[Chapter 11-15]

경영학과/2018100233/김정민

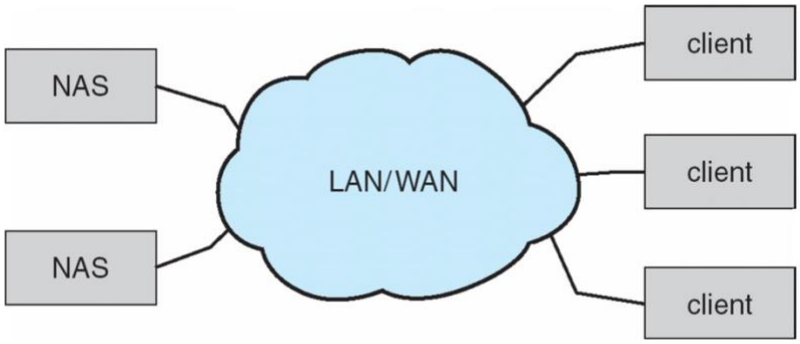
I. Mass-Storage

A. Mass Storage 종류

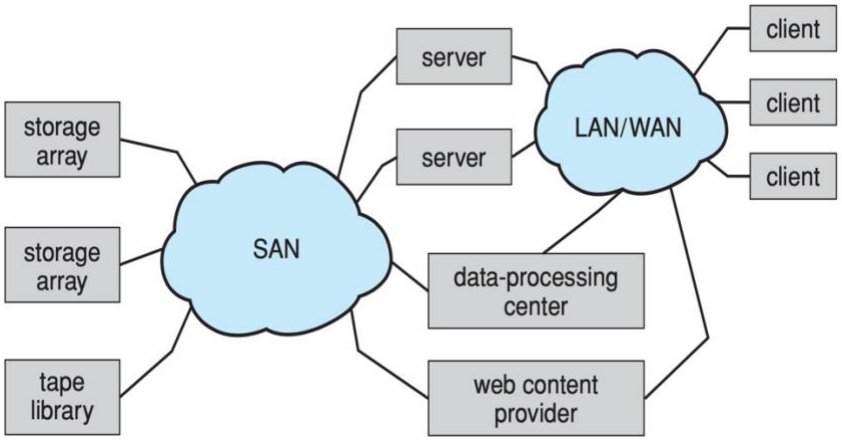
1. HDD (Hard Disk Drive)
2. SSD (Solid State Disk)
3. RAM disk
   - 데이터 I/O가 많이 일어난다. -> 메모리의 일부분을 디스크처럼 사용 -> 마지막엔 디스크에 백업
4. Magnetic tape

B. Disk 연결방식(Attachment)

1. Host attached via an I/O port
   - 실제 컴퓨터의 I/O 포트에 연결
2. Network attached via a network connection
   - 네트워크를 통한 디스크 연결
   - Network-Attached Storage (NAS)



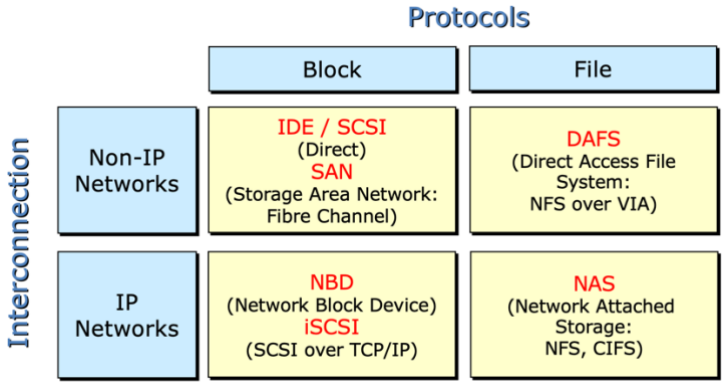
   - ■ IP 네트워크를 통해 접근할 수 있는 저장 장치
   - ■ NFS, CIFS 등 PC와 논리적으로 연결하는 프로토콜이 있다.
   - ■ 파일 단위로 처리해서 저장한다. (특이한 케이스)
   - ■ 해당 파일을 클라이언트로 전송할 때는 패킷으로 쪼개서 전송한다.
   - ■ 내장된 컨트롤러가 존재한다.
   - Storage-Area Network (SAN)



