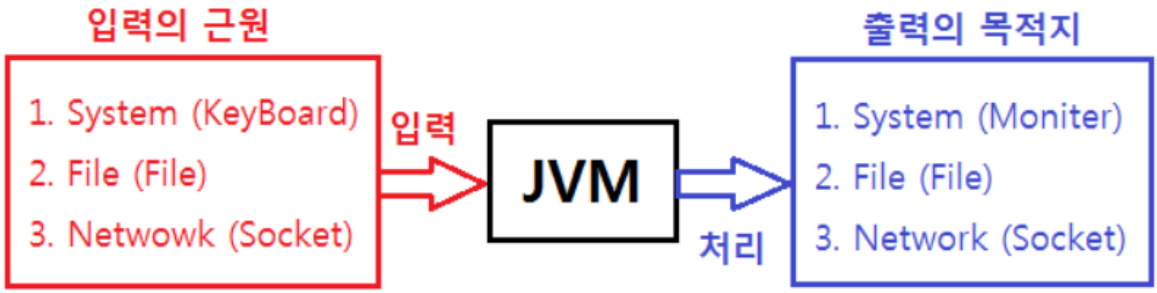
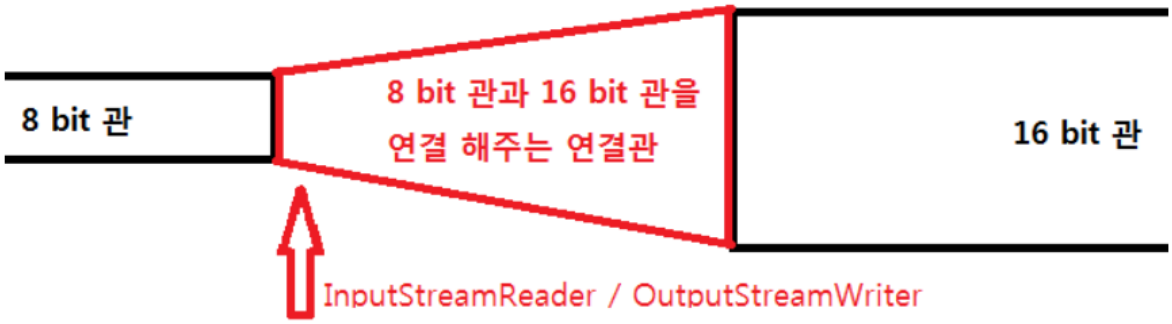
IO 스터디

IO

IN JAVA

8bit(1byte) Stream		16bit(2byte) Stream	
모든 데이터를 사용 할 수 있다. 속도가 빠르다.		문자열을 전용으로 다룬다. 속도가 느리다.	
입력	출력	입력	출력
InputStream	**OutputStream**	**Reader**	**Writer**
DataInputStream FileInputStream(모든파일) ObjectInputStream …	DataOutputStream FileOutputStream(모든파일) ObjectOutputStream …	FileReader(문자열파일) BufferReader …	FileWriter(문자열파일) BufferWriter …





- 한글 입력 받기 위해서 2byte필요!
- Q: 유저 IO는 어디에 달려있을까?(FILE IO는 디스크에, network IO는 socket에)

Object(객체)의 입출력

- 객체(instance/object)는 크기가 정해져있지 않기 때문에(대부분 8bit 이상) Stream을 이용해 JVM 외부로 내보 낼 수 없다.

(기본형데이터형은 JVM 외부로 내보낼 수 있다.)
- 객체를 8bit 크기로 잘라서 내보내는 방법을 사용해야 한다.
- 객체를 일정 크기로 잘라내는 행위를 **직렬화(marshalling)**라고 한다.
- **Serializable 인터페이스를 상속** 받은 클래스를 참조한 객체는 직렬화가 된다.
- 직렬화 된 **객체를 내보내는 스트림을 marshall 스트림**이라고 한다.
- 직렬화 작업으로 인해 분리된 객체를 원래대로 만드는 것을 **역직렬화(unmarshalling)**라고 한다.
- 직렬화된 객체에서 특정 자원을 JVM 밖으로 내보내는 것을 막으려면 **직렬화방지키워드 transient**를 사용하면 된다.

⦿ Object의 입력

- 객체를 JVM 밖으로 내보내기 위해서 직렬화(marshalling) 작업이 필요하다.

사용 방법

① 객체를 만들기 전에 참조할 데이터클래스를 미리 만들어두고
Serializable 인터페이스를 상속받아 직렬화 시킨다.

```
public class MyData implements Serializable { ... }
```

② 클래스를 참조하여 객체를 생성한다.

```
MyData md = new MyData();
```

③ 스트림을 통해 해당경로에 파일을 생성한다.

```
File file = new File("파일의경로");
```

④ 파일에 8bit 스트림을 연결한다.

```
FileOutputStream fos = new FileOutputStream(file);
```

⑤ 객체를 사용하기 위해 스트림을 확장시킨다.

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

⑥ 스트림에 객체를 기록한다.

```
oos.writeObject(md);
```

⑦ 스트림에 있는 객체를 파일로 분출시킨다.

```
oos.flush();
```

⦿ Object의 출력

- 객체를 JVM 안으로 가져오기 위해서 역직렬화(unmarshalling) 작업이 필요하다.

사용 방법

① 해당경로에서 파일을 불러온다.

```
File file = new File("경로");
```

② 파일에 8bit 스트림을 연결한다.

```
FileInputStream fis = new FileInputStream(file);
```

③ 객체를 사용하기 위해 스트림을 확장시킨다.

