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

2019년 7월 29일

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.

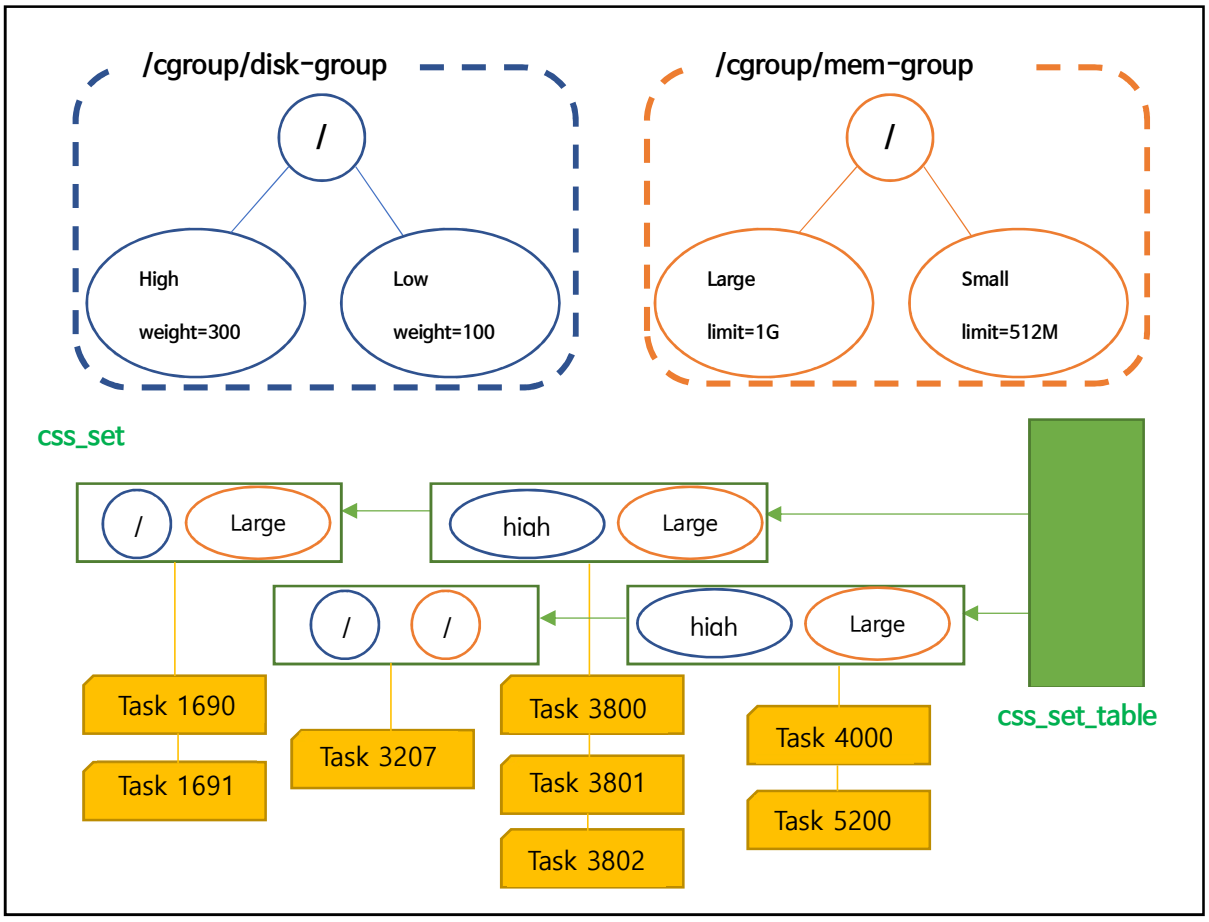
그룹	유형	vCPUs	메모리 (GiB)	네트워크 성능
General purpose	t2.nano	1	0.5	낮음에서 중간
General purpose	t2.micro 프리 티어 사용 가능	1	1	낮음에서 중간
General purpose	t2.small	1	2	낮음에서 중간
General purpose	t2.medium	2	4	낮음에서 중간
General purpose	t2.large	2	8	낮음에서 중간
General purpose	t2.xlarge	4	16	보통
General purpose	t2.2xlarge	8	32	보통

[그림 1. Amazon EC2 인스턴스 유형 선택 화면]

이러한 SLA 를 지키기 위해서는 성능 분배의 fairness 를 보장하는 것이 필요하다. 클라우드 기반 시스템에서는 많은 서비스 사용자가 존재하고, 이 서비스 사용자들이 서버가 제공하는 기능들을 사용하기 위해 많은 요청을 보낸다. Fairness 를 보장하지 못하는 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 분배의 fairness 를 보장함으로써 서비스 사용자들이 SLA 를 통해 부여받은 리소스만을 사용하도록 할 필요가 있다.

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

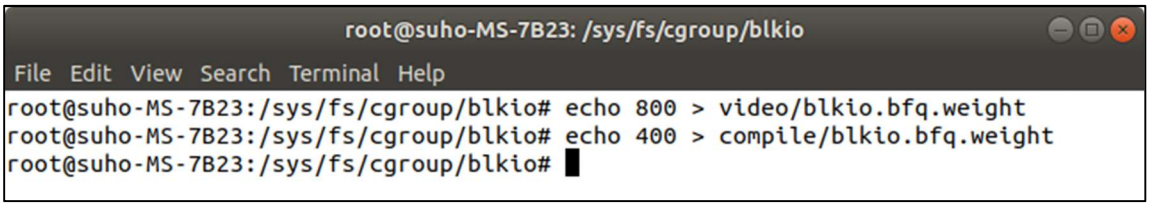
[그림 1]은 아마존 웹 서비스(AWS)에서 새로운 EC2 인스턴스를 생성하는 그림이다. CPU 와 메모리 그리고 네트워크 성능 그룹별로 나누어 클라우드 서비스 사용자들은 이를 옵션별로 선택할 수 있다.