   - ■ 여러 개의 storage array를 SCSI나 Fibre(광케이블) 네트워크로 연결한다.
   - ■ 저장 장치를 위해 연결된 사적인 네트워크를 통해 데이터를 주고받는다.
   - ■ 블락 단위로 처리해서 저장한다. (저장장치 역시 논리적, 물리적 주소가 분리되어 있다.)
   - ■ SAN을 위한 파일 시스템은 GFS 같은 것들이 있다.

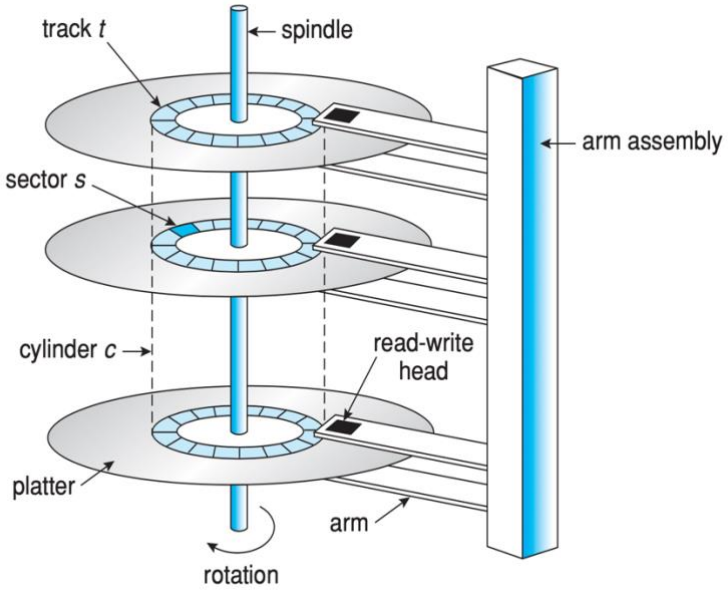
- 각각의 storage array는 랙에 HDD를 여러 개 꽂는 형식이다.

3. Storage Architecture
   - 프로토콜 : 블락 단위 vs 파일 단위
   - Interconnection : IP 네트워크 여부



C. HDD

1. Host-attached & Block-level
2. 디스크는 복잡한 물리적 저장 장치이다.
   - 에러, 잘못된 블락 등
3. OS가 할 일은 이런 복잡함을 상위 레벨의 애플리케이션에서는 알 수 없도록 숨기는 것이다. (추상화)
   - Low-level device drivers (디스크 read를 시작한다.)
   - Higher-level abstractions (파일, 데이터베이스 등)
   - OS는 각각의 클라이언트한테 다른 수준의 디스크 접근을 제공한다.
     - Physical disk block (surface, cylinder, sector)
     - Disk logical block (disk block #)
     - Logical file (filename, block or record or byte #)



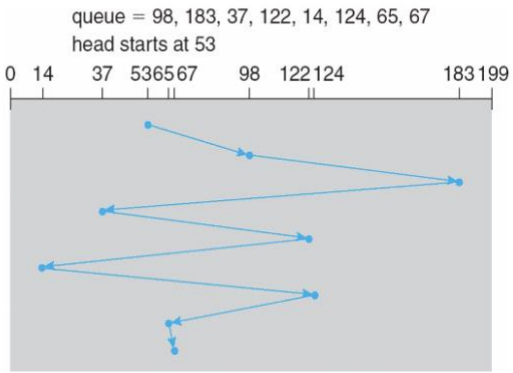
4. 디스크와 상호작용
   - 디스크에 요청하는 것은 상당히 많은 정보를 필요로 한다.
     - Cylinder #, surface #, track #, sector #, transfer size
     - CPU에서 명령을 내려준다.
   - 과거 디스크는 해당 정보를 얻기 위해 OS가 필요했다.
     - OS가 모든 디스크 파라미터를 알아야 했다. (모든 논리적인 동작)
     - 최근에는 하드웨어에 있는 디스크 컨트롤러가 대신 한다.