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

④ 불러온 파일에서 객체 읽어들이기

```
Mydata md = (MyData)ois.readObject();
            ㄴ 강제형변환
```

Disk IO

page cache

- **개념**

  > 리눅스에서 파일 I/O가 일어날 때 커널은 PageCache를 이용해서 디스크에 있는 파일의 내용을 메모리에 잠시 저장하고, 필요할 때마다 메모리에 접근해서 사용한다고 배웠다. 이를 통해서 디스크보다 더 빠른 메모리의 접근 속도를 활용할 수 있고, 전체적으로 시스템의 성능을 향상시킬 수 있다.

- **과정**

```
1. (user mode) request file read
2. (kernel mode) check page cache
3. (disk) ask disk to read
4. (kernel mode) return file content
5. (kernel mode) save in page cache
6. (user mode) return file content
```

이 과정에서 dirty page가 생성이 되는 경우가 있다. 만약 파일이 update, write가 되는 경우 커널은 해당 메모리 영역에 파일이 변경되었음을 알려주는 dirty bit를 켠다. 이렇게 dirty bit가 생성되면 내용이 바뀌었다는 뜻이므로 케시에 있는 파일과 디스크에 있는 파일이 sync가 맞지 않는 상태이다. 그렇기에 dirty bit가 있는 상태의 파일은 flush를 통해서 이 문제를 해결해야 한다.

dirty page를 flush하기 위한 기준(파라미터)

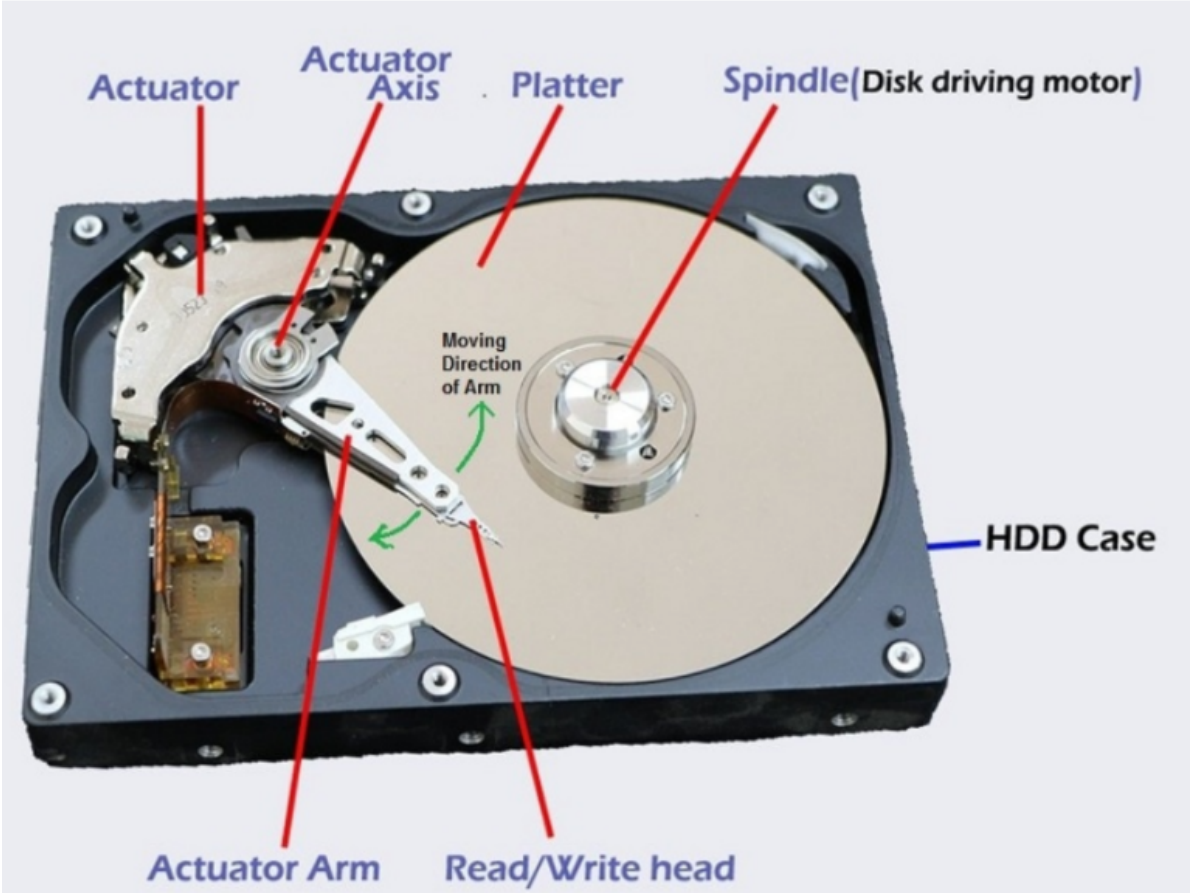
- dirty_background_ratio
  - 해당 비율만큼 dirty page가 쌓였으면
- dirty_background_bytes
  - 해당 바이트 만큼 dirty page가 쌓였으면
- dirty_ratio(강제성) - 해당 프로세스의 IO작업을 모두 중지하고 write
  - 해당 비율만큼 dirty page가 쌓였으면
- dirty_bytes(강제성)
  - 해당 바이트 만큼 dirty page가 쌓였으면
- dirty_writeback_centisecs
  - 몇초 간격으로 flush 커널 스레드를 깨울지
  - dirty_expire_centisecs
    - 디스크에 flush할 기준 페이지를 정한다. 예를들어 n초 동안 flush되지 않은 페이지를 동기화시킨다.