[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

1.2.1 실험환경

CPU	Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz
메모리	64G
NVMe SSD	Intel DC P4500 2TB NVMe PCIe 3.0 3D TLC 2.5'' 1DWPD FW13D
OS	Ubuntu 18.04, Linux Kernel 5.1.15

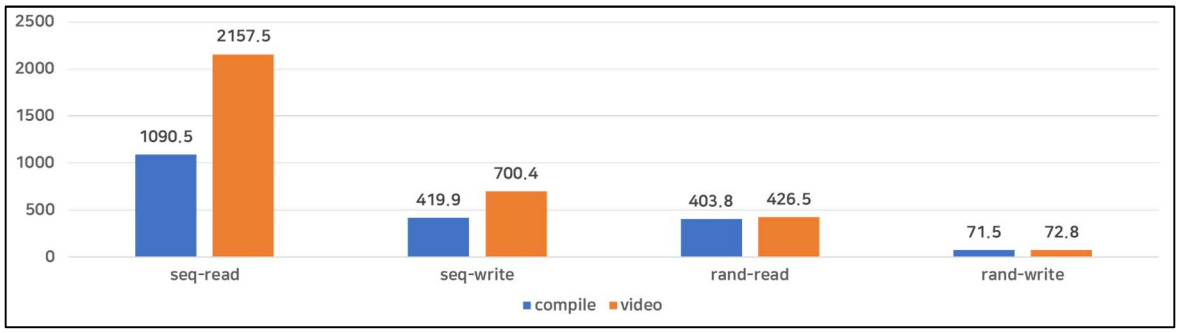
[표 1. 시스템 구성]

section	iodepth	Runtime	rw	bs	cgroup
Compile	32	20s	Seq-read	256k	Compile (weight=400)
			Seq-write		
			Rand-read	4k	
			Rand-write		
Video			Seq-read	256k	Video (weight=800)
			Seq-write		
			Rand-read	4k	
			Rand-write		

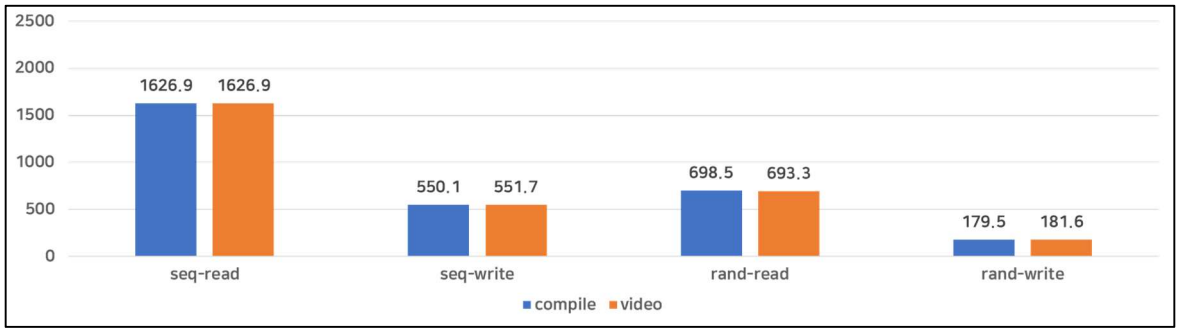
[표 2. fio 글로벌 설정 및 실험 환경]

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

1.2.2 BFQ/카이버 스케줄러 성능평가



(a) BFQ scheduler bandwidth (MB/s)



(b) Kyber scheduler bandwidth (MB/s)

[그림 5. fio results]

실험결과에서 크게 두가지를 주목해볼 수 있다. 첫번째로 주목할 점은 BFQ 스케줄러가 sequential read/write 의 경우에는 fairness 를 잘 보장하고 있지만, random read/write 에서는 fairness 를 보장 못하고 있다는 점이다. 이는 random I/O 작업이 요청될 때, BFQ 스케줄러가 기존의 'sector-domain fairness'를 그대로 사용할 경우 throughput 이 감소되고, latency 가 증가되기 때문에 싱글-큐 블록 레이어의 스케줄러인 CFQ I/O 스케줄러처럼 'time-domain fairness'로 전환한다. 하지만, 멀티-큐 블록 레이어에서는 시간에 따른 fairness 보장이 잘 되지 않기 때문에 [그림 5]의 (a)와 같은 결과를 보인 것으로 추측된다. 두번째로 주목할 점은 random read/write 에서 BFQ 스케줄러는 Kyber 스케줄러의 절반에 해당하는 성능을 보인다는 점이다. 실제 I/O 작업의 90%는 random I/O request 이기 때문에 random I/O 작업에서 위 그림과 같이 bandwidth 가 절반으로 감소하는 성능저하를 보이는 것은 시스템의 성능에 심각한 영향을 끼치는 문제이다.

1.3 과제 목표

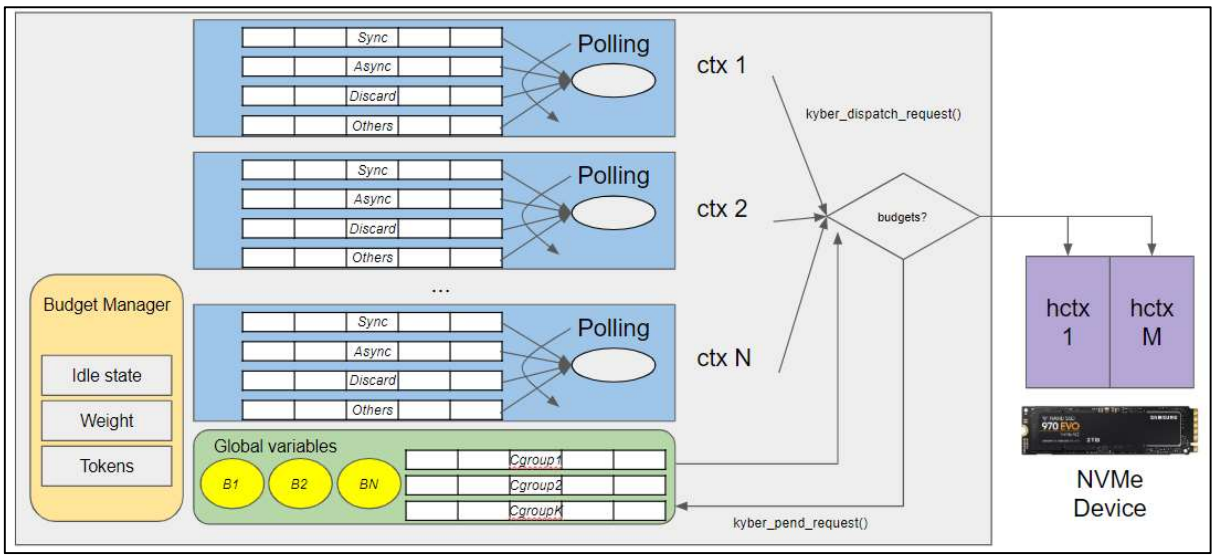
최근에는 고성능의 블록 디바이스들이 출시됨에 따라 싱글-큐 블록 레이어를 사용하는 것보다는 멀티-큐 블록 레이어를 많이 사용하고 있다. 하지만, 멀티-큐 블록 레이어에서 I/O 성능 분배의 fairness 를 보장하고자 하는 스케줄러는 BFQ 스케줄러뿐이며, 이 또한 아직 fairness 를 제대로

