 최대한 활용할 수 있게 한다.



[그림 6. 리눅스 커널 멀티-큐 블록 레이어 구조]

멀티-큐 블록 레이어는 리눅스 커널 내에서 파일시스템과 블록 디바이스 드라이버 사이에 위치한다. 파일시스템은 요청된 논리적 주소를 파일 descriptor 등을 통해 물리적 주소로 바꿔주는 역할을 하며, 블록 디바이스 드라이버는 실제 블록 디바이스와 커널을 연결시켜주는 역할을 한다. 따라서, 멀티-큐 블록 레이어는 파일시스템에서 받은 물리적 주소를 디바이스 드라이버에 전달하는 역할을 하게 되며 그 중간과정에서 성능을 위한 추가적인 작업을 실시하게 된다.

파일시스템은 블록 I/O 요청의 연산 종류, 해당 연산에 대한 디스크 영역의 정보와 블록 I/O 요청을 수행할 데이터를 저장하기 위한 메모리 영역 정보 등을 포함하는 bio 구조체를 submit_bio() 함수를 통해 멀티-큐 블록 레이어로 전달한다. 그리고 멀티-큐 블록 레이어는 blk_mq_make_request() 함수를 통해 bio를 블록 디바이스에 맞게 request 구조체로 만든다. 그리고 q->mq_ops->queue_rq() 함수를 통해 request 들을 드라이버로 전달하는데, 중간에 성능 향상을 위해서 스케줄러를 통해 추가적인 작업을 실시할 수 있다.

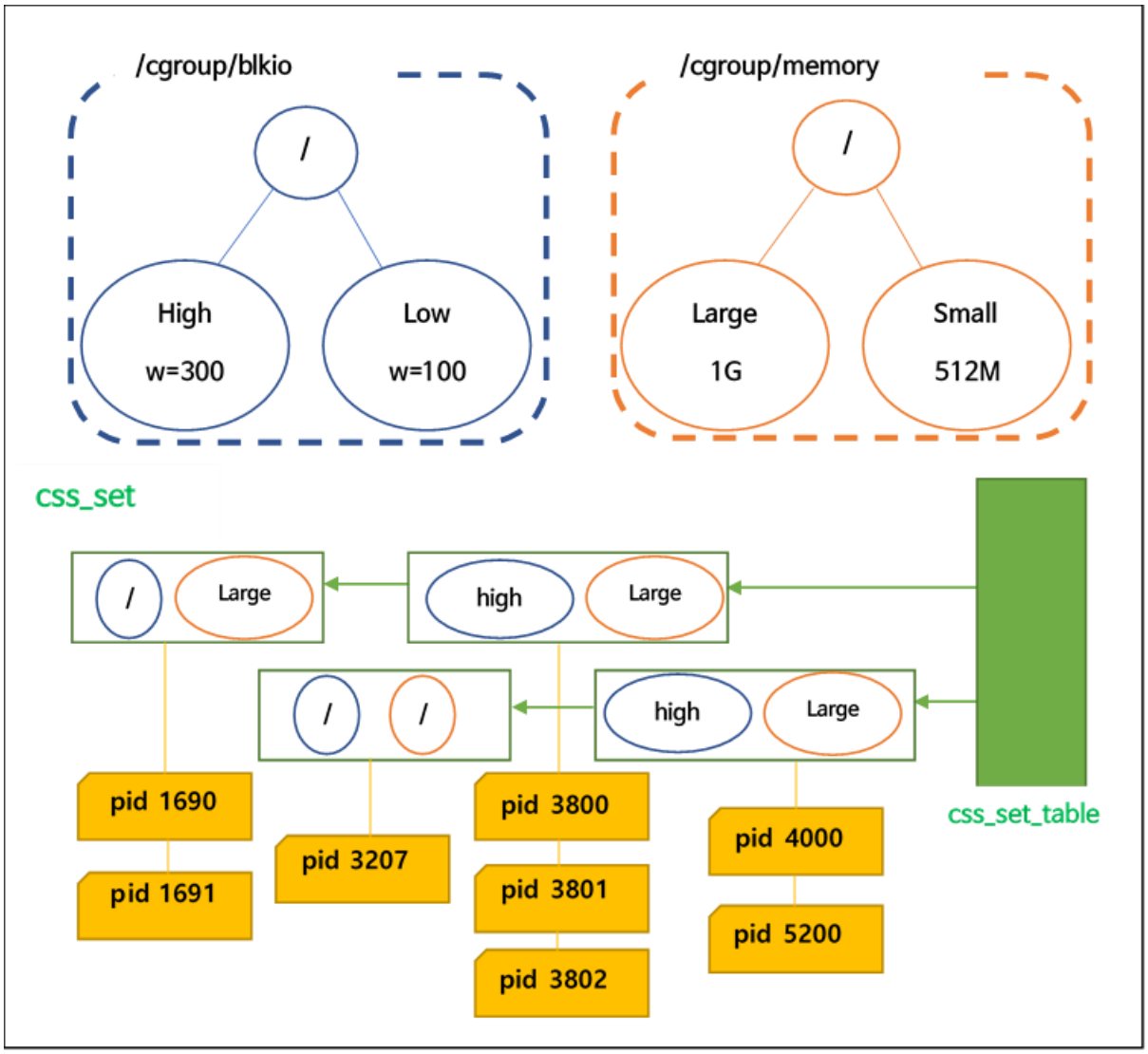


[그림 7. 멀티-큐 블록 레이어 스케줄러에서 request 처리 과정]

멀티-큐 블록 레이어 스케줄러는 software staging queue(이하 S/W 큐)에서 hardware dispatch queue(이하 H/W 큐)로 dispatch 할 때 스케줄링을 하면서 최적의 throughput 과 latency 를 얻기 위해 노력한다. None 스케줄러를 제외한 스케줄러들을 사용할 시에 호출되는 함수는 대표적으로 bio_merge(), insert_requests(), has_work(), dispatch_request(), completed_request(), finish_request()가 있고, 이를 통해 스케줄러가 request 들을 제어한다. bio_merge()는 bio 를 request 로 변경할 때, S/W 큐에 merge 할 수 있는 request 가 존재한다면 bio 구조체를 해당 request 에 merge 시킨다. 이렇게 하나의 request 가 여러 개의 bio 를 관리하여 블록 디바이스에 보내는 request 의 수를 줄이고, 성능을 향상시킬 수 있다. insert_requests()는

bio_merge()를 통해 생성된 request 를 블록 I/O 레이어의 S/W 큐에 추가하는 작업을 한다. dispatch_request()는 H/W 큐에 있는 request 를 블록 디바이스 드라이버로 디스패치한다. 마지막으로 블록 디바이스에서 request 처리가 완료된 경우 completed_request()와 finish_request() 함수가 호출된다. [그림 7]은 함수 호출 순서를 요약한 것이다.

2. LINUX CGROUP

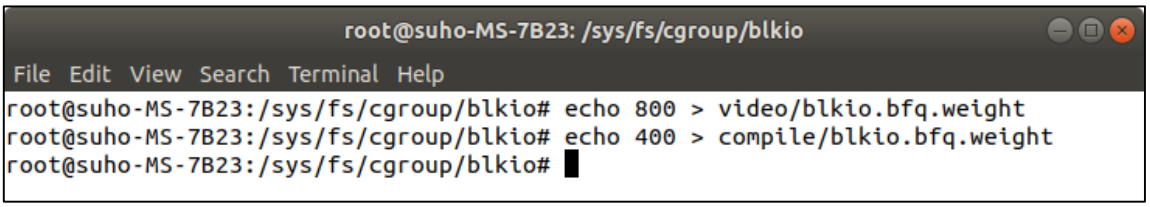


[그림 8. cgroup 파일 시스템 예시]

리눅스 컨테이너는 cgroup(control group) [4]을 이용해 프로세스의 자원을 할당하고 관리한다. 커널 v2.6.24 에 처음 merge 된 cgroup 은 시스템 상에서 동작 중인 프로세스들을 그룹으로 만들어

제어할 수 있도록 도와주는 기능이다. cgroup 은 [그림 8]과 같이 파일시스템으로 마운트 한 뒤 사용한다. cgroup 자체만으로는 동작 중인 태스크의 그룹만 만들어주며, 실제 자원(CPU, memory, disk, network)의 분배는 net_cls, ns, cpuacct, devices, freezer 와 같은 서브시스템이 필요하다.