동기화

- **백그라운드 동기화**
  - 백그라운드에서 동기화
- **주기적인 동기화**
  - 특정 주기로 깨워서 동기화
- **명시적인 동기화**
  - 명령어를 통함

IO BLOCK 스케줄러

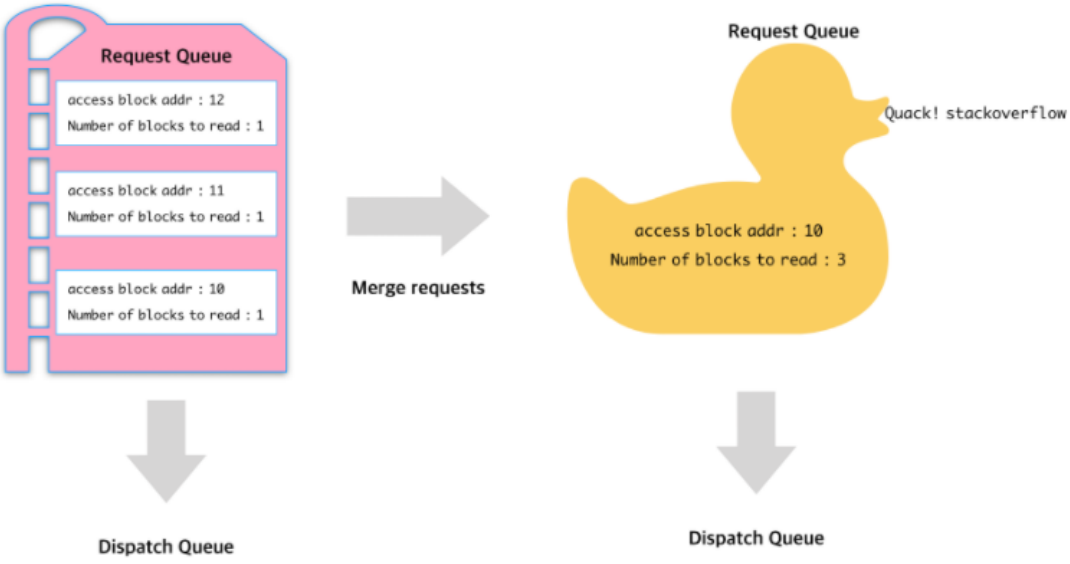
HDD 구조



Platter가 움직이면서 디스크의 내용을 읽는다. 즉 이 움직임이 최소한으로 줄어드는 것이 이상적이다.

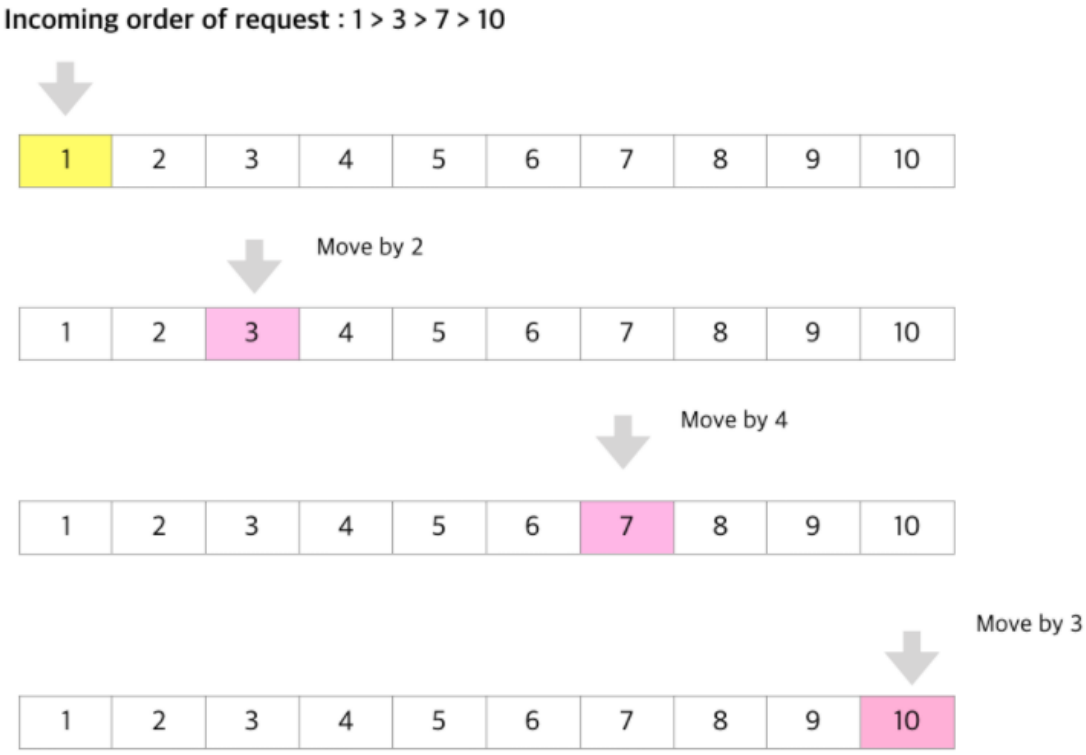
IO 요청의 병합

- IO작업이 쌓이는 일이 발생할 경우(즉 IO작업이 무진장 들어오는 경우) 여러개의 요청을 하나로 합치는 작업이 필요하다.