보장하지 못하고 있다. 또한, random I/O request 를 처리할 때 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.

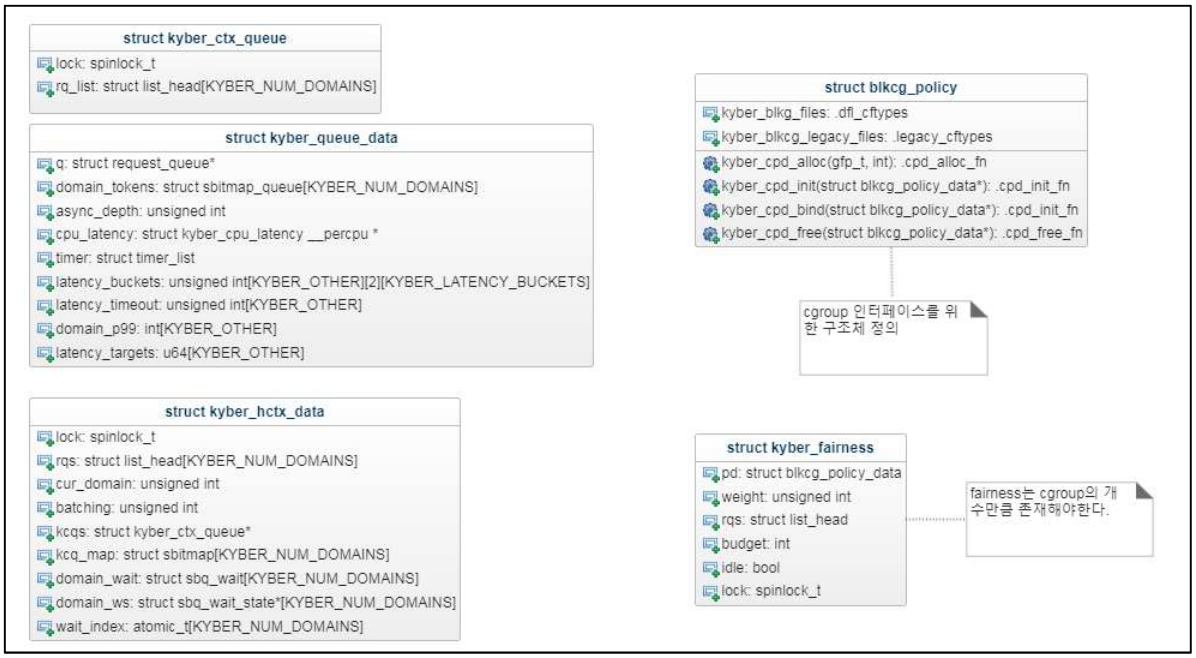
따라서 본 팀은 멀티-큐 블록 레이어에 존재하는 Kyber 스케줄러를 수정하여 I/O 성능 분배의 fairness 를 보장할 수 있도록 할 계획이다.

2 설계 및 구현

2.1 Kyber-fairness 스케줄러



[그림 6. Kyber-fairness 스케줄러 Overview]



[그림 7. Kyber 스케줄러 UML Diagram]

기존의 Kyber 스케줄러는 kyber_ctx_queue, kyber_queue_data, kyber_hctx_data 총 세가지의 구조체로 구성되어 있었다. 본 팀은 여기에 cgroup 인터페이스를 사용하기위한 blkcg_policy 구조체를 생성하여 구조체에서 필요로 하는 파일들과 콜백함수를 생성 후

맵핑시켜주었다. 그리고 cgroup 이 생성될 때 마다 해당 cgroup 에 1 대 1 로 맵핑되는 kyber_fairness 구조체를 만들어 kyber 스케줄러 내에서 fairness 정보를 가지고 있을 수 있도록 설계하였다.

```
struct kyber_fairness {
        struct blkcg_policy_data pd;
        unsigned int weight;
        struct list_head rqs;
        int budget;
        bool idle;
        spinlock_t lock;
};
```

[그림 8. kyber_fairness 구조체]

kyber_fairness 구조체는 총 6 개의 멤버변수로 구성되어있다.