- 최근 디스크는 더욱 복잡해졌다.
  - 모든 섹터가 같은 크기를 가지진 않고, 섹터들은 다시 매핑된다.
- 최근 디스크는 더욱 높은 수준의 인터페이스를 제공한다. (ex. SCSI)
  - 디스크는 데이터를 블락들의 logical array 형태로 내보낸다.
  - 디스크는 논리적 블락들을 cylinder, surface, track, sector로 매핑한다.
  - 데이터를 쓰거나 읽기 위해서는 오직 logical block #만 알면 된다.
  - 결과적으로 물리적인 파라미터 값들은 OS로부터 숨겨진다.
  - OS가 logical block #을 보내주면, 디스크 컨트롤러가 그걸 해석해서 실제적인 파라미터값을 가지고 디스크로 접근한다.

5. 디스크 성능
- 성능은 몇 개의 단계에 의해 결정된다.
- Seek : 디스크 arm을 찾고자 하는 cylinder로 옮긴다.
  - 얼마나 빨리 디스크 arm을 움직일 수 있느냐가 관건 (매우 천천히 증가)
- Rotation : sector가 head 아래에서 위치할 때까지 기다린다.
  - 디스크의 회전율이 관건 (증가하지만, 느린 편)
- Transfer : 데이터를 surface에서 디스크 컨트롤러로 보내고, 다시 호스트에게 보내는 과정이다.
  - 디스크의 바이트가 밀집되어 있는 것이 관건 (증가하고 매우 빠름)
- Disk scheduling
  - 탐색하는 과정은 비용이 많이 들기 때문에, OS는 디스크를 기다리고 있는 디스크 요청들을 스케줄링한다.
  - 탐색 거리(시간) 최소화가 중요하다.
  - 탐색 시간은 탐색 거리에 비례한다.
  - HDD 성능이 좋다는 말은 seek distance가 짧다는 것이다.