Incoming order of request : 1 > 7 > 3 > 10

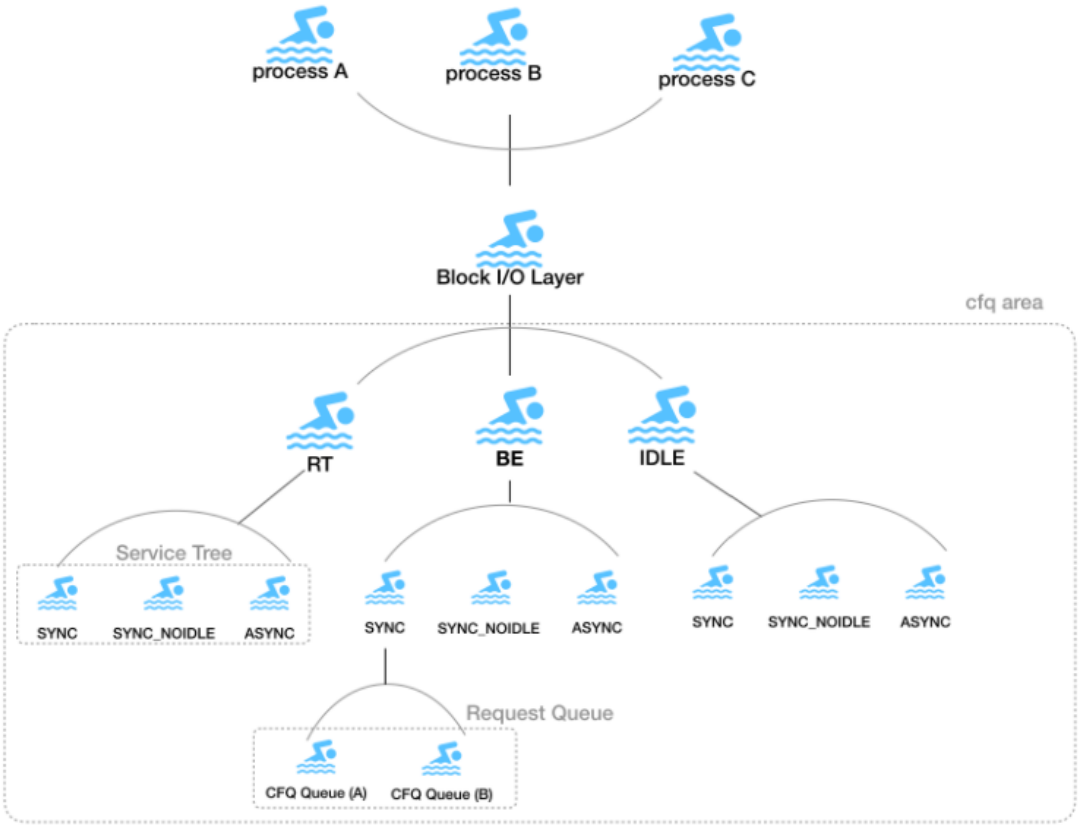
총 17번 움직인다.



총 9번 움직인다.

- 하지만 이러한 작업은 HDD에만 해당된다. SSD는 요청이 더 늦게 처리되는 문제가 발생할 수 있음
  - SSD는 컨트롤러를 통해 전기적 신호를 저장하기 때문

cfg IO 스케줄러 (Completely Fair Queueing)



- IO의 특성에 따라 RT(Real Time), BE(Best Effort), IDLE 중 하나로 IO 요청을 정의한다.

  ionice를 통해 변경이 가능하다. 대부분의 요청은 BE에 속한다. BE -> RT -> IDLE 순으로 처리한다.

셋을 분류한 뒤에 service tree라는 워크로드별 그룹으로 다시 나눈다.

그후 해당 노드에 A프로세스에서 발생시킨 요청은 cfq queue(A)에, B프로세스에서 발생시킨 요청은 cfq queue(B)에 저장한다.

- **SYNC(주로 read)**

  순차적인 동기화 IO 작업, 주로 순차적 read -> 순차적으로 읽기 때문에 디스크 헤드와 가까운 위치의 작업이 들어올 확률이 높기 때문

  그래서 대기시간이 있떠라도 가까운 곳의 IO요청을 처리하게 되면 더 좋은 성능을 내는데 도움이 되기 때문에 임시 대기하게 된다. (slice_idle의 값으로 대기 시간 설정 가능)

- **SYNC_NOIDLE**

  여기에 속한 queue는 임의 작업은 디스크 헤드를 많이 움직이게 하기 때문에 굳이 다음 요청을 기다려서 얻게 되는 성능상의 이점이 없다. 그렇기 때문에 큐에 대한 처리를 완료한 후 대기시간 없이 바로 다음 큐에 대한 IO작업을 실행

- **ASYNC(주로 write)**

  비동기화 IO작업은 ASYNC트리에 모아 두고 한번에 처리

- **스케줄러의 파라미터**

  ```
  back_seek_max   fifo_expire_async   group_idle   low_latency
  slice_async  slice_idle
  back_seek_penalty  fifo_expire_sync  group_isolation  quantum
  slice_async_rq  slice_sync
  ```