- pd: cgroup 에 대한 정보를 가지고 있는 변수로 항상 첫번째 멤버변수로 존재해야한다.
- weight: cgroup 의 blkio.kyber.weight 파일에서 읽어오는 변수로 초기값은 100이고 최대 1000까지 설정 가능하다.
- rqs: request 를 dispatch 할 때, budget 이 부족하여 pending 중인 request 들을 가지고 있는 remainder queue 이다.
- budget: fairness 를 위해 해당 cgroup 이 소모할 수 있는 최대 sector 수를 가리키고 있다.
- idle: software queue 들에 해당 cgroup 의 request 가 없을 때 true 가 되는 변수이다.
- lock: kyber_fairness 구조체는 전역 변수로 존재하기 때문에 lock 을 이용하여 동기적으로 데이터를 관리하여야 한다.

```
static struct request *kyber_dispatch_request(struct blk_mq_hw_ctx *hctx)
{
        struct kyber_queue_data *kqd = hctx->queue->elevator->elevator_data;
        struct kyber_hctx_data *khd = hctx->sched_data;
        struct kyber_fairness *kf;
        struct request *rq;
        int ret;
        int i;
        spin_lock(&khd->lock);
        /*
         * First, if we are still entitled to batch, try to dispatch a request
         * from the batch.
         */
        if (khd->batching < kyber_batch_size[khd->cur_domain]) {
                rq = kyber_dispatch_cur_domain(kqd, khd, hctx);
                if (rq)
                        goto out;
        }
        /*
         * Either,
```

```
		 * 1. We were no longer entitled to a batch.
		 * 2. The domain we were batching didn't have any requests.
		 * 3. The domain we were batching was out of tokens.
		 *
		 * Start another batch. Note that this wraps back around to the original
		 * domain if no other domains have requests or tokens.
		 */
		khd->batching = 0;
		for (i = 0; i < KYBER_NUM_DOMAINS; i++) {
			if (khd->cur_domain == KYBER_NUM_DOMAINS - 1)
				khd->cur_domain = 0;
			else
				khd->cur_domain++;
			rq = kyber_dispatch_cur_domain(kqd, khd, hctx);
			if (rq)
				goto out;
		}
		rq = NULL;
	out:
		spin_unlock(&khd->lock);
		return rq;
}
```

[그림 9. kyber_dispatch_request()]

```
static struct request *
kyber_dispatch_cur_domain(struct kyber_queue_data *kqd,
			  struct kyber_hctx_data *khd,
			  struct blk_mq_hw_ctx *hctx)
{
		struct list_head *rqs;
		struct request *rq;
		struct blkcg_gq *blkg;
		struct kyber_fairness *kf;
		int ret;
		int nr;
		int i;
		rqs = &khd->rqs[khd->cur_domain];
		/*
		 * If we already have a flushed request, then we just need to get a
		 * token for it. Otherwise, if there are pending requests in the kcqs,
		 * flush the kcqs, but only if we can get a token. If not, we should
		 * leave the requests in the kcqs so that they can be merged. Note that
		 * khd->lock serializes the flushes, so if we observed any bit set in
		 * the kcq_map, we will always get a request.
		 */
		rq = list_first_entry_or_null(rqs, struct request, queuelist);
		if (rq) {
			nr = kyber_get_domain_token(kqd, khd, hctx);
			if (nr >= 0) {
				khd->batching++;
				rq_set_domain_token(rq, nr);
				list_del_init(&rq->queuelist);
				return rq;
			} else {
				trace_kyber_throttled(kqd->q,
						      kyber_domain_names[khd->cur_domain]);
			}
		} else if (sbitmap_any_bit_set(&khd->kcq_map[khd->cur_domain])) {
			nr = kyber_get_domain_token(kqd, khd, hctx);
			if (nr >= 0) {
				kyber_flush_busy_kcqs(khd, khd->cur_domain, rqs);
				rq = list_first_entry(rqs, struct request, queuelist);
```

```
                                khd->batching++;
                                rq_set_domain_token(rq, nr);
                                list_del_init(&rq->queuelist);
                                return rq;
                        } else {
                                trace_kyber_throttled(kqd->q,
                                                      kyber_domain_names[khd->cur_domain]);
                        }
                }
                /* There were either no pending requests or no tokens. */
                return NULL;
}
```

[그림 10. kyber_dispatch_cur_domain()]

```
static void kyber_completed_request(struct request *rq, u64 now)
{
        struct kyber_queue_data *kqd = rq->q->elevator->elevator_data;
        struct kyber_cpu_latency *cpu_latency;
        unsigned int sched_domain;
        u64 target;
        sched_domain = kyber_sched_domain(rq->cmd_flags);
        if (sched_domain == KYBER_OTHER)
                return;
        cpu_latency = get_cpu_ptr(kqd->cpu_latency);
        target = kqd->latency_targets[sched_domain];
        add_latency_sample(cpu_latency, sched_domain, KYBER_TOTAL_LATENCY,
                           target, now - rq->start_time_ns);
        add_latency_sample(cpu_latency, sched_domain, KYBER_IO_LATENCY, target,
                           now - rq->io_start_time_ns);
        put_cpu_ptr(kqd->cpu_latency);
        timer_reduce(&kqd->timer, jiffies + HZ / 10);
}
```

[그림 11. Kyber_completed_request()]

기존의 kyber 스케줄러는 kyber_dispatch_request() 함수를 사용하여 software queue 에 있던 request 들을 hardware queue 로 dispatch 한다. 이 때, kyber_dispatch_request() 함수는 어떤 request 를 dispatch 할지 정하는 함수인 kyber_dispatch_cur_domain() 함수를 호출하는데, 현재 도메인의 token 수를 계산하여 현재 도메인의 request 를 dispatch 할 수 있는지 확인하고, token 수가 충분하다면 request 를 현재 도메인의 queue 에서 선택하여 dispatch 하게 된다. 만약 token 수가 충분하지 않다면 다른 도메인에서 request 를 보내는 시도를 하게 된다.

request 의 처리가 끝나면 kyber_completed_request() 함수가 실행되어 latency 들을 크기에 따라 해당하는 bucket 에 추가하는 작업을 한다.

본 팀은 kyber_dispatch_cur_domain() 함수와 kyber_completed_request() 함수에 budget 을 계산하는 함수를 추가하여 fairness 를 보장할 수 있도록 할 예정이다.

2.2 cgroup

2.2.1 BFQ 스케줄러에서의 cgroup 활용

weight 정보는 cftype 이라는 구조체를 통해 시스템 파일 생성을 정의한 다음 수를 입력 받아 얻을 수 있다. 이는 blkcg_policy 구조체에서 정의할 수 있다.