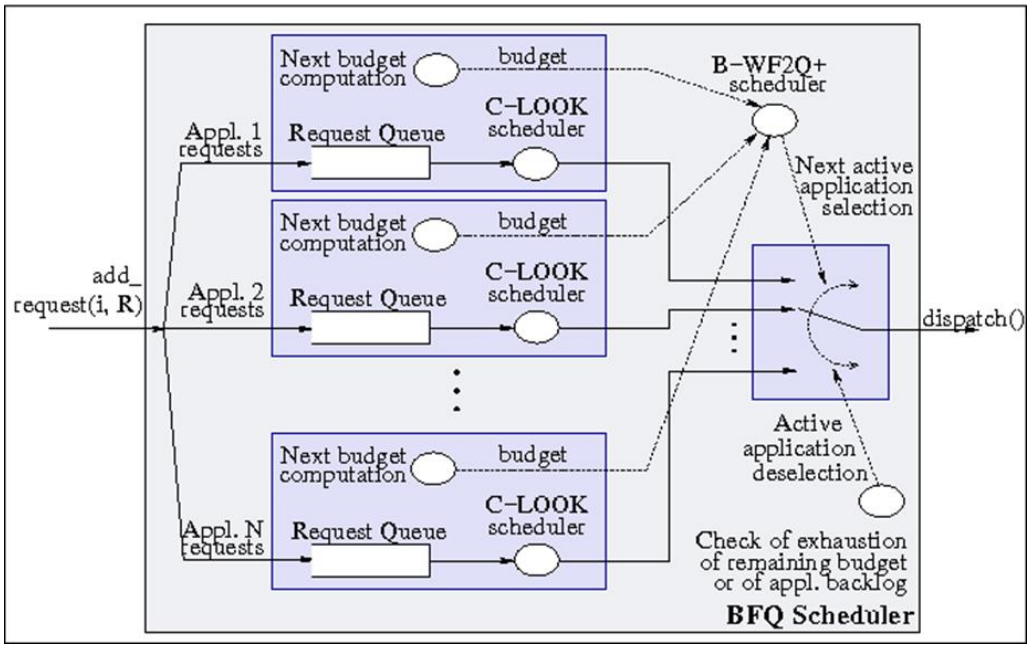
특히 cgroup 은 'blkio'라는 블록 디바이스를 위한 서브시스템을 지원하고 있고, 싱글-큐 블록 레이어에서는 CFQ 스케줄러, 멀티-큐 블록 레이어에서는 BFQ 스케줄러가 블록 디바이스 자원을 관리하기 위해 존재한다. [그림 9]는 blkio 서브시스템의 blkio.bfq.weight 가상 파일을 통해 video cgroup 의 weight 를 800, compile cgroup 의 weight 를 400 으로 할당하는 예시이다.



[그림 9. blkio 를 통한 cgroup weight 변경 예시]

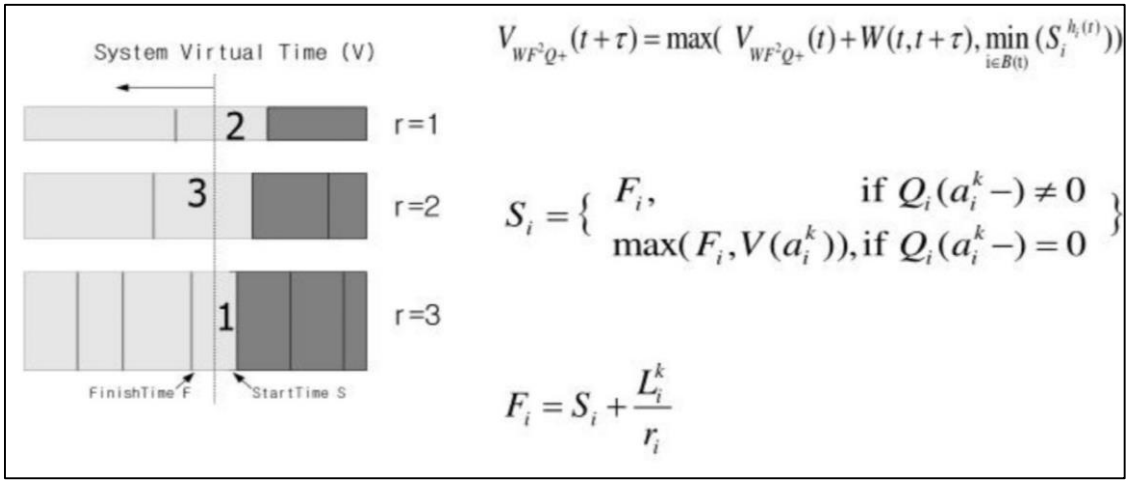
3. 멀티-큐 블록 I/O 스케줄러

1) BFQ 스케줄러



[그림 10. BFQ 스케줄러 오버 뷰]

BFQ 스케줄러 [5]는 싱글-큐 블록 레이어에 있던 CFQ 스케줄러를 멀티-큐 블록 레이어에 구현한 스케줄러이다. 두 스케줄러 목적은 weight 를 이용해 블록 디바이스 자원을 비례 배분하여 디스크의 throughput 을 나눠 가진다는 데에 있으며, cgroup 의 인터페이스를 사용한다는 공통점을 가지고 있다.

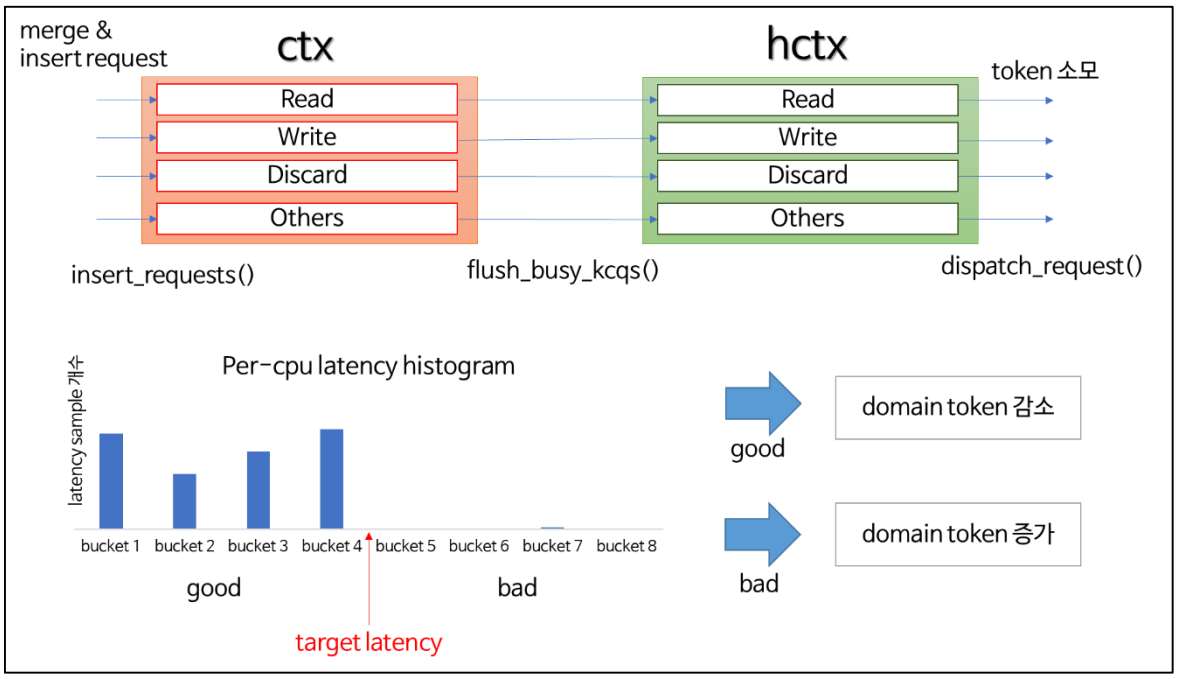


[그림 11. Worst-case Fair weighted Fair Queuing]

BFQ 스케줄러의 스케줄링은 네트워크에서 사용하는 스케줄링 기법인 WF^2Q+ [6]에 기초하여 스케줄링 한다. WF^2Q+는 weight 를 고려하여 현재 시점에서 모든 큐들의 request finish time 을 계산하고, 가장 작은 finish time 을 가지는 큐를 선택하는 방식으로 스케줄링 된다.

하지만 **Ⅰ. 과제 배경 지식**에서도 언급했듯이 BFQ 스케줄러는 random request 에 대해서는 성능이 급격하게 감소한다. 그 이유는 request 의 크기와 연관이 있다. Sequential request 의 경우엔 bio 들이 기존에 S/W 큐에 있는 request 들과 활발하게 merge 하여 request 들의 크기가 크지만, random request 는 merge 가 거의 일어나지 않기 때문에 크기가 작다. 따라서 sequential request 의 경우 스케줄링이 빈번하게 일어나지 않아 오버헤드가 적은 반면, random request 의 경우에는 request 의 크기가 작기 때문에 스케줄링이 빈번하게 발생하고 전체적인 오버헤드가 커져 성능이 저하된다.

2) 카이버 스케줄러



[그림 12. 카이버 스케줄러 오버 뷰]

카이버 스케줄러는 S/W 큐의 깊이를 조절하는 방식으로 최상의 latency 를 얻는 것을 목표로 하는 멀티-큐 블록 레이어의 스케줄러이다. 카이버 스케줄러는 S/W 큐와 H/W 큐 마다 Read, Write, Discard, Others 의 총 4 가지의 도메인을 가지고 있다. 이 중에 주로 사용되는 도메인은 Read 도메인과 Write 도메인이다. 일반적으로 Read 도메인은 동기(Synchronous) request 들을 위한 것이고, Write 도메인은 비동기(Asynchronous) request 들을 위한 것이라고 말할 수도 있다. 만약 비동기 request 인 write request 가 많아진다면 이는 동기 request 인 read request 의 굶주림 문제가 발생할 수 있다. 따라서, 비동기 request 에 대한 최대 비율을 75%로 설정하여 25%의 큐 공간은 동기 요청이 보장받을 수 있도록 하여 굶주림 문제를 해결하였다.