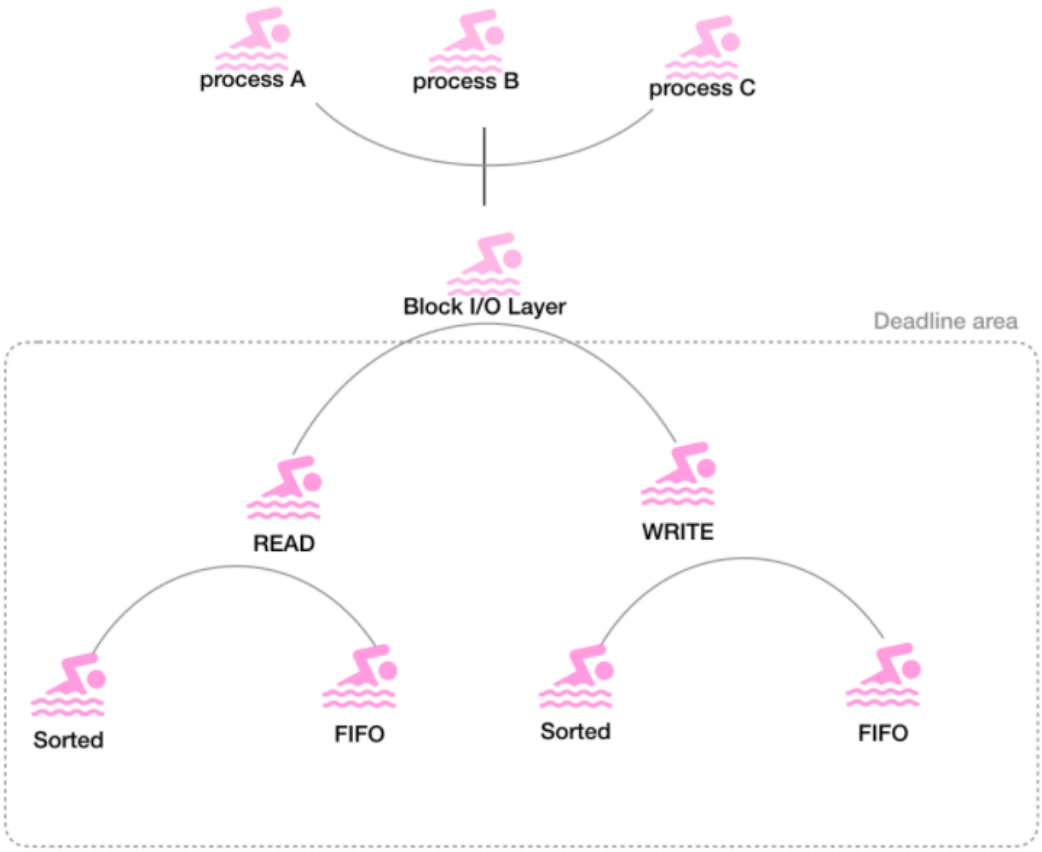
  - back_seek_max
    - queue의 작업을 끝내고 기다리는 시간 동안 헤더에서 이동할 거리가 바로 다음 작업으로 생각되는 max값
  - back_seek_penalty
    - IO작업이 sorted된 상태로 진행될 때 그냥 밀리면 억울하니까 뒤로가는 대신 실행할 값에 이득을 주자 (비교하는 순간에)
  - fifo_expire_async
    - async요청의 만료 시간. 대부분 flush는 OS가 관리하기 때문에 이 flush를 날리고 성공여부를 기다리는 시간이다.
    - sync 요청의 만료 시간.
  - group
    - cgroup에서 같은 그룹으로 묶은 것을 큐를 이동할때 기다리는 시간없이 바로 이동.
  - group_isolation (SYNC_NOIDLE 상황)
    - 값이 0이면 cgroup의 루트로 접근되기에 헤더를 더 많이 움직여야되기에 IO요청 완료까지 시간이 오래걸림. 값이 1이면 cgroup 값에 영향을 받아 cgroup 분류가 명확해짐 -> cgroup을 활용하는 환경에서 큰 영향을 주는 값
  - low_latency
    - I/O 요청을 처리하다가 발생할 수 있는 대기 시간을 줄이는 역할을 한다. 이 값은 boolean으로 0이나 1, 즉 disable/enable을 의미한다. cfq는 현재 프로세스별로 별도의 큐를 할당하고 이 큐들은 그 성격에 따라 RT(Real Time), BE(Best Effect), IDLE 이렇게 세개의 큐로 다시 그룹화된다. 만약 I/O를 일으키는 5개의 프로세스가 있다고 가정했을 때 아무 설정을 하지 않으면 이 5개의 프로세스는 각각 sorted, fifo 2개씩 큐를 할당 받고 이 큐들은 BE 그룹으로 묶인다. 이때 low_latency가 설정되어 있다면 각 큐에 time_slice를 할당하기 전에 각 그룹별로 요청이 몇개씩 있는지 확인한다. 각 그룹별로 있는 큐의 요청들을 전부 합친 후에 time_slice를 곱하게 되는데 이때 expect_latency값이 생성된다. 즉 해당 그룹의 큐를 모두 처리하는데 걸릴 시

간을 계산하고 이 계산 결과가 target_latency 값보다 크다면 이 값을 넘지 않도록 조절한다.

과거에는 target_latency도 설정이 가능한 매개변수였는데 최근에는 300ms로 소스 코드상에 하드 코딩된다.

```
    만약 low_latency가 켜져 있으면 그룹의 큐를 확인하지 않게 되고, 큐에 I/O요청이 많을 수록 한번의 time slice로는 처리가 어렵다. 이렇게 소요시간이 많이 걸리는 요청들은 자신의 time slice를 다 쓰고 다시 차례가 돌아오기를 기다려야하기 때문에 처리하는데 많은 시간이 소요된다. low_laency 설정 값을 통해서 전체적인 I/O 요청을 바탕으로 time slice를 조절한다. 아마 성능에 가장 큰 영향을 끼치는 요소가 될 것이다.

* slice_idle

  * 보통의 IO요청은 random access, sequential access가 많다. 그렇기에 queue가 끝나고 잠시 기다리는 시간.

* slice_sync

  * sync에 대한 time slice의 기준. queue에 작업이 하나라도 있으면 이 값만큼 실행한다.

* slice_async

  * async에 대한 time slice의 기준

* slice_async_rq

  * 큐의 값을 한번에 꺼내어 dispatch queue에 넘기는 최대 요청수(async)

* quantum

  * sync 요청을 꺼내서 dispatch에 넘기는 최대 요청수.

    이값이 커진다면 큐에서 한번에 꺼낼 수 있는 요청의 개수가 증가하겠지만 그만큼의 하나의 큐가 실행될 때 걸리는 시간이 늘어나기 때문에 경우에 따라서는 성능이 저하될 수 있다.
```

deadline IO 스케줄러



I/O요청 별로 완료되어야하는 deadline을 가지고 있는 I/O 스케줄러이다. 가능한 한 해당 deadline을 넘기지 않도록 동작한다.

읽기 요청에 대한 처리를 우선

sorted list: 각각 읽기 요청과 쓰기 요청을 저장하고 있으며 섹터 기준으로 정렬된다.

fifo list: 시간을 기준

평상시에는 sorted list에서 정렬된 상태의 요청을 꺼내서 처리

ifo list에 있는 요청들 중 deadline을 넘긴 요청이 있다면 fifo list에 있는 요청을 꺼내서 처리한다.