BFQ 스케줄러는 cgroup 당 정보를 blkcg 라는 구조체로 알아낸다. blkcg 와 blkcg_policy 는 blkcg_policy_data 를 통해 cgroup 별, policy 별로 연결되어 있다. 즉 blkcg 내의 blkcg_policy_data 들은 blkcg_policy 의 plid(policy id)로 각각 접근할 수 있다. 이는 blkcg_policy 가 cgroup 이 생성될 때 마다 특정 변수를 생성하고 싶을 때 blkcg_policy_data 를 활용하여 해당 변수를 할당할 수 있게 된다. 그리고 request_queue 와 관련 있는 cgroup 당 정보를 blkcg_gq(blkg)라는 구조체로 알아낸다. 즉, blkcg_gq(blkg)는 request_queue 와 cgroup 을 연결시켜주는 구조체이다. 이것 또한 blkcg 에서 blkcg_policy_data 와 비슷한 역할을 하는 blkg_policy_data 가 blkcg_gq 및 blkcg_policy 마다 존재한다.

다음은 BFQ 스케줄러에서 blkcg_policy_data, blkcg_policy_data 를 사용하는 구조체이다.

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

```
struct bfq_group {
        struct blkg_policy_data pd;
        //…
};
```

[그림 9. bfq_group_data, bfq_group 구조체]

bfq_group_data 는 시스템 파일을 통해 입력 받은 weight 를 저장하기위해 사용되었다. weight 정보는 cgroup 마다 존재하므로 blkcg_policy_data 를 사용하여 kernfs_open_file → cgroup_subsys_state → blkcg → blkcg_policy_data → bfq_group_data 의 경로로 구조체가 메모리상에 할당된 위치를 알아내어 cgroup 별 weight 정보를 bfq_group_data 의 weight 에 저장할 수 있다.

bfq_group 은 block device 의 request_queue 와 연관된 cgroup 의 상태를 추적할 수 있게 blkg_policy_data 라는 구조체를 사용하였다. 즉, 장치 및 cgroup 마다 bfq_group 이 생성될 수 있게 blkcg_policy 를 설정한 다음, blkg(blkcg_gq)을 사용하였다.

본 팀이 설계한 스케줄러에서는 cgroup 별 정보만 얻으면 되므로, kyber_fairness 에 blkcg_policy_data 타입의 변수를 할당하여 policy 등록 및 서브시스템 활성화를 할 때 cgroup 마다 kyber_fairness 가 생성될 수 있게 하였다.

2.2.2 blkcg_policy_kyber

새롭게 설계한 kyber_fairness 스케줄러에서 fairness 를 보장할 때 기준으로 두는 것은 cgroup 별로 설정한 weight 이다. 이를 kyber_fariness 의 weight 에 저장하며 weight 정보를 cgroup 파일 시스템으로부터 얻기 위해서는 blkcg_policy 구조체를 사용해야 한다.

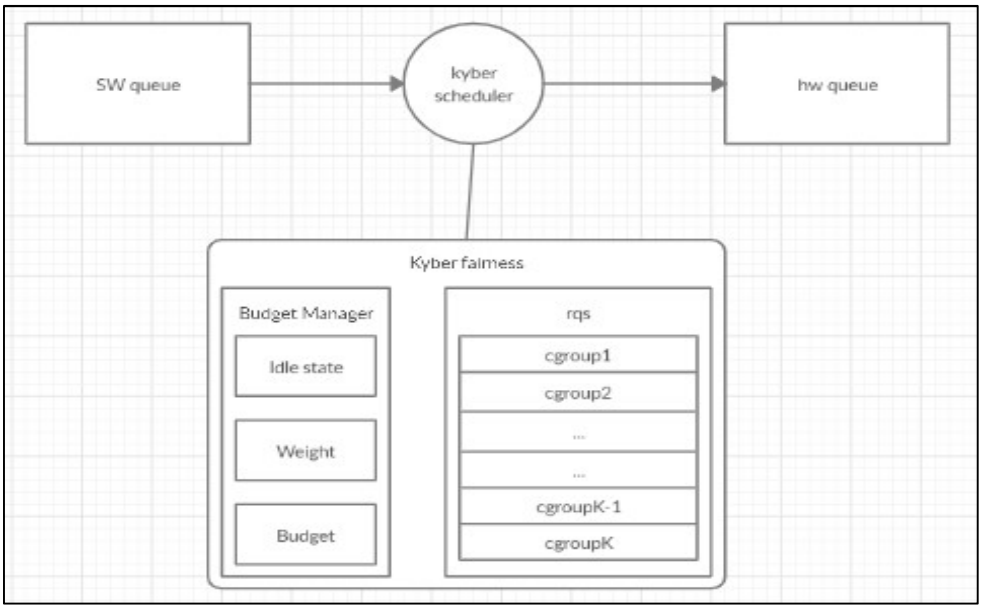
다음은 kyber_fairness 스케줄러에서 정의한 blkcg_policy 이다.

```
struct blkcg_policy blkcg_policy_kyber = {
        .dfl_cftypes                    = kyber_blkg_files,
        .legacy_cftypes                 = kyber_blkcg_legacy_files,
        .cpd_alloc_fn                   = kyber_cpd_alloc,
        .cpd_init_fn                    = kyber_cpd_init,
        .cpd_bind_fn                    = kyber_cpd_init,
        .cpd_free_fn                    = kyber_cpd_free,
};
```

[그림 10. blkcg_policy_kyber 구조체]