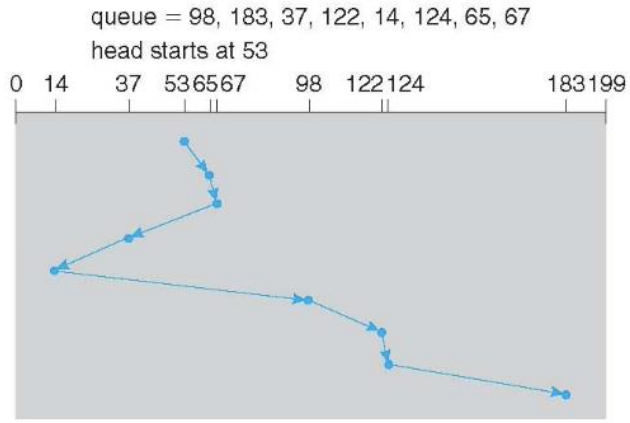
D. Disk scheduling

1. FCFS
- 큐에 들어온 순서대로 탐색한다.
- 성능이 좋지 않다.
  - Illustration shows total head movement of 640 cylinders



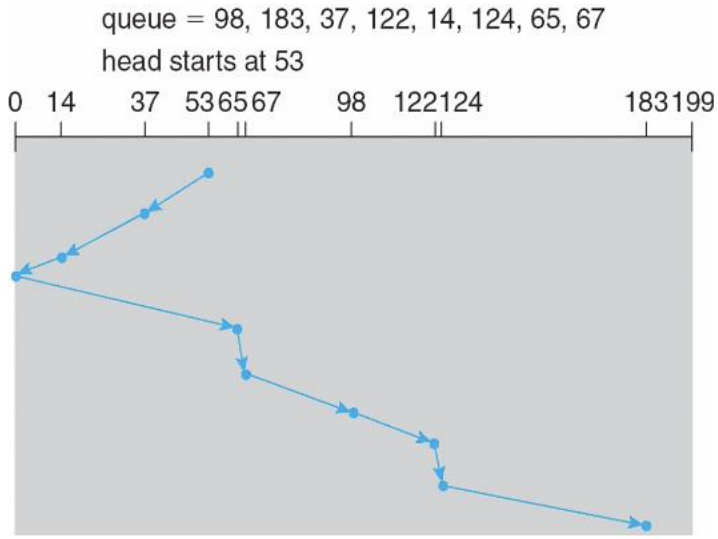
-

2. SSTF (Shortest Seek Time First)
- 현재 헤드의 위치에서 가장 짧은 탐색 거리를 갖는 요청을 선택한다.
- SJF 스케줄링의 형태
- 가장 멀리 있는 데이터의 경우, starvation이 발생할 수 있다. (Disk I/O 요청 프로세스가 많은 경우)
- 성능은 최적에 가깝다.
- HDD에서 SSTF의 개량형을 실제로 많이 사용한다. (성능 제일 좋다.)



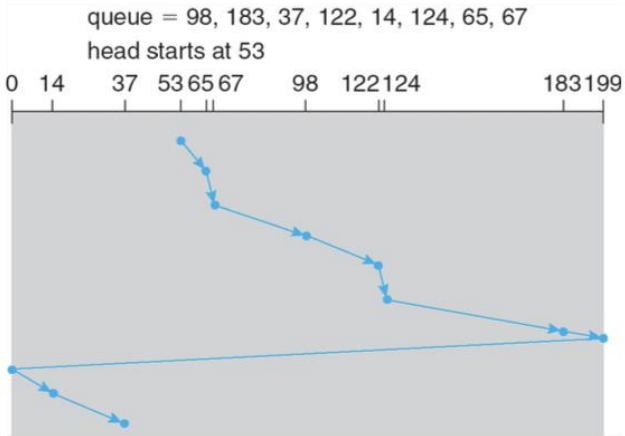
-

3. SCAN (elevator algorithm)
   - 엘리베이터처럼 내려갈 땐 쭉 내려가고, 올라갈 땐 쭉 올라가는 알고리즘
   - 내려가는 과정, 혹은 올라가는 과정에서 탐색해야 할 데이터를 모두 탐색하며 이동한다.
   - 탐색 거리의 합이 증가한다.
   - 지그재그로 움직이기 때문에 반대편 끝에 있는 데이터는 접근하는 데 엄청 오래 걸린다.



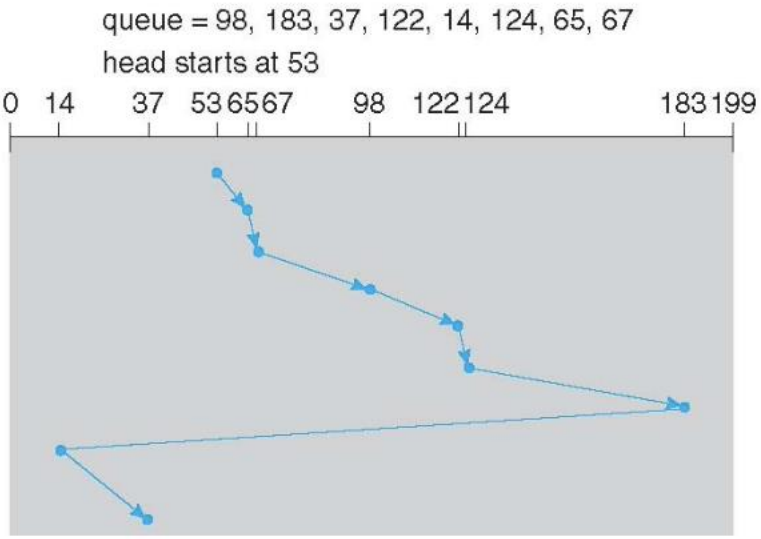
-

4. C-SCAN
   - 한 방향으로 이동할 때만 탐색하는 엘리베이터
   - 현재 위치에서 올라가면서 탐색해야 할 데이터를 모두 탐색하고, 마지막 위치에 도달하면 다시 디스크의 시작 위치로 돌아가서 위의 과정을 수행한다.
   - 찾을 데이터가 없는 구간 (시작 위치 근처, 마지막 위치 근처)까지도 움직여야 하므로 엄청 비효율적이다.
   - 사용되지 않는다.