요청이 처리되고 나면 처리된 요청을 기준으로 sorted list를 재정렬해서 그 이후의 I/O 요청을 처리한다.

매개변수

```
fifo_bacth    frount_merges    read_expire    write_expire    writes_starved
```

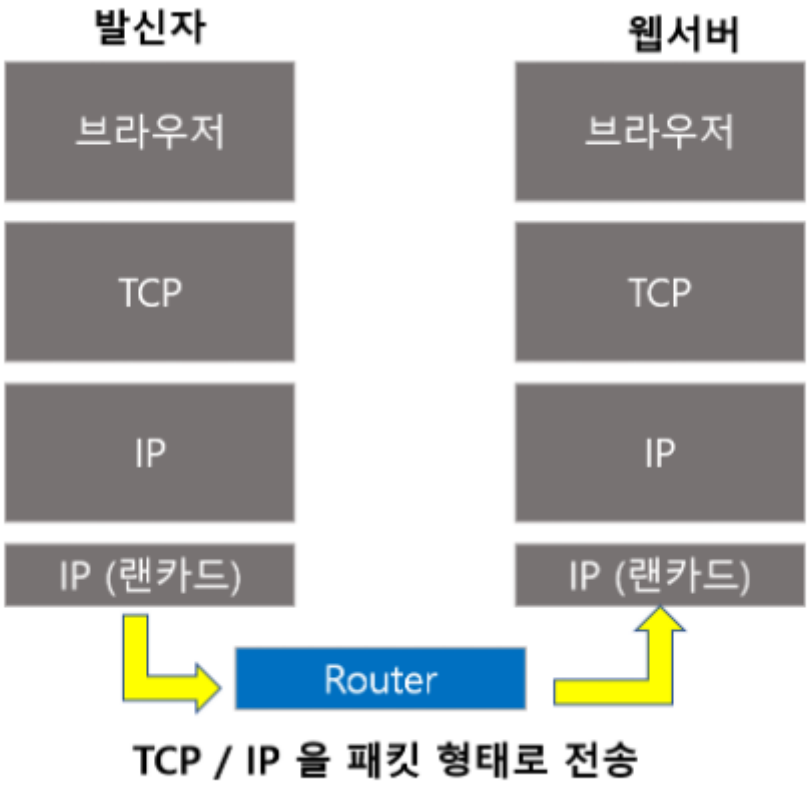
noop IO 스케줄러

정렬은 하지 않고 병합만 한다. SSD기준으로 섹터에 대한 의미가 존재하지 않기 때문 다만 헤더에 얼마나 자주 접근해야 하는지가 중요하기에 병합은 진행함.

성능

- cfq, deadline 각각이 유리한 환경
    - cfq - 여러 프로세스가 동시에 실행 및 빠른 응답을 원함
    - deadline - 한 프로세스가 다량의 IO작업을 먹고 있는 경우
    - DB서버, 웹서버

Network IO



packet: 보낼 데이터를 잘라서 보내는 단위.

IP: internet protocol -> 통신 규약

Network

소켓 통신

클라이언트

1. create: 소켓을 생성
    - 타입 지정 가능 (TCP, UDP, ...)
2. connect: 서버 측에 연결 요청
    - 연결 요청에 대한 결과(성공, 거절, 시간 초과 등)가 결정되기 전에는 **connect()**의 실행이 끝나지 않는다
    - Block 방식
3. send/recv: 데이터를 송수신
    - Block 방식
    - send는 보내기만 / recv는 받기만
4. close: 소켓을 닫기

서버

1. create: 소켓을 생성
2. bind: 소켓에 서버가 사용할 ip, port를 결합
    - 소켓들이 같은 port를 쓰면 받은 데이터를 어디로 보낼지 모르게 되기에 같은 포트를 쓰려하면 에러 반환
3. listen: 클라이언트로 connect요청이 오는지 주시
    - return은 SUCCESS, FAIL만
    - 클라이언트 연결 요청에 대한 정보는 시스템 내부적으로 관리되는 큐(Queue)에서 쌓이게 되는데, 이 시점에서 클라이언트와의 연결은 아직 완전히 연결되지 않은(NOT ESTABLISHED state) 대기 상태가 된다.
4. accept: 데이터 통신을 위한 소켓을 생성(connect가 수신되면)
    - 내부의 socket을 만든다.
5. send/recv: 데이터를 송수신
6. close: **서버 소켓** 닫기

Q실험: MB단위의 큰데이터 넘기는 클라 소켓으로 서버에 전달. Byte단위의 데이터를 넘기는 클라 소켓으로 서버에 전달.

종류

Protocol (프로토콜)	
TCP/IP	TCP(Transmission Control Protocol) / IP(Internet Protocol)
	- ServerSocket 클래스 사용 - 전화와 같다. - 속도가 느리다. - 신뢰성 있는 통신을 할 수 있다. (확인작업을 거치기 때문에) - 오류검출bit를 사용하여 오류를 검출하고 데이터를 제대로 받을때까지 재전송한다. - 패킷이 고정되어있다. (8bit) - 주로 돈과 관련된 작업에서 많이 사용된다.
UDP/IP	UDP(User Datagram Protocol) / IP(Internet Protocol)
	- DatagramSocket 클래스 사용 - 우편과 같다. - 속도가 빠르다. - 비신뢰성 통신이다. - 오류검출을 하지 않는다. - 패킷을 개발자가 마음대로 정의할 수 있다. - 속도에 민감한 경우(화상채팅, 게임 등등)에 사용된다.