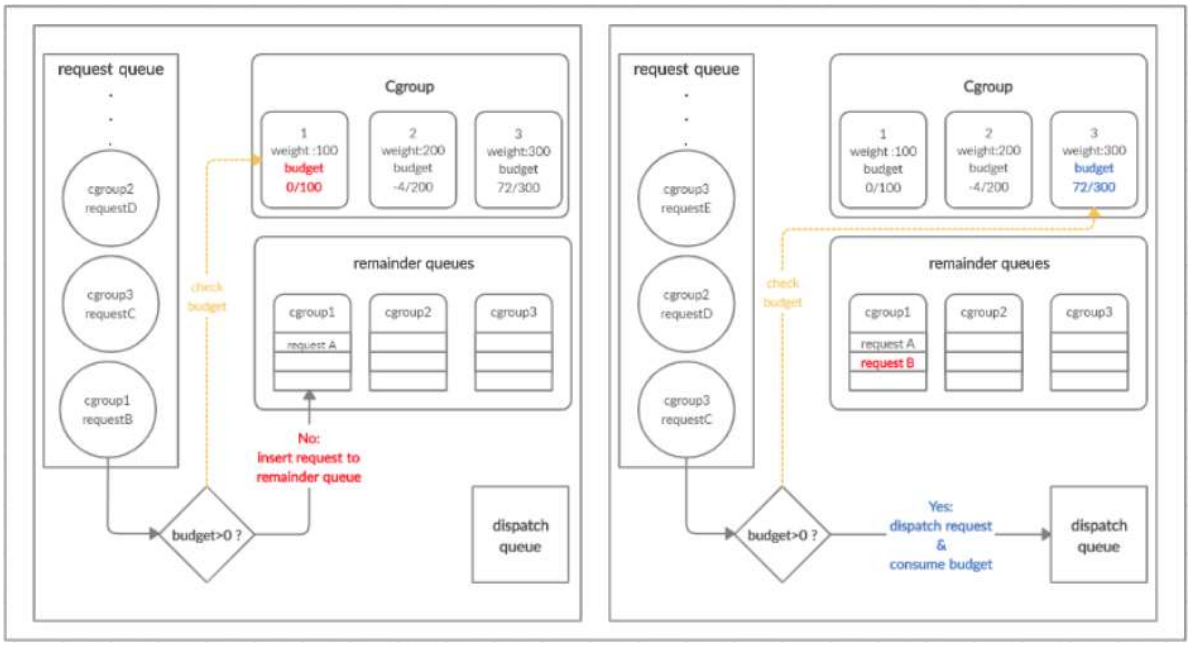
- kyber_blkg_files, kyber_blkcg_legacy_files: 시스템 파일을 통해 weight 를 입력 받을 수 있는 함수(write, write_u64)가 존재하는 cftype 구조체이다.
- kyber_cpd_alloc: Kyber_fairness 스케줄러에서 kyber_fairness 구조체를 cgroup 마다 생성(메모리에 할당)하기 위한 콜백함수이다.
- kyber_cpd_init: kyber_fairness 구조체를 초기화하기 위한 콜백함수이다.
- kyber_cpd_free: cgroup 설정이 변경됐을 때 사용되는 콜백함수이다. 여기서 kyber_fairness 구조체의 메모리 할당을 해제하여야 한다.

2.3 fairness 알고리즘



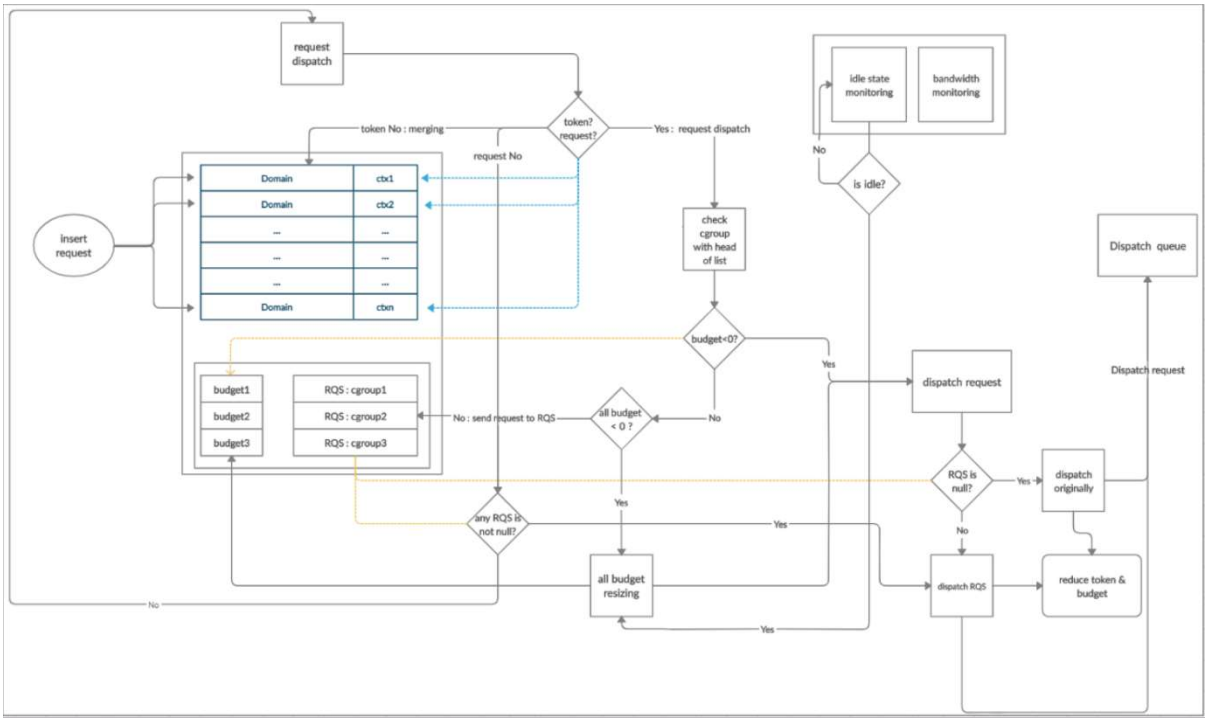
[그림 11. kyber_fairness 구조체의 위치]

각 cgroup 에 weight 를 기반으로 한 budget 을 할당하여서 fairness 를 보장하도록 하였다. kyber_fairness 구조체는 fairness 를 위해 throttling 을 걸 수 있는 remainder queue 인 rqs 멤버변수가 존재한다.