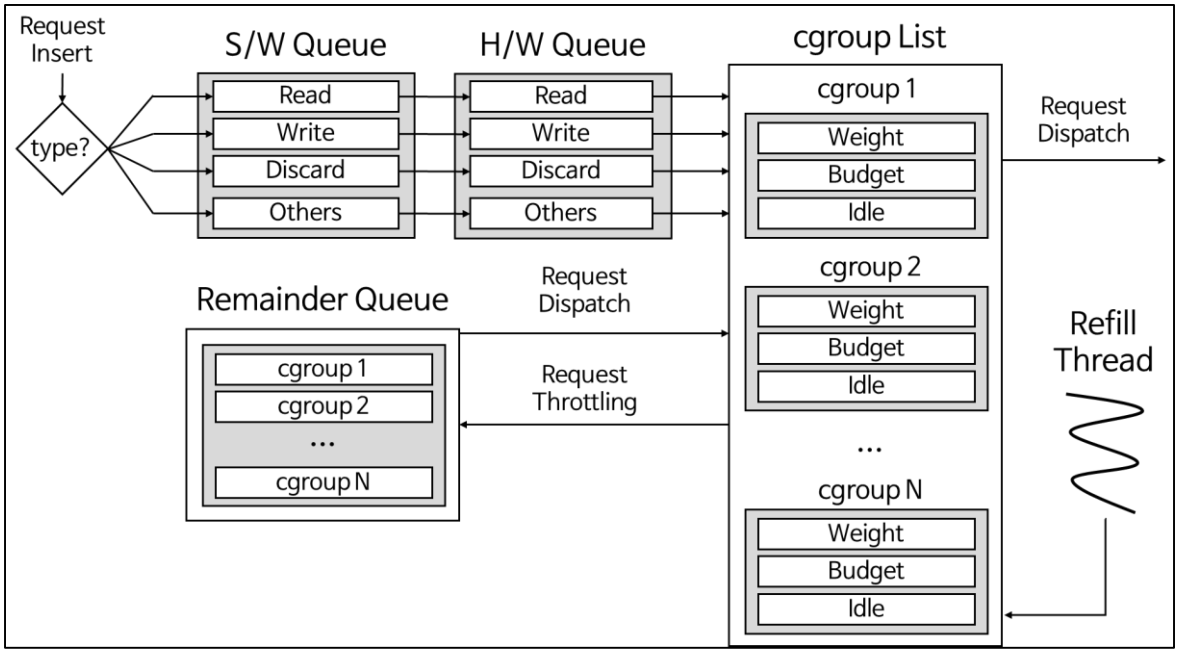
사용자는 카이버 스케줄러를 사용할 때, “/sys/block/*/queue/iosched” 폴더 아래에 존재하는 “read_lat_nsec”, “write_lat_nsec”의 파일에 target read latency 와 target write latency 를 입력하여 초기 target latency 를 설정하고, 입력된 latency 목표를 달성하기 위해 토큰 수를 조절한다. 스케줄러는 H/W 큐에서 디스패치될 때 토큰 수를 감소시키고, 토큰이 부족하다면 디스패치를 쓰로틀링 함으로써 latency 를 유지하려고 노력한다. 토큰 수는 request 가 complete 될 때 기록된 값이 저장되어있는 8 개의 latency 버킷을 통한 히스토그램으로 계산된다. 히스토그램에서 90%, 99%에

위치한 latency 를 선택하여 값이 만약 타겟 latency 보다 크다면 token 수를 늘려 latency 를 줄이려고 노력한다. 반대로 타겟 latency 보다 작다면 token 수를 줄여 S/W 큐에 오래 머물게 하고, 기다리는 동안 merge 가 많이 일어나게 해 throughput 을 높인다.

III. 과제 수행 내용

1. 초기 설계 문제점 및 개선된 설계

1) 초기 설계 및 문제점



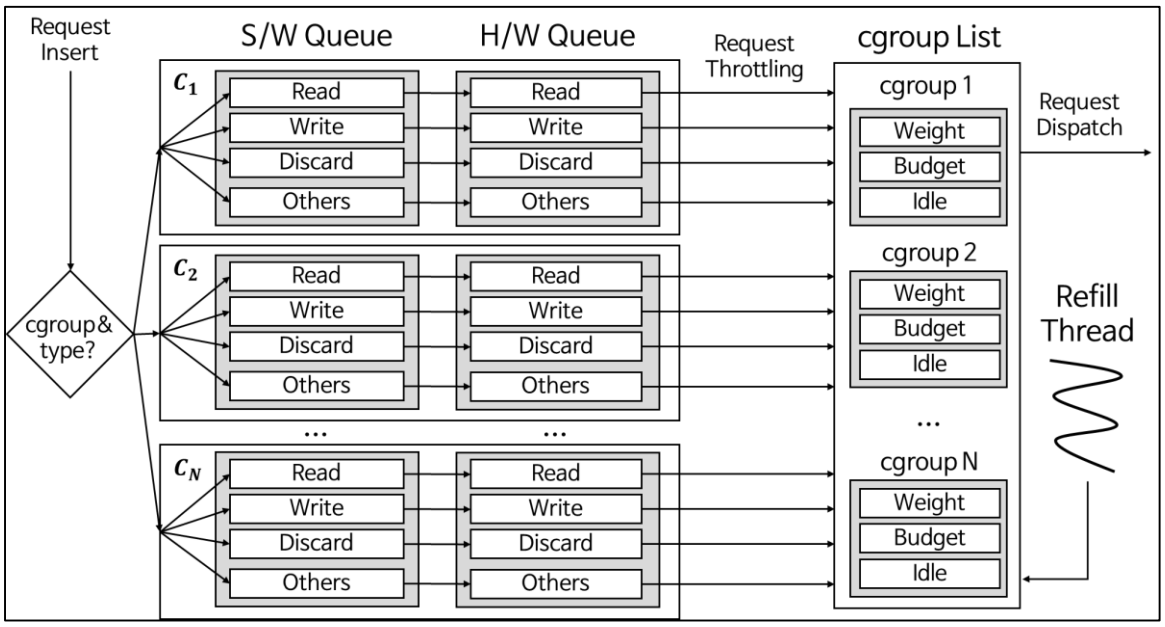
[그림 13. Kyber Fairness 스케줄러 초기 설계]

본 팀이 설계한 Kyber Fairness 스케줄러의 스케줄링 방식은 budget 에 기반하여 request 를 처리함으로써 cgroup 들 간의 fairness 를 보장하는 스케줄링이다. Request 를 디스패치할 때 먼저 해당 request 를 보낸 cgroup 의 budget 을 확인하여 0 보다 크다면 request 의 크기만큼 뺀 뒤에 request 를 처리하고, budget 이 0 보다 작다면 그 request 는 쓰로틀링 하는 방식이다.

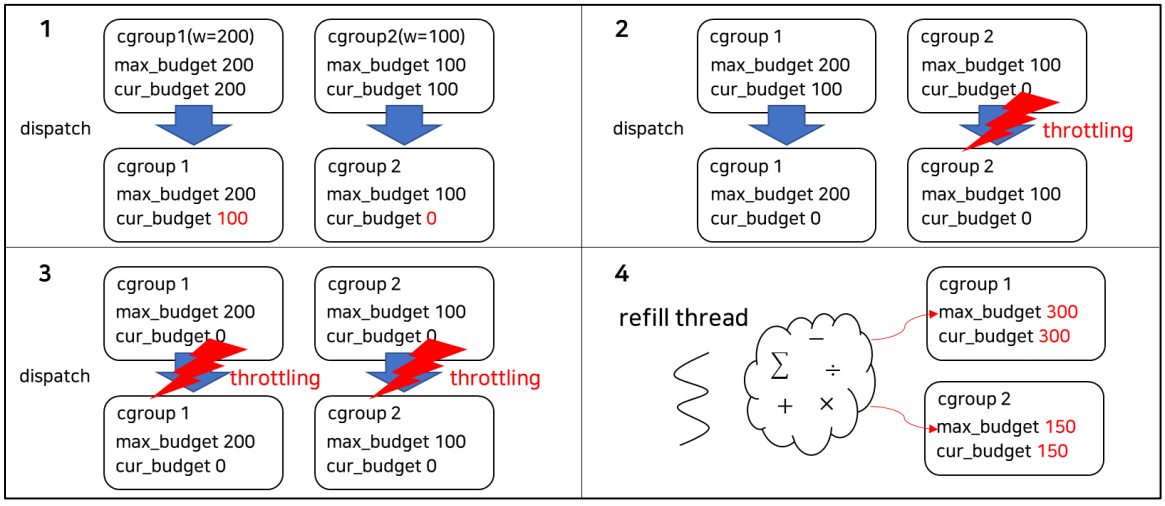
초기 설계는 기존의 S/W 큐와 도메인을 그대로 유지하고, 하나의 S/W 큐에 다양한 cgroup 의 request 가 존재한다고 생각하여 budget 이 부족한 cgroup 의 request 를 잠시 대기시켜 놓는 remainder 큐를 구상하였다. Request 를 디스패치할 때 해당 request cgroup 의 budget 이 없는 상황이라면 request 를 remainder 큐로 이동시키고, budget 이 리필 됐을 때 remainder 큐에 있는 request 들을 우선적으로 처리하는 방식을 고안하였다.

하지만 초기 설계의 문제점은 실제로 프로세서가 프로세싱을 할 때에 일반적으로 하나의 cgroup 에 대한 처리를 지속적으로 할 확률이 높다는 것을 알았고, 새로 추가한 remainder 큐는 글로벌하게 존재하기 때문에 bottle-neck 이 발생한다. 따라서 본 팀은 cgroup 별로 S/W 큐와 H/W 큐를 생성하였다.

2) 개선된 설계



(a) 개선된 설계 오버 뷰



(b) 스케줄러 동작 매커니즘

[그림 14. Kyber fairness 구조체 오버 뷰 및 매커니즘]

구조적으로는 cgroup 별로 S/W 큐와 H/W 큐를 생성하였고, request 를 디스패치할 때 request 가 속한 cgroup 의 budget 이 부족하다면 request 를 블록 디바이스로 디스패치하지 않고 해당 H/W 큐에 쓰로틀링 시켜 둔다. 또한 budget 을 리필 해야 할 때 리필을 도와주는 리필 쓰레드가 kyber_fairness_global 구조체 정보를 이용하여 cgroup 별로 max_budget 을 계산하고, max_budget 만큼 cur_budget 을 채워주는 방식으로 새롭게 설계하였다.