- 통신을 통해 컴퓨터와 컴퓨터를 연결하는 것

- Protocol을 사용해서 규칙에 맞게 통신하고 데이터를 전송한다.
- Server와 Client가 존재한다.
- Socket 통신이다.
- Socket은 데이터를 주고받기 위해서 Server와 Client를 연결하는 역할을 한다.
- Server 소켓과 client 소켓이 있다.
- OSI 7계층에서 상위 3계층 수준의 코딩이다.

TCP의 응답 방식

TCP FLAG(신호) 6가지

이러한 세션연결과 해제 이외에도 데이터를 전송하거나 거부, 세션 종료 같은 기능이 패킷의 FLAG 값에 따라 달라지게 되는데, TCP FLAG는 기본적으로 6 가지로 구성.

- **1. SYN (Synchronization: 동기화) - 연결 요청 플래그**

  TCP에서 세션을 성립할때 가장 먼저 보내는 패킷

  시퀀스 번호를 임의적으로 설정하여 세션을 연결하는 데에 사용 (초기에 시퀀스 번호를 보내게 됨)

- **2. ACK(Acknowledgement: 답신) - 응답**

  상대방으로 부터 패킷을 받았단걸 알려주는 패킷

  다른 플래그와 같이 출력되는 경우도 있으며 받는 사람이 보낸사람 시퀀스 번호에 TCP 계층에서 길이 또는 데이터 양을 더한 것과 같은 ACK를 보냄(일반적으로 + 1)

  ACK 응답을 통해서 보낸 패킷에 대한 성공, 실패를 판단하여 재전송하거나 다음 패킷을 전송하게 됨.

- **3.RST(Reset) - 재 연결 종료**

  재설정을 하는 과정이며 양방향에서 동시에 일어나는 중단 작업

  비정상적인 세션 연결 끊기에 해당한다.

  이패킷을 보내고 잇는 곳이 현재의 연결을 즉시 끊고자 할때 사용

- **4. PSH(Push) - 밀어넣기**

  텔넷(TELNET)과 같은 상호 작용이 중요한 프로토콜의 경우 빠른 응답이 중요하게 되는데 이때 받은 데이터를 즉시 목적인 OSI 7 Layer의 Application 계층으로 전송하다록 하는 flag.

  대화형 트래픽에 사용되는 것으로 버퍼가 채워지기를 기다리지 않고 데이터를 전달

  텔넷(telnet)은 원격지의 컴퓨터를 인터넷을 통해 접속하여 자신의 컴퓨터처럼 사용할 수 있는 원격 접속 서비스

- **5.URG(Urgent) - 긴급 데이터**

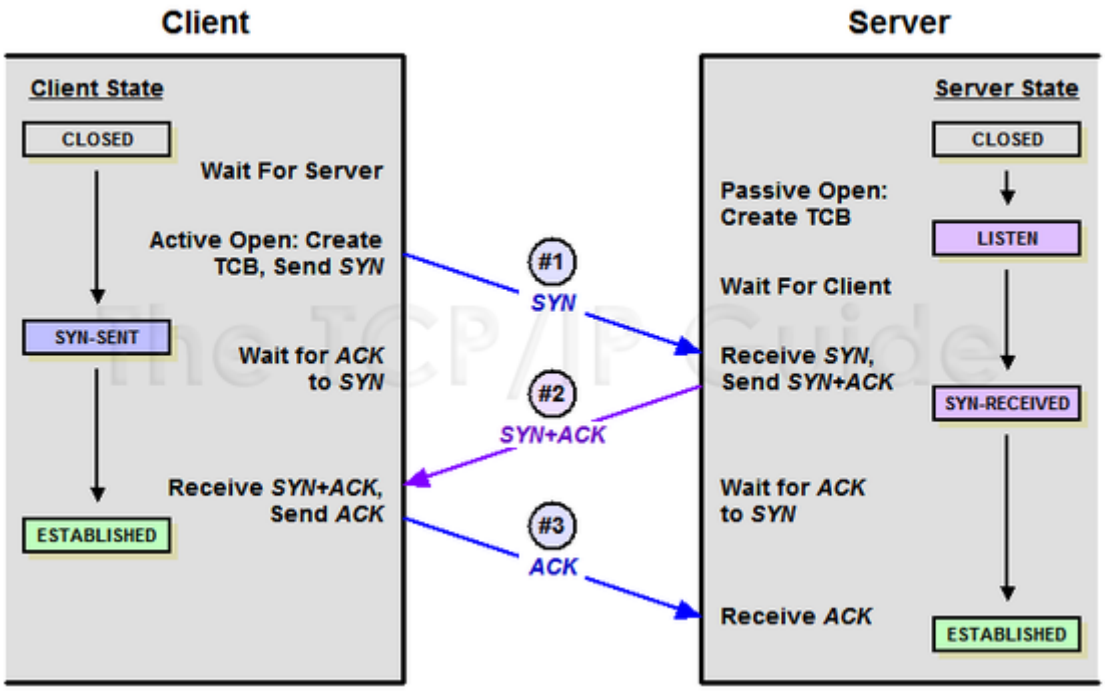
  Urgent pointer(전송하는 데이터 중에서 긴급히 전달해야하는 내용이 있는경우 사용)가 유효 한것인지 나타냄

  다른 데이터보다 우선순위가 높아야함.

- **6.FIN(Finish) - 연결 종료 요청**

  세션 연결을 종료시킬 때 사용되며 더이상 전송할 데이터가 없을을나타냄.

3 way handshaking



#1

A클라이언트는 B서버에 접속을 요청하는 SYN 패킷을 보낸다. 이때 A클라이언트는 SYN 을 보내고 SYN/ACK 응답을 기다리는SYN_SENT 상태가 되는 것이다.