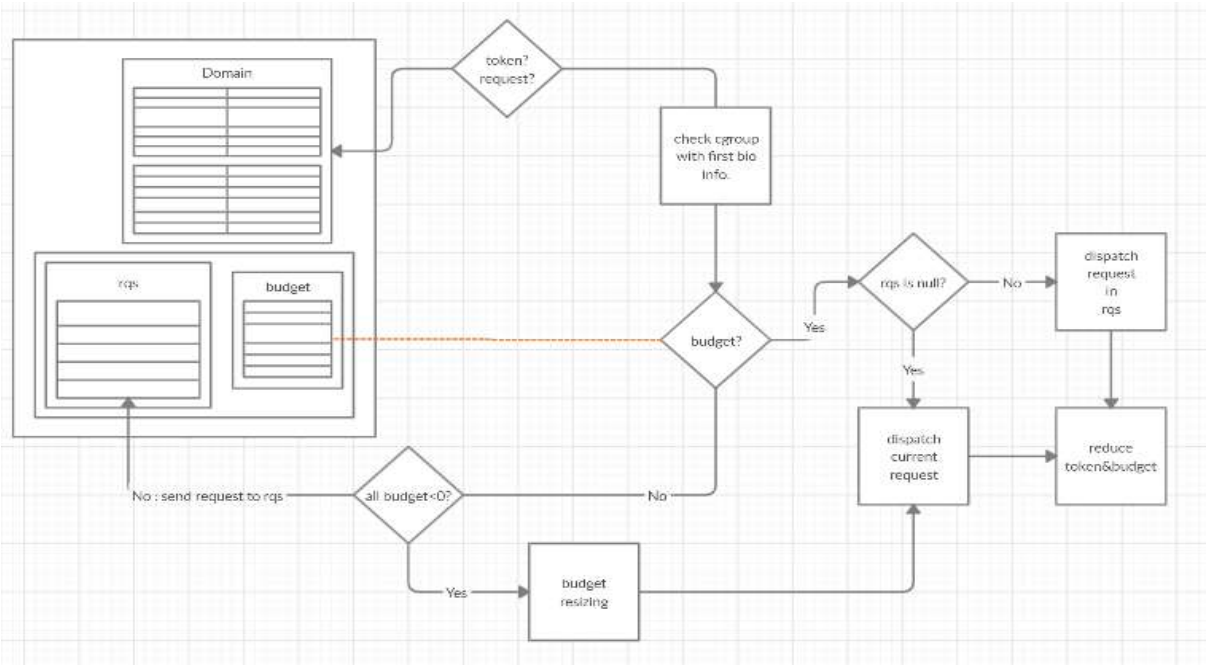


[그림 11. kyber_fairness 스케줄러의 fairness 보장 방법]

본 팀은 fairness 를 보장하기 위해 [그림 11]처럼 cgroup weight 에 맞게 budget 을 할당하였다. 어떤 request 를 처리하고자 할 때, 해당 request 가 속한 cgroup 의 budget 을 확인하고, [그림 11-1]에서처럼 budget 을 다 소모한 경우 throttling 을 걸어 해당 request 가 속한 cgroup 의 rqs 로 보낸다. 만약 [그림 11-2]처럼 budget 이 있는 cgroup 의 request 경우 hardware queue 로 dispatch 를 하여 fairness 를 보장하도록 설계하였다.



[그림 12. kyber_fairness 스케줄러 전체 알고리즘]

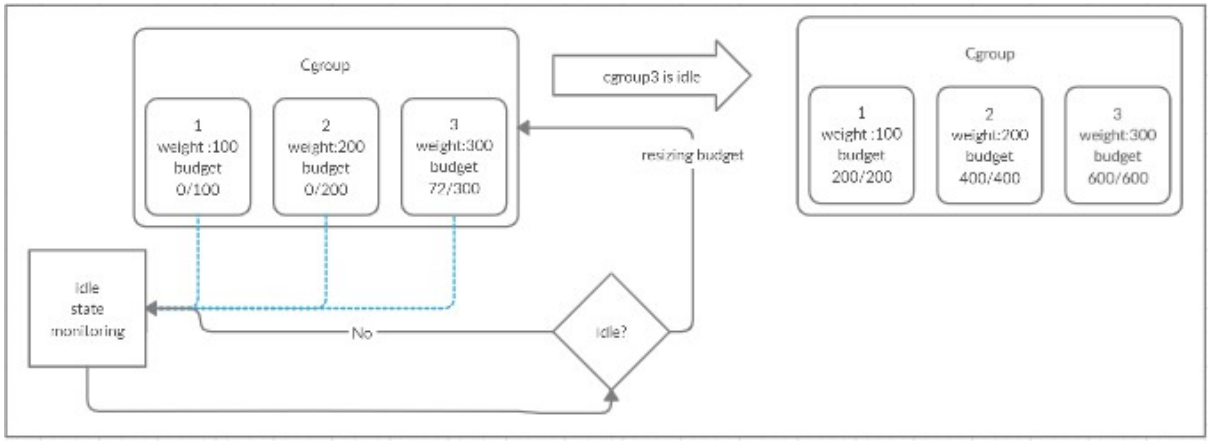


[그림 13. kyber_fairness 스케줄러 알고리즘 일부]

Dispatch 함수가 실행되면 먼저 software queue 에 request 가 있는지 확인하고 현재 domain 에 token 이 있는지 확인한다. 그리고나서 request 의 bio 변수를 통하여 해당 request 가 어느 cgroup 인지 확인한다. (software queue 에서 bio 들이 merge 되어 request 가 형성되는데 보통의 경우 software queue 에는 같은 cgroup 의 bio 들로 구성되어 있다.) 해당 cgroup 에

할당된 budget 을 모두 소모한 경우 request 는 hardware queue 로 dispatch 하지않고, cgroup 에 맞는 rqs 로 전송된다. 여기서 rqs 에 있는 request 는 domain 에 있는 request 들보다 더 오래된 것들이므로 dispatch 를 하는 경우 rqs 에 request 가 있는지 확인하고, rqs 에 속한 request 를 우선적으로 dispatch 해준다.

Budget 의 리필은 모든 cgroup 의 budget 이 다 소모되었을 때 (0 이하일때) 실행하도록 한다. 하지만, I/O 작업을 하지 않고 있는 cgroup 이 존재할 수 있으므로 만약 그러한 cgroup 이 있다면 해당 cgroup 의 idle 을 true 로 설정하여 리필 조건 확인에서 예외로 둔다. Idle 상태에 관한 정의는 다음 단락에서 계속하겠다.