[그림 14.b]는 Kyber Fairness 스케줄러가 어떤 매커니즘으로 동작하고 있는지 보여주고 있다. [그림 14.b.1]은 두 cgroup 이 각각 200, 100 의 weight 를 가지고 있고 크기가 100 인 request 가 들어와 cgroup 의 budget 이 100 씩 감소해 각각 100 과 0 으로 변경된 모습이다. [그림 14.b.2]는 budget 이 각각 100 과 0 인 상황에서 request 가 도착했을 때 budget 이 남아있던 cgroup1 은 크기가 100 인 request 를 처리하고 budget 이 0 으로 변경된 모습이며, budget 이 0 인 cgroup2 는 request 를 처리하지 못하고 쓰로틀링 당하고 있는 모습이다. [그림 14.b.3]은 또 다른 request 가 도착하였지만 두 cgroup 모두 남아있는 budget 이 없어 쓰로틀링 당한 모습이다. 이렇게 모든 cgorup 의 budget 이 0 이하여서 쓰로틀링 당한 상황이라면 [그림 14.b.4]처럼 리필 쓰레드를 생성하여 cgroup 들에게 budget 을 새로 할당한다. 리필 쓰레드는 이전에 request 를 처리했던 정보를 활용하여 총 bandwidth 를 재계산하고 budget 의 양을 조절한다. [그림 14.b.4]는 총 300 의 budget 을 기준 시간인 100ms 보다 빠르게 처리했기 때문에 총 budget 을 450 으로 변경시키고 weight 에 따라 각각의 cgroup 에게 300, 150 의 budget 을 나눠준 모습이다.

2. 구현 코드 설명

1) cgroup blkio 서브시스템 연동

I/O 성능 fairness 를 보장하기 위해선 먼저 cgroup 의 서브시스템인 blkio 를 이용하여 weight 정보를 얻을 수 있게 해야 한다. blkio weight 정보는 cftype 이라는 구조체를 통해 시스템 파일을 정의하여 등록하면 blkio 의 weight 를 입력할 수 있는 파일이 생성되고, 사용자가 해당 파일에 weight 정보를 입력한다면 weight 정보를 얻을 수 있다. 이는 blkcg_policy 인터페이스를 구현함으로써, 함수 포인터의 호출을 통해 이루어진다. [그림 15]는 blkcg_policy 인터페이스를 구현해 만든 blkcg_policy_kyber 이다.

```
static struct blkcg_policy blkcg_policy_kyber = {
	/* weight 설정 */
	.dfl_cftypes		= kyber_blkg_files,
	.legacy_cftypes		= kyber_blkcg_legacy_files,
	/* blkcg_policy_data 타입 변수 할당 및 해제 시 호출되는 함수 */
	.cpd_alloc_fn		= kyber_cpd_alloc,
	.cpd_init_fn		= kyber_cpd_init,
	.cpd_free_fn		= kyber_cpd_free,
	/* blkg_policy_data 타입 변수 할당 및 해제 시 호출되는 함수 */
	.pd_alloc_fn		= kyber_pd_alloc,
	.pd_init_fn		= kyber_pd_init,
	.pd_free_fn		= kyber_pd_free,
};
```

[그림 15. blkcg_policy_kyber]

커널에서 cgroup 별 정보는 blkcg 라는 구조체를 통해 알아낼 수 있다. blkcg 와 blkcg_policy 는 blkcg_policy_data 를 통해 cgroup 마다 모든 policy 가 연결되어 있다. 쉽게 말해 cgroup 수 x blkcg_policy 수만큼의 blkcg_policy_data 가 존재하는 것이다.

Kyber Fairness 스케줄러에서는 cgroup weight 정보를 저장하여 스케줄러가 사용할 수 있게 하기 위해, kyber_fairness_data 에 weight 멤버변수와 함께 blkcg_policy_data 타입의 멤버변수를 포함하여 policy 등록 및 서브시스템 활성화를 할 때 cgroup 마다 kyber_fairness_data 가 생성될 수 있게 하였다.

```
struct kyber_fairness_data {
	struct blkcg_policy_data pd;
	unsigned int weight;
};
```

[그림 16. kyber_fairness_data]

```
static struct blkcg_policy_data *kyber_cpd_alloc(gfp_t gfp)
{
        struct kyber_fairness_data *kfd;
        kfd = kzalloc(sizeof(*kfd), gfp);
        if (!kfd)
                return NULL;
        return &kfd->pd;
}
```

```
static void kyber_cpd_init(struct blkcg_policy_data *cpd)
{
        struct kyber_fairness_data *kfd = cpd_to_kfd(cpd);
        kfd->weight = cgroup_subsys_on_dfl(io_cgrp_subsys) ?
                CGROUP_WEIGHT_DFL : KYBER_WEIGHT_LEGACY_DFL;
}
```

```
static void kyber_cpd_free(struct blkcg_policy_data *cpd)
{
        kfree(cpd_to_kfd(cpd));
}
```

[그림 17. kyber_fairness_data 구조체 생성]

blkcg_policy_data 가 생성될 때 [그림 16]의 kyber_fairness_data 도 생성하기 위해 [그림 17]의 kyber_cpd_alloc(), kyber_cpd_init() 함수가 필요하다. kyber_cpd_alloc()은 Kyber Fairness 스케줄러에서 kyber_fairness_data 구조체를 cgroup 마다 생성하기 위한 함수이고, kyber_cpd_init()은 kyber_cpd_alloc()에서 생성된 kyber_fairness_data 구조체를 초기화하기 위한 함수이다. kyber_cpd_free()는 cgroup 설정이 변경되거나 서브시스템이 비활성화 됐을 때 사용되는 함수이다. 여기서 kyber_fairness_data 구조체의 메모리 할당을 해제하여야 한다.

Kyber Fairness 스케줄러는 블록 디바이스마다 request 를 관리하며 블록 디바이스는 하나의 request_queue 를 가진다. request_queue 와 관련 있는 cgroup 별 정보는 blkcg_gq(blkg)라는 구조체로 알아낼 수 있다. 다시 말해, blkg 는 request_queue 와 cgroup 을 연결시켜주는 구조체이다. blkg 구조체 또한 blkcg 에서의 blkcg_policy_data 와 비슷한 역할을 하는 blkg_policy_data 가

blkcg_gq 및 blkcg_policy 마다 존재한다. 따라서 blkg_policy_data 를 사용하면 블록 디바이스마다, 그리고 cgroup 마다 [그림 18]의 kyber_fairness 구조체를 생성할 수 있다.

```
struct kyber_fairness {
        struct blkg_policy_data pd;
        struct list_head kf_list;
        struct kyber_fairness_data *kfd;
        unsigned int id;
        unsigned int weight;
        u64 next_budget;
        s64 cur_budget;
        bool idle;
        spinlock_t lock;
};
```

[그림 18. kyber_fairness]

새로운 cgroup 이 생성되면 kyber_fairness 구조체는 블록 디바이스마다 생성된다. 이 때, kyber_fairness_data 구조체는 kyber_fairness 구조체 보다 먼저 생성된다.

```
static struct blkg_policy_data *kyber_pd_alloc(gfp_t gfp, int node)
{
        struct kyber_fairness *kf;
        kf = kzalloc_node(sizeof(*kf), gfp, node);
        if (!kf)
                return NULL;
        return &kf->pd;
}
```

```
static void kyber_pd_init(struct blkg_policy_data *pd)
{
        struct blkcg_gq *blkg = pd_to_blkg(pd);
        struct kyber_fairness_data *kfd = blkcg_to_kfd(blkg->blkcg);
        struct kyber_queue_data *kqd = blkg->q->elevator->elevator_data;
        struct kyber_fairness_global *kfg = kqd->kfg;
```

```
        struct kyber_fairness *kf = pd_to_kf(pd);

        /* kyber_fairness 초기화 */

        kf->kfd = kfd;

        kf->weight = kfd->weight;

        kf->next_budget = kfd->weight * KYBER_SCALE_FACTOR;

        kf->cur_budget = kf->next_budget;

        kf->id = ++kfg->nr_kf;

        kfg->kf_list[kf->id] = kf;

        kf->idle = true;

        spin_lock_init(&kf->lock);

}
```

```
static void kyber_pd_free(struct blkg_policy_data *pd)

{

        kfree(pd_to_kf(pd));

}
```

[그림 19. kyber_fairness 구조체 생성]

위의 함수들은 위에서 보았던 kyber_cpd_alloc(), kyber_cpd_init(), kyber_cpd_free() 함수들과 유사하다. kyber_pd_alloc() 함수로 kyber_fairness 를 생성하며 kyber_pd_init() 함수를 통해 kyber_fairness 의 변수들을 초기화한다. kyber_pd_free()는 cgroup 설정이 변경되거나 서브시스템이 비활성화 됐을 때 kyber_fairness 구조체의 메모리 할당을 해제한다.

```
static struct cftype kyber_blkg_files[] = {

{

        .name = "kyber.weight",

        .flags = CFTYPE_NOT_ON_ROOT,

        .seq_show = kyber_io_show_weight,

        .write = kyber_io_set_weight,

        },

        {} /* terminate */

};
```

```
static struct cftype kyber_blkcg_legacy_files[] = {
{
        .name = "kyber.weight",
        .flags = CFTYPE_NOT_ON_ROOT,
        .seq_show = kyber_io_show_weight,
        .write_u64 = kyber_io_set_weight_legacy,
        },
        {} /* terminate */
};
```