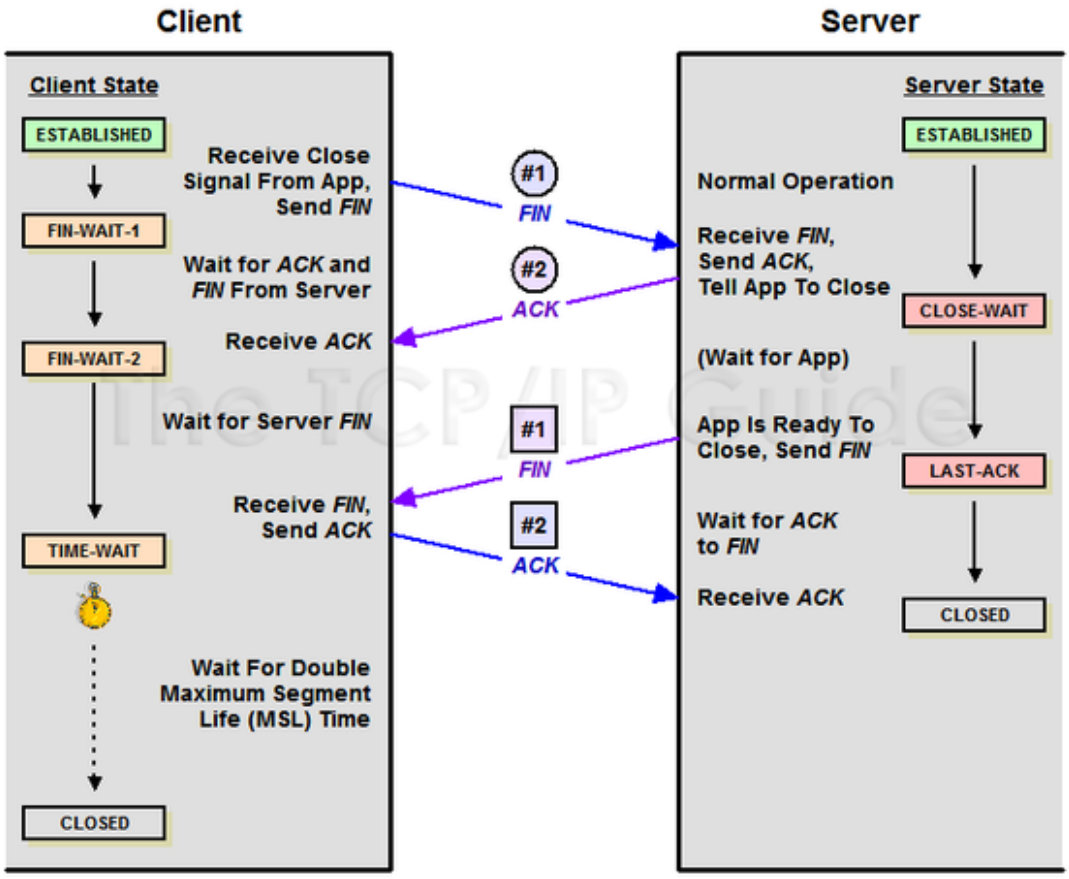
#2

B서버는 SYN요청을 받고 A클라이언트에게 요청을 수락한다는 ACK 와 SYN flag 가 설정된 패킷을 발송하고 A가 다시 ACK으로 응답하기를 기다린다. 이때 B서버는 SYN_RECEIVED 상태가 된다.

#3

A클라이언트는 B서버에게 ACK을 보내고 이후로부터는 연결이 이루어지고 데이터가 오가게 되는것이다. 이때의 B서버 상태가 ESTABLISHED 이다.

4 way handshaking(Q: 3way로 왜 안돼~)



#1

클라이언트가 연결을 종료하겠다는 FIN플래그를 전송한다.

#2

서버는 일단 확인메시지를 보내고 자신의 통신이 끝날때까지 기다리는데 이 상태가 **TIME_WAIT**상태다.

#3

서버가 통신이 끝났으면 연결이 종료되었다고 클라이언트에게 FIN플래그를 전송한다.

#4

클라이언트는 확인했다는 메시지를 보낸다.

status TIME-WAIT

FIN을 받고 routing delay 패킷 유실로 인한 재전송이 FIN보다 늦을 경우를 대비하여 TIME-WAIT 상태가 필요하다. DEFAULT 240초 동안 세션을 남겨놓고 CLOSED상태가 된다.

JAVA 기준 NETWORK

Server (서버)		Client (클라이언트)

① 서버소켓을 생성하고 대기 중
ServerSocket server = new ServerSocket(65000);
 	
③ 클라이언트의 접속 허가
Socket client = server.**accept();**
 	
④ 데이터를 내보내기 위한 스트림 연결
Data**Output**Stream dos =
new DataOutputStream(client.**getOutputStream**());
 	
④-1 스트림에 메시지 보내기
dos.writeUTF("내보낼메세지");
 	
④-2 스트림의 메세지 분출
dos.flush();
		② 소켓을 생성하고 서버로 접근
		Socket clinet = new Socket("211.63.89.170",65000);
		
		⑤ 소켓에 스트림을 연결
		Data**Input**Stream dis =
		new DataInputStream (clinet.**getInputStream**());
		
		⑤ 메세지 받아오기
		dis.readUTF();

참조

https://m.blog.naver.com/uok02018496/221787888256

https://mindnet.tistory.com/entry/%EB%84%A4%ED%8A%B8%EC%9B%8C%ED%81%AC-%EC%89%BD%EA%B2%8C-%EC%9D%B4%ED%95%B4%ED%95%98%EA%B8%B0-22%ED%8E%B8-TCP-3-Way Handshake-4-WayHandshake

http://blog.naver.com/PostView.nhn?blogId=lunatic918&logNo=157893292&categoryNo=1&parentCategoryNo=0&viewDate=¤tPage=1&postListTopCurrentPage=1&from=postView

https://recipes4dev.tistory.com/153