[그림 14. Idle 상태일때 budget 의 리필]

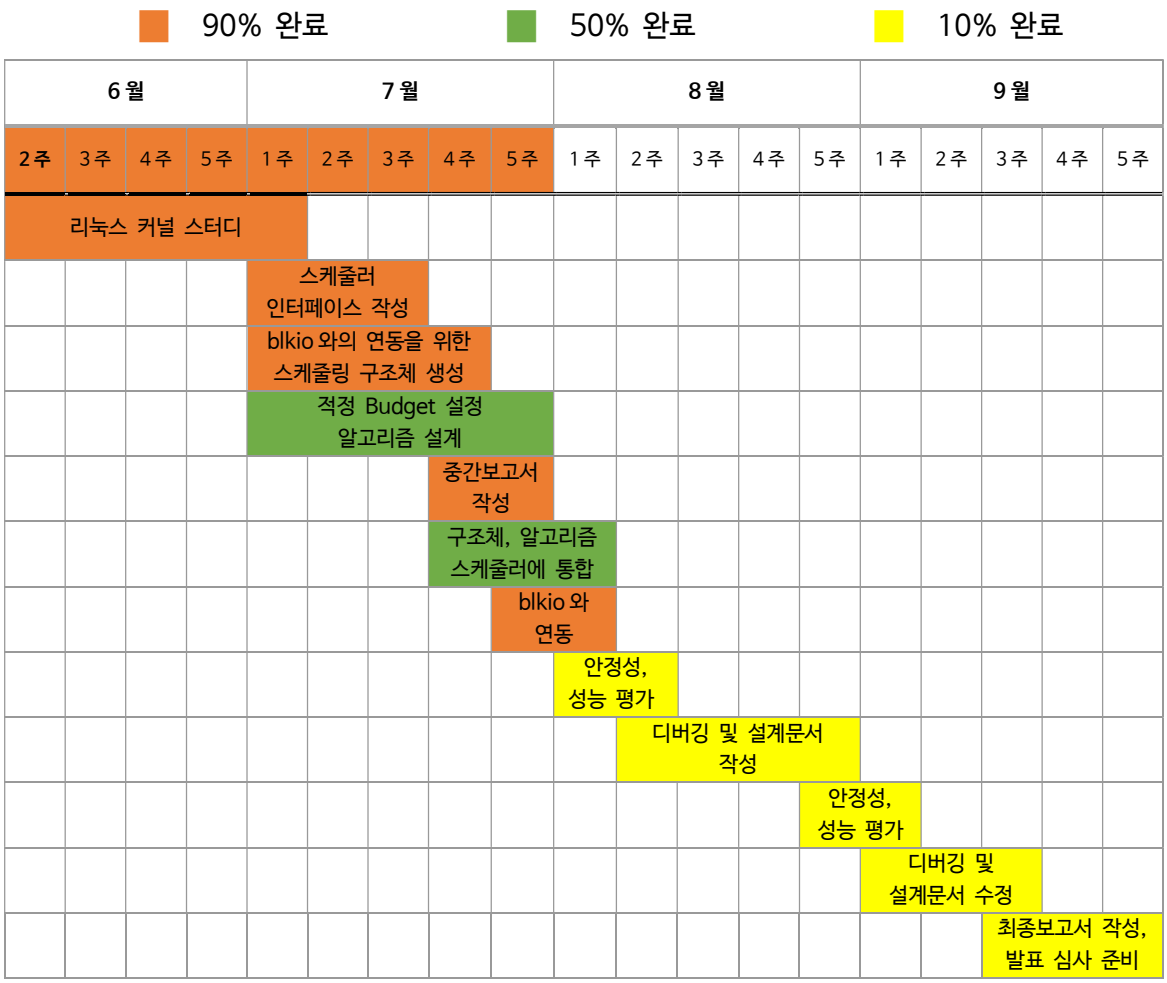
본 팀은 work-conserving 한 스케줄러를 만들기 위해 Idle 인 cgroup 을 구분할 필요가 있다고 생각하였다. cgroup 이 idle 하다의 의미는 해당 cgroup 의 I/O request 가 단위시간동안 존재하지 않는다는 의미로 정의하였다. 해당 cgroup 의 request 가 단위시간동안 존재하지 않는다는 것은 budget 의 변동이 없다는 것과 같으며, idle 인 cgroup 의 기준을 budget 의 변동이 없는 경우라고 할 수도 있다. 하지만, 단위시간을 얼마로 정할 것인가 하는 것은 어려운 일이기 때문에 이 부분에 대해서는 조금 더 고민이 필요할 것 같다.

[그림 14]는 cgroup3 이 idle 상태로 판단되었고, 1:2:3 으로 나누어져 있던 budget 을 1:2 로 나누어 가지고 cgroup3 이 idle 에서 빠져나올 수 있기 때문에 budget 도 할당해주었다. 전체 budget 이 늘어나 순간적으로 fairness 는 하락할 수 있지만 리필을 하는 작업을 덜 해도되기 때문에 전체적인 throughput 은 상승한다.

Budget 크기는 블록 디바이스의 bandwidth 에 맞게 설정하는 것이 가장 이상적이라고 생각한다. 따라서 현재 블록 디바이스의 실시간 bandwidth 를 구하는 것을 목표로 연구 중이다. 하지만 bandwidth 를 구하지 못할 경우를 대비하여, 기존 Kyber 스케줄러가 블록 디바이스의 latency 를 조절하기위해 사용하던 token 과 소모된 budget 정보를 이용하여 bandwidth 를 얻는 방법을 고려하고 있다.

3 개발 일정 및 역할분담

3.1 개발 일정



3.2 역할분담 및 구성원별 진척도

이름	역할 분담	진척도	
	Kyber 스케줄러에 fairness 와 관련된 구조체 생성 및 알고리즘 구현	Kyber 스케줄러 분석	O
		request throttling 함수 구현	O
		kyber_dispatch_request() 함수 수정	X
	최적의 Budget 을 계산할 수 있는 알고리즘 설계	State diagram 설계	O
		Idle 상태 정의	X
		디바이스의 bandwidth 획득	X
	cgroup 의 서브시스템인 blkio 와 연동	BFQ 스케줄러 분석	O
		blkcg_policy 구조체 완성	O
		kyber_fairness 구조체 생성 및 할당	O

[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.