[그림 20. kyber_blkg_files, kyber_blkcg_legacy_files]

kyber_blkg_files, kyber_blkcg_legacy_files 는 시스템 파일을 통해 weight 를 입력 받을 수 있는 함수(write, write_u64)가 존재하는 cftype 구조체이다. weight 를 최초로 입력할 때에는 kyber_blkg_files 가 사용되고 weight 를 수정할 때에는 kyber_blkcg_legacy_files 가 사용된다.

```
oslab4@oslab-server4:~$ ls /sys/fs/cgroup/blkio/highest
blkio.bfq.io_service_bytes                blkio.throttle.io_serviced_recursive
blkio.bfq.io_service_bytes_recursive      blkio.throttle.read_bps_device
blkio.bfq.io_serviced                     blkio.throttle.read_iops_device
blkio.bfq.io_serviced_recursive           blkio.throttle.write_bps_device
blkio.bfq.weight                          blkio.throttle.write_iops_device
blkio.kyber.weight                        cgroup.clone_children
blkio.reset_stats                         cgroup.procs
blkio.throttle.io_service_bytes           notify_on_release
blkio.throttle.io_service_bytes_recursive tasks
blkio.throttle.io_serviced
```

[그림 21. blkio.kyber_weight]

아래는 사용자가 [그림 21]의 blkio.kyber_weight 파일에 weight 를 입력할 때 호출되는 함수들이다.

```
static int kyber_io_set_weight_legacy(struct cgroup_subsys_state *css,
                struct cftype *cftype,
                u64 val)
{
        struct blkcg *blkcg = css_to_blkcg(css);
        struct blkcg_gq *blkg;
        struct kyber_fairness_data *kfd = blkcg_to_kfd(blkcg);
```

```
        int ret = -ERANGE;

        if (val < KYBER_MIN_WEIGHT || val > KYBER_MAX_WEIGHT)

                return ret;

        ret = 0;

        spin_lock_irq(&blkcg->lock);

        /* kyber_fairness_data 의 weight 를 변경 */
        kfd->weight = (unsigned int)val;

        /* kyber_fairness 의 weight 를 변경 */
        hlist_for_each_entry(blkg, &blkcg->blkg_list, blkcg_node) {

                struct kyber_fairness *kf = blkg_to_kf(blkg);

                if (kf) {

                        spin_lock(&kf->lock);

                        kf->weight = kfd->weight;

                        spin_unlock(&kf->lock);

                }

        }

        spin_unlock_irq(&blkcg->lock);

        return ret;

}
```

```
static ssize_t kyber_io_set_weight(struct kernfs_open_file *of,

                char *buf, size_t nbytes,

                loff_t off)

{

        u64 weight;

        /* First unsigned long found in the file is used */

        int ret = kstrtoull(strim(buf), 0, &weight);

        if (ret)

                return ret;

        ret = kyber_io_set_weight_legacy(of_css(of), NULL, weight);

        return ret ?: nbytes;

}
```

```
static int kyber_io_show_weight(struct seq_file *sf, void *v)
{
        struct blkcg *blkcg = css_to_blkcg(seq_css(sf));
        struct kyber_fairness_data *kfd = blkcg_to_kfd(blkcg);
        unsigned int val = 0;
        if (blkcg)
                val = kfd->weight;
        seq_printf(sf, "%u\n", val);
        return 0;
}
```

[그림 22. Weight 입력]

weight 를 최초로 입력할 때 불리는 함수는 kyber_io_set_weight()이고 weight 를 수정할 때 불리는 함수는 kyber_io_set_weight_legacy()이다. weight 값을 사용자가 출력할 때 불리는 함수는 kyber_io_show_weight()이다.

정리하자면 blkio 의 weight 정보는 파일시스템(kernfs_open_file) → cgroup 서브시스템(cgroup_subsys_state) → blkcg → blkcg_policy_data → kyber_fairness_data 의 경로로 구조체가 메모리상에 할당된 위치를 알아내어 cgroup weight 정보를 kyber_fairness_data 의 weight 에 저장할 수 있다.

```
static int kyber_init_sched(struct request_queue *q, struct elevator_type *e)
{
        struct kyber_queue_data *kqd;
        struct elevator_queue *eq;
        ktime_t ktime = ktime_set(0, KYBER_REFILL_TIME * NSEC_PER_MSEC);
        int ret;
        eq = elevator_alloc(q, e);
        /*...*/
        eq->elevator_data = kqd;
        q->elevator = eq;
        ret = blkcg_activate_policy(q, &blkcg_policy_kyber);
```

```
        if (ret) {
                kfree(kqd);
                kobject_put(&eq->kobj);
                return ret;
        }
        /*...*/
}
```

```
static void kyber_exit_sched(struct elevator_queue *e)
{
        struct kyber_queue_data *kqd = e->elevator_data;
        /*...*/
        blkcg_deactivate_policy(kqd->q, &blkcg_policy_kyber);
        kfree(kqd);
}
```

[그림 23. kyber_init_shced(), kyber_exit_sched()]

blkcg_policy_kyber 는 kyber_init_sched()에서 활성화할 수 있고 kyber_exit_sched()에서 비활성화 할 수 있다. kyber_init_sched()는 커널에서 I/O 스케줄러를 Kyber Fairness 스케줄러로 변경할 때 호출되며 Kyber_exit_sched()는 I/O 스케줄러를 Kyber Fairness 스케줄러에서 다른 스케줄러로 변경할 때 호출된다.

```
static int __init kyber_init(void)
{
        int ret;
        ret = blkcg_policy_register(&blkcg_policy_kyber);
        if (ret)
                return ret;
        ret = elv_register(&kyber_sched);
        if (ret) {
                blkcg_policy_unregister(&blkcg_policy_kyber);
                return ret;
        }
```

```
        return 0;
}
```

```
static void __exit kyber_exit(void)
{
        elv_unregister(&kyber_sched);
        blkcg_policy_unregister(&blkcg_policy_kyber);
}
```

[그림 24. kyber_init(), kyber_exit()]

kyber_init() 함수는 커널 부팅 시 호출되는 함수이다. 여기서 우리가 만든 blkcg_policy_kyber 를 등록함으로써 스케줄러가 활성화되면 즉시 사용할 수 있게 된다. 반대로 kyber_exit() 함수는 커널 종료시 불리며 blkcg_policy_kyber 를 등록 해제한다.

2) Insert request

```
static int bio_to_css_id(struct bio *bio)
{
        return blkg_to_kf(bio->bi_blkg)->id;
}
```

```
static void kyber_insert_requests(struct blk_mq_hw_ctx *hctx,
                struct list_head *rq_list, bool at_head)
{
        list_for_each_entry_safe(rq, next, rq_list, queuelist) {
                int id = bio_to_css_id(rq->bio);
                struct list_head *head = &kcq->rq_list[id][sched_domain];
                if (id > kfg->nr_kf)
                        kyber_kf_lookup_create(hctx->queue);
                kf = kfg->kf_list[id];
                spin_lock(&kcq->lock);
                if (at_head)
                        list_move(&rq->queuelist, head);
                else
                        list_move_tail(&rq->queuelist, head);
```

```
                sbitmap_set_bit(&khd->kcq_map[id][sched_domain],
                                rq->mq_ctx->index_hw[hctx->type]);
                spin_unlock(&kcq->lock);
        }
}
```

```
static void kyber_kf_lookup_create(struct request_queue *q)
{
        rcu_read_lock();
        while (1) {
                css = css_from_id(id++, &io_cgrp_subsys);
                if (!css)
                        break;
                blkg = blkg_lookup_create(css_to_blkcg(css), q);
                kf = blkg_to_kf(blkg);
        }
        rcu_read_unlock();
}
```

[그림 25. kyber_insert_requests(), kyber_kf_lookup_create()]

S/W 큐에 request 를 보내기 위해서 kyber_insert_requests()가 호출된다. Kyber Fairness 스케줄러는 cgroup 별로 S/W 큐를 가지고 있기 때문에, bio_to_css_id() 함수를 통해 request 의 cgroup id 를 읽어 일치하는 S/W 큐를 선택하고 insert 해 주어야 한다. 만약 request 의 cgroup id 가 kyber_fairness 구조체의 개수보다 크다면 kyber_kf_lookup_create()를 호출하여 kyber_fairness 구조체를 생성한다. Insert 가 완료되면 sbitmap_set_bit()을 통하여 request 가 있음을 표시한다.

3) Dispatch request

```
    static int kyber_is_active(int id, struct blk_mq_hw_ctx *hctx)
    {
          for (domain = 0; domain < KYBER_NUM_DOMAINS; domain++) {
                    if (!list_empty_careful(&khd->rqs[id][domain]) ||          // H/W 큐 확인
                              sbitmap_any_bit_set(&khd->kcq_map[id][domain]))  // S/W 큐 확인
                                        return 1;
          }
          return 0;
    }
```

```
static bool kyber_has_work(struct blk_mq_hw_ctx *hctx)
{
          struct request_queue *q = hctx->queue;
          struct kyber_queue_data *kqd = q->elevator->elevator_data;
          struct kyber_fairness_global *kfg = kqd->kfg;
          for (id = 1; id <= kfg->nr_kf; id++) {
                    if (kyber_is_active(id, hctx))
                              return true;
          }
          return false;
}
```

```
  static int kyber_choose_cgroup(struct blk_mq_hw_ctx *hctx)
  {
          for (id = kf->id; id <= kfg->nr_kf; id++) {
                    kf = kfg->kf_list[id];
                    spin_lock(&kf->lock);
          /* 활성화 된 kyber_fairness 가 있을 때 */
                    if (!kf->idle && kf->cur_budget > 0)
                              throttle = false;
          /* S/W 큐 또는 H/W 큐에 request 가 있을 때 */
                    if (kyber_is_active(id, hctx)) {
```

```
                        if (kf->cur_budget > 0) {
                                khd->cur_kf = kf;
                                spin_unlock(&kf->lock);
                                return kf->id;
                        } else {
        /* request 는 있으나 budget 이 없는 경우 */
                                hasreq = -1;
                        }
                }
                spin_unlock(&kf->lock);
        }
        /* 모든 cgrouop 의 budget 이 0 보다 작을 때 */
        if (throttle) {
                 (hrtimer_try_to_cancel(&kfg->timer) >= 0) {
                        kyber_refill_budget(q);
                        hrtimer_start(&kfg->timer,ktime,HRTIMER_MODE_REL);
                }
        }
        return hasreq;
}
```

```
static struct request *kyber_dispatch_request(struct blk_mq_hw_ctx *hctx)
{
        /* 쓰로틀링 중인 kyber_fairness 가 있다면 while 문을 이용 budget 을 지속적으로 확인 */
        do {
                cgroup_id = kyber_choose_cgroup(hctx);
                if (!cgroup_id)
                        return NULL;
        } while (cgroup_id < 0);
        rq = kyber_dispatch_cur_domain(kqd, khd, hctx, cgroup_id);
}
```

[그림 26. kyber_is_active(), kyber_choose_cgroup(), kyber_dispatch_request()]

H/W 큐에서 request 를 dispatch 하기 위해 kyber_dispatch_request() 가 호출된다. 이 때, dispatch 할 H/W 큐를 선택해야 하고 kyber_choose_cgroup() 함수를 통해 request 를 dispatch 할 어떤 cgroup 의 H/W 큐를 선택한다. kyber_choose_cgroup() 함수에서 cgroup 을 선택할 때에 고려하는 사항이 두가지가 있는데 첫번째는 해당 cgroup 의 S/W 큐 또는 H/W 큐에 request 가 존재하는지 보는 것이고 나머지 하나는 budget 이 0 이상인지 확인하는 것이다.

첫번째로 request 가 있는지 확인하는 작업을 하기 위해서 kyber_is_active() 함수를 호출한다. kyber_is_active()는 S/W 큐는 sbitmap, H/W 큐는 list 를 확인함으로써 해당 cgroup 에 작업이 있는지 없는지 판단한다. 하지만 만약 request 는 있는데 budget 이 없어 처리하지 못하는 cgroup 이 있다면 NULL 을 리턴하지않고, do-while 문을 통하여 해당 cgroup 의 budget 이 리필 될 때까지 kyber_choose_cgroup()을 호출한다. 그 이유는 초기 설계의 실패에서 보았듯이 프로세스는 하나의 cgroup 에 대한 request 를 주로 처리하기 때문에 함수를 종료하지 않고 budget 이 리필 되기를 기다렸다가 신속하게 처리한다.

```
static struct request *
kyber_dispatch_cur_domain(struct kyber_queue_data *kqd,
                  struct kyber_hctx_data *khd,
                  struct blk_mq_hw_ctx *hctx,
                  int cgroup_id)
{
        kf = kf_from_rq(rq);
        if (kf) {
        /* 해당 kyber_fainress 의 budget 을 request 의 크기만큼 감소시킴 */
                  spin_lock(&kf->lock);
                  kf->cur_budget -= blk_rq_sectors(rq);
                  spin_unlock(&kf->lock);
        }
        return rq;
}
```

[그림 27. kyber_dispatch_cur_domain()]

디스패치할 어떤 cgroup 의 H/W 큐가 선택되었다면, 다음은 기존의 kyber 스케줄러처럼 domain 을 선택해야한다. Domain 을 선택하기 위해서 먼저 이전에 디스패치 되었던 도메인부터 확인하고 라운드로빈 방식으로 나머지 도메인도 확인한다. 도메인을 확인하기 위해서 스케줄러는 [그림 27]의 kyber_dispatch_cur_domain() 함수를 호출한다. 이 함수에서 request 가 선택된다는 것의 의미는 budget 이 0 이상이고 디스패치 된다는 의미이기 때문에 해당 cgroup 의 budget 을 request 의 크기만큼 감소시킨다.

4) Budget 리필

```
static int kyber_init_sched(struct request_queue *q, struct elevator_type *e)
{
        /* budget 을 리필하는 쓰레드 생성 */
        kqd->kfg->timer_thread =
                kthread_run(kyber_refill_thread_fn, kqd->kfg, "refill thread");
        /* 타임아웃 설정 */
        hrtimer_start(&kqd->kfg->timer, ktime, HRTIMER_MODE_REL);
        return 0;
}
```

```
static int kyber_refill_thread_fn(void *arg)
{
        struct kyber_fairness_global *kfg = arg;
        while (!kthread_should_stop()) {
                kyber_refill_budget(kfg->q);
                set_current_state(TASK_INTERRUPTIBLE);
                io_schedule();
        }
        return 0;
}
```

[그림 28. Kyber_refill_thread]

쓰레드를 통하여 타임아웃마다 budget 을 리필해주도록 하기 위해서 kyber_init_sched()에서 kthread_run()을 통해 쓰레드를 생성하고 쓰레드의 콜백함수로 kyber_refill_thread_fn()을

설정해준다. 그리고 hrtimer_start() 함수를 통하여 리필 쓰레드를 타임아웃마다 깨우도록 설정한다. 여기서 타임아웃에 budget 을 리필 하는 이유는 모든 cgroup 의 budget 이 소모될 때만 리필하는 방식으로 구현하면 budget 을 사용하지 않는 cgroup 이 있을 때 리필이 되지 않으므로 타임아웃을 설정해 두었다.

3. 구현 이슈

1) Lock mechanism - 산업체 멘토링 결과 반영

스케줄러에서 request 를 디스패치하거나, 리필 쓰레드를 실행시킬지 결정하기 위해서 모든 cgroup 들의 budget 을 확인해야 한다. 따라서 스케줄러는 cgroup list 를 관리하고 있어야 하는데, 멀티-큐 블록 레이어는 다수의 H/W 큐를 가지고 있으면서 동시에 다수의 S/W 큐를 가지고 있다. 즉, 다수의 프로세서가 cgroup 을 접근해야 한다는 의미이며 lock 을 통한 동기화가 필요하다.

초기에 본 팀은 cgroup list 를 리눅스 커널의 list 를 사용하여 구현하였으나, list 의 경우 lock 을 통해 list 를 동기화해야 하는 문제가 있었다. 따라서, list 를 사용하지 않고 array 를 생성해 cgroup 의 id 를 사용하여 array 에 접근하였다. cgroup 의 총 개수는 kyber_fairness_global 구조체에 nr_kf 로 저장하고 있으며, blkio subsystem 에 새롭게 그룹을 생성하거나 스케줄러가 초기화될 때 pd_init() 함수가 실행되면 해당 cgroup 의 id 를 할당 받음과 동시에 nr_kf 가 1 증가된다.

```
static void kyber_pd_init(struct blkg_policy_data *pd)
{
        kf->id = ++kfg->nr_kf;
        kfg->kf_list[kf->id] = kf;
}
```

[그림 29. Array 를 통한 cgroup 관리]

또한 for 문을 통해 kyber_fairness 구조체를 탐색하는 것은 확장성 이슈가 존재할 수 있다. 예를 들어 cgroup 이 100 개 존재한다면 최악의 경우 for 문으로 항상 100 번의 루프가 필요하다. 따라서, 항상 1 번 kyber_fairness 부터 탐색하는 것이 아니라 마지막으로 dispatch 한 kyber_fairness 의 id 를 저장해두고 다음번에 해당 id 부터 탐색을 시작한다. 그 이유는 위에서 언급했듯이 하나의 프로세서가

처리하는 프로세스들은 하나의 cgroup 에 속해 있을 가능성이 높기 때문에 이런 지역성에 기초하여 id 를 저장해 둔다. 이를 통해 확장성 이슈를 해결할 수 있다.

2) 최적의 budget 도출 - 산업체 멘토링 결과 반영

n: 총 cgroup 의 수, C_x: cgroup x, w_x : cgroup x의 weight

T_{spend} : 마지막 리필 이후로부터 흐른 시간

B^x_{next} : cgroup x가 새로 할당 받는 budget

B^x_{used} : cgroup x가 T_{spend} 동안 사용한 budget

$B^x_{remainder}$: Idle 이 아닌 cgroup x가 T_{spend} 동안 사용하고 남은 budget

$$\therefore B^k_{next}\ (1 \le k \le n,\ C_k \neq idle)$$
$$= \left(\sum_{i=1}^{n} B^i_{used} \times \frac{100ms}{T_{spend}} - \sum_{i=1}^{n} B^i_{remainder} \right) \times \frac{w_k}{\sum_{i=1}^{n} w_i (C_i \neq idle)} + B^k_{remainder}$$

[그림 30. budget 계산 수식]

스케줄러의 성능을 위해선 블록 디바이스의 bandwidth 에 상응하는 budget 을 할당하여 리필이 최소한으로 발생하는 것이 바람직하다. Kyber Fairness 스케줄러는 bandwidth 를 예측하는 방법으로 마지막 리필 이후로부터 흐른 시간을 측정하고 그동안 처리된 소모된 budget 의 총합을 타임아웃(100ms)기준으로 환산하여 budget 을 계산한다. 예를 들어 모든 cgroup 들이 budget 을 빨리 소모해 리필 쓰레드가 타임아웃보다 일찍 깨어난다면, 타임아웃까지 request 를 내려 보냈을 때 이전에 할당된 budget 보다 많이 소모할 수 있다는 뜻이므로 전체적인 budget 을 증가시켜준다.

Budget 리필에서 고려되어야 할 점이 또 하나 있는데 바로 remainder budget 이다. Budget 을 리필하는 시점에서 예상 bandwidth 를 weight 에 비례하여 나누어 주는 것은 전체적인 관점에서 보면 fairness 를 해칠 수 있다. 그러므로 budget 을 리필할 때 계산된 budget 에서 remainder budget 을 제외한 나머지를 weight 에 비례하여 할당해 주고 remainder budget 은 cgroup 이 그대로 가져가도록 설계하였다.

3) Idle 상태 정의

```
static void kyber_refill_budget(struct request_queue *q)
{
        for (id = 1; id <= kfg->nr_kf; id++) {
                kf = kfg->kf_list[id];
                if (kf->cur_budget != kf->next_budget) {
                        active_weight += kf->weight;
                } else {
                        kf->idle = true;
                        kf->next_budget = kf->weight * KYBER_SCALE_FACTOR;
                        kf->cur_budget = kf->next_budget;
                }
}
```

[그림 31. Idle cgroup]

어떤 cgroup 은 블록 I/O 작업을 하지 않을 수도 있다. 이런 cgroup 은 idle 상태로 정의하여 budget 리필을 할 때 budget 을 가져가지 못하도록 해야 한다. S/W 큐와 H/W 큐를 검사하여 idle 인지 판단하는 방법은 큐를 다 확인해야 한다는 오버헤드와 블록 디바이스의 처리속도가 빠르기 때문에 정확하지 못했다. 따라서, 리필 쓰레드가 실행되었을 때 어떤 cgroup 이 이전 리필때의 budget 과 현재 budget 의 차이가 없다면 idle 상태로 설정하였다. Idle 상태의 해제는 request insert 시 해당 cgroup 의 idle 을 해제한다.

```
static void kyber_insert_requests(struct blk_mq_hw_ctx *hctx,
                struct list_head *rq_list, bool at_head)
{
                spin_lock(&kf->lock);
                kf->idle = false;
                spin_unlock(&kf->lock);
}
```

[그림 32. Idle 상태의 해제]

Ⅳ. 성능 측정

1. 실험 환경

CPU	Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz, 32 코어
메모리	32G
NVMe SSD	Intel DC P4500 2TB NVMe PCIe 3.0 3D TLC 2.5'' 1DWPD FW13D
OS	Ubuntu 18.04, Linux Kernel 5.2.0 + Kyber Fairness Scheduler

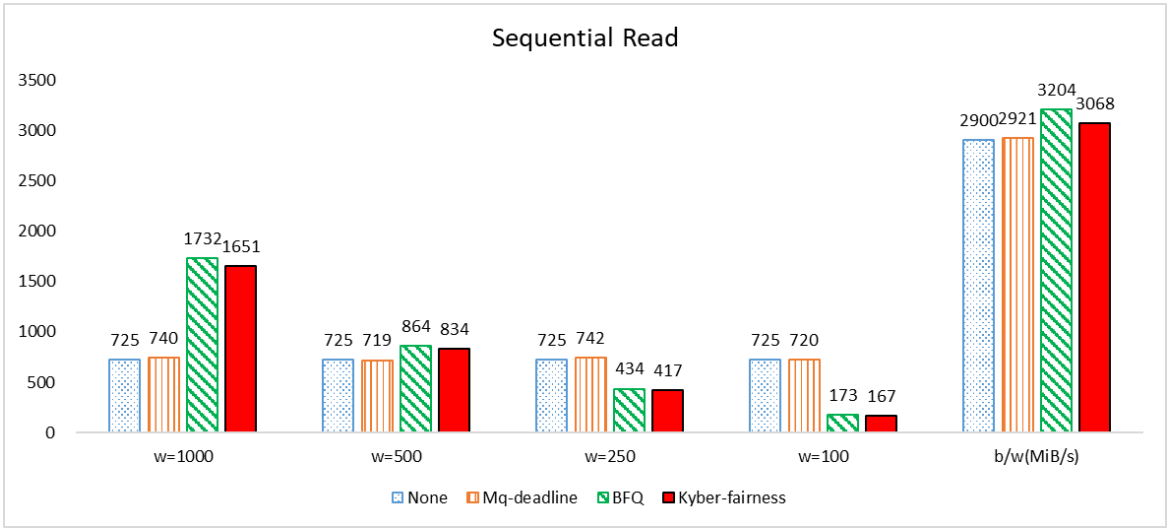
[그림 33. 시스템 사양]

Ioengine	libaio
Iodepth	32
Time_based	
Runtime	60
Direct	1
Sequential-read	Block size = 1m
Sequential-write	
Random-read	Block size = 4k
Random-write	
Random-read/write	Block size = 4k, Read / Write = 50% / 50%
Random-read/write	

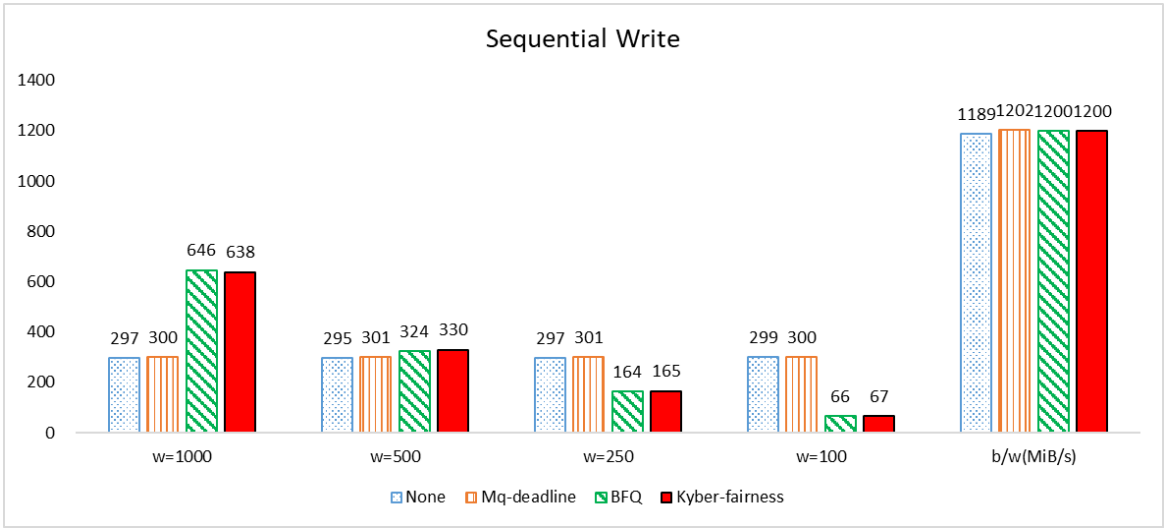
[그림 34. Fio global 설정]

각 워크로드는 총 10 번 실행되며, 평균 값을 실험 결과로 설정하였다.

2. 실험 결과



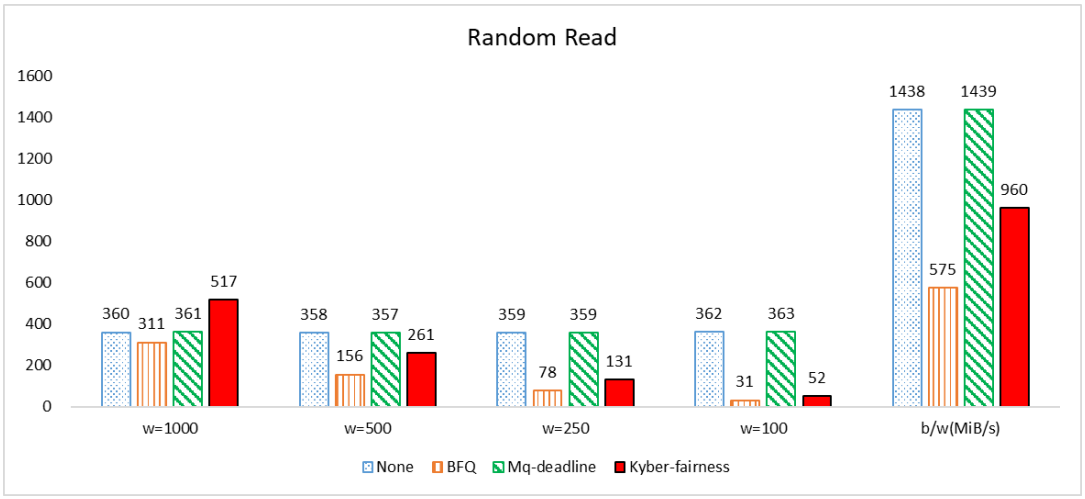
(a) Sequential Read



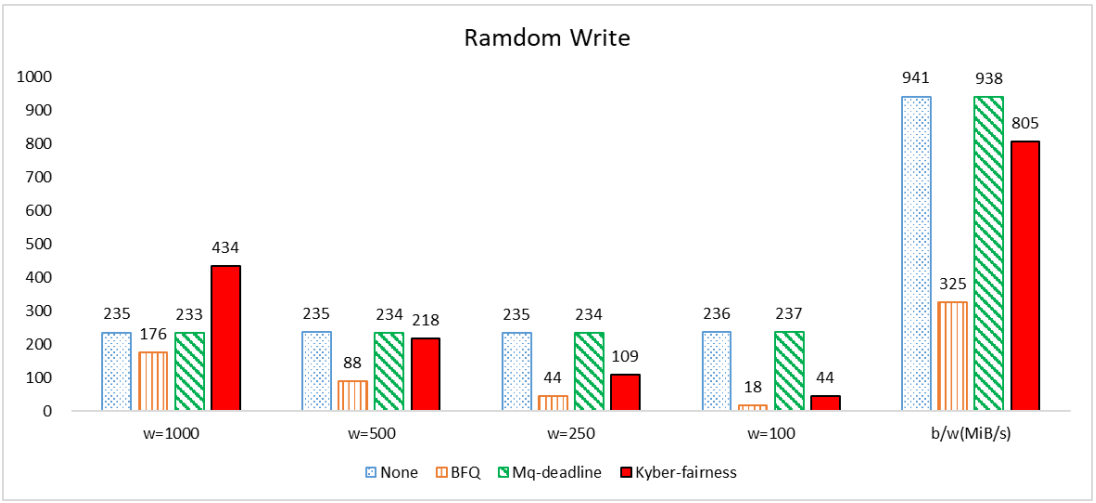
(b) Sequential Write

[그림 35. Sequential request]

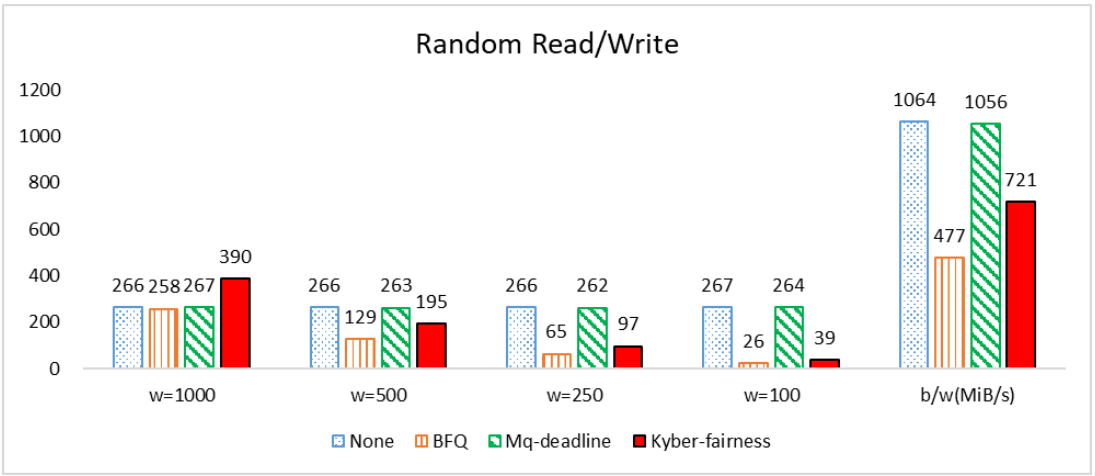
Sequential read 에서는 BFQ 스케줄러가 가장 우수한 성능을 보였고, 그 다음으로 Kyber Fairness 스케줄러가 우수한 성능을 보였다. 또한 두 스케줄러 모두 1000:500:250:100 의 weight 에 따라 fairness 를 잘 보장하고 있음을 알 수 있다.



(a) Random Read



(b) Random Write



(c) Mixed Random Read/Write (50% / 50%)

[그림 36. Random request]

Kyber Fairness 스케줄러는 Random request 들에 대해서 BFQ 스케줄러보다 좋은 성능을 보이고 있다. 심지어 random write 에서는 bandwidth 의 차이가 2 배이상 나는 것을 확인할 수 있다.

BFQ 스케줄러를 제외한 다른 스케줄러들과 비교해 보았을 때 random write 에서는 큰 차이가 없었지만 random read 에서는 총 bandwidth 차이가 많이 발생했다. Random request 에서 총 bandwidth 감소한 원인은 스케줄러가 request 를 처리할 때마다 추가적인 작업을 하는데, sequential request 일 때보다 처리할 request 가 많아졌기 때문에 발생한다. 하지만, 전체 bandwidth 가 감소하였다고 하더라도 weight 가 1000 인 cgroup 의 bandwidth 는 다른 스케줄러보다 더 높은 bandwidth 를 보여주고 있다. 따라서 Kyber Fairness 스케줄러를 이용해서 cgroup 별로 구분하여 사용하는 것이 의미 있다고 할 수 있다.

[그림 36.c]는 random read request 와 random write request 가 50:50 으로 혼합돼 있는 워크로드의 실험 결과를 나타내고 있는 그래프이다. read 와 write 가 섞인 워크로드의 경우에도 문제없이 cgroup 의 weight 에 따라 fairness 를 보장하고 있는 모습을 관찰할 수 있었다.

V. 결론

1. 활용 방안

우선 Kyber Fairness 스케줄러는 개요에서 언급했듯이 SLA 의 보장 및 비즈니스 목적으로 클라우드 서비스에서 쓰일 수 있다. 클라우드 서비스 제공자들이 사용자들에게 더 많은 스토리지 옵션을 제공하는 것이 가능해짐으로써 제공자는 판매전략을 다양하게 설정할 수 있고, 사용자는 자신의 필요에 맞게 가격과 성능을 조절하여 자신의 인스턴스를 커스터마이징 할 수 있게 된다.

데이터 센터가 아니더라도 일반 사용자들에게도 유용하게 사용될 수 있다. 예를 들어 동영상을 보면서 다른 작업을 할 때 동영상을 제외한 작업이 I/O 를 많이 소모한다면, 동영상을 불러올 때 bandwidth 가 부족해져서 동영상이 지연될 수 있다. 이 때, 동영상을 실행하는 프로세스를 높은 weight 의 cgroup 에 추가한다면 충분한 bandwidth 를 보장받아 동영상의 끊김없이 작업을 진행할 수 있다.

마지막으로 자율주행자동차에 사용될 수 있다. 센서 정보를 종합하여 유용한 정보를 도출하는 프로세싱 하거나, 경로를 생성하거나, black-box 데이터를 기록하는 것 등과 같은 많은 메모리 서브시스템들이 자율주행자동차에 필수적인 요소들이다 [7]. 예를 들어 Kyber Fairness 스케줄러를 자율주행자동차에 적용시켜 본다면, 만약 사고가 날 것 같은 상황에서 black-box 데이터를 기록하는 프로세스를 weight 가 높은 cgroup 에 포함시켜 고화질로 영상을 남기는 것이 가능해진다.

2. 향후 과제

하지만 현재 Kyber Fairness 스케줄러의 다음 budget 을 구하는 알고리즘에는 이전 사용량에 기반한다는 문제점이 있다. 문제가 되는 워크로드를 다음과 같이 생각 해볼 수 있다. 만약 90ms 마다 작은 크기의 블록 I/O 요청을 보내는 cgroup 이 있다고 하면 해당 cgroup 에 의해서 전체적인 throughput 이 낮아지는 결과를 낳을 수 있다. 따라서, idle 한 cgroup 을 찾는 방법을 좀 더 개선하거나 다음 budget 을 계산하는 알고리즘을 개선하여 어느 워크로드에서도 적용할 수 있는 스케줄러를 구현해야 할 것이다.

VI. 역할 분담 및 개발 일정

1. 역할 분담

이름	역할 분담	세부 업무
	Kyber 스케줄러에 fairness 와 관련된 구조체 생성 및 알고리즘 구현	리필 쓰레드 및 타이머 생성
		Kyber_is_active 및 Has_work 함수 구현
		Insert 및 Dispatch 함수 수정
	최적의 Budget 을 계산할 수 있는 알고리즘 설계	리필 함수 구현
		kyber_fairness 의 Idle 상태 정의
		디바이스의 bandwidth 예측
	cgroup 의 서브시스템인 blkio 와 연동	BFQ 스케줄러 분석
		blkcg_policy 구조체 완성
		kyber_fairness 구조체 생성 및 할당

2. 개발 일정

6 월				7 월					8 월					9 월				
2 주	3 주	4 주	5 주	1 주	2 주	3 주	4 주	5 주	1 주	2 주	3 주	4 주	5 주	1 주	2 주	3 주	4 주	5 주
리눅스 커널 스터디																		
				스케줄러 인터페이스 작성														
				blkio 와의 연동을 위한 스케줄링 구조체 생성														
				적정 Budget 설정 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		

참고 문헌

[1] 안성원, “클라우드 가상화 기술의 변화,” *Issue Report,* p. 17, 10 12 2018.

[2] J. Axboe, “Flexible I/O tester,” [온라인]. Available: https://github.com/axboe/fio.

[3] M. Bjørling, “Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems,” *SYSTOR*, Haifa, Israel, 2013.

[4] R. Rosen, “www.haifux.org,” 5 2013. [온라인]. Available: http://www.haifux.org/lectures/299/netLec7.pdf.

[5] F. C. Paolo Valenta, “High Throughput Disk Scheduling with Fair Bandwidth Distribution,” *IEEE*, 2010.

[6] H. Z. Jon C.R. Bennett, “WF2Q:Worst-case Fair Weighted Fair Queueing,” *INFOCOM*, San Francisco, California, 1996.

[7] micron, “Micron Insight,” micron, [온라인]. Available: https://www.micron.com/insight/on-the-road-to-full-autonomy-self-driving-cars-will-rely-on-ai-and-innovative-memory.

[8] S. Ahn, “Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems,” *HotStorage*, Denver, 2016.