-

5. C-LOOK
   - C-SCAN은 무조건 시작과 끝을 찍어야 하는 비효율이 존재한다.
   - 이를 보완하기 위해 무조건 시작 위치에서 출발해서 마지막 위치까지 도달하는 것이 아니라, 찾아야 하는 데이터 중 최소 위치 및 최대 위치 사이에서 C-SCAN과 같은 과정을 반복한다.



   -
6. 디스크 스케줄링을 선택하는 방법 (기준)
   - 성능은 디스크 I/O 요청의 수와 타입에 의해 결정된다.
     - SSTF가 가장 일반적인 접근법이다.
     - SCAN과 C-SCAN은 디스크에 많은 부하가 걸리는 시스템에서 성능이 좋다. (하지만 오버헤드가 너무 커서 사용하지 않는다.)
   - 일반적으로, 디스크 스케줄링은 요청이 많지 않으면 큰 의미가 없다. (서버 컴퓨터에서는 중요, PC에는 비교적 덜 중요)
   - 최근 디스크들은 스스로 디스크 스케줄링을 수행한다.
     - 디스크는 OS보다 디스크의 레이아웃에 대해 더 잘 알아서 최적화를 수행하기에 더 용이하다.
     - OS에 의해 수행된 스케줄링을 무시하거나 취소할 수 있다.

E. Disk Controllers (하드웨어)
1. 똑똑하다.
2. 최근에는 대부분의 디스크 컨트롤러가 작은 CPU와 함께 제작되고, 많은 킬로바이트의 메모리를 가진다.
3. 디스크 컨트롤러는 CPU로부터 받은 I/O 요청을 수행하기 위해 컨트롤러 제작자(사)에 의해 만들어진 프로그램을 실행한다.
4. 기능들
   - Read-ahead : Prepaging 개념. 현재의 "track"을 읽어온다. -> spatial locality
   - Caching : 자주 쓰이는 블락들을 캐싱한다.
   - Request reordering : 탐색 혹은 회전의 최적화를 위해 요청들을 재배치한다.
   - Request retry on hardware failure : 컨트롤러 단에서 핸들링 -> OS의 오버헤드를 줄일 수 있다.
   - Bad block identification : bad sector까지도 알아낼 수 있다.
   - Bad block remapping : 빈 블락이나 트랙에 블락을 다시 매핑한다.

F. Swap-Space Management
1. Swap-space
   - Virtual Memory(VM)가 디스크 영역을 메인 메모리의 연장선처럼 사용하는 것이다.
2. Windows
   - 파일 형태로 관리한다.

- 제어판 -> swap-space를 실시간으로 늘릴 수 있다.
- 신뢰성과 안전성은 감소, 유연성은 증가

3. Linux
- static하게 분리된 디스크 파티션을 가진다.
- 파티션 사이즈를 바꾸는 게 쉽지 않다. (유연성 bad)
- 보통 피지컬 메모리의 2배만큼 설정한다.
- 간섭이 적고 데이터 오염 확률이 적다. (신뢰성, 안전성 good)

4. Management
- 프로세스가 시작되면 4.3BSD가 swap-space를 할당한다.
  - text segment와 data segment를 담고있다.
- 커널은 swap maps를 참조해서 swap-space 정보 매핑을 한다.
- Solaris 2는 오직 페이지가 피지컬 메모리를 벗어날 때만 swap-space를 할당한다.
  - virtual memory 페이지가 생성될 때가 아니다.

II. RAID

A. Redundant Array of Inexpensive Disks

1. Inexpensive : 싸고 작은 칩 디스크를 여러 개 묶어서 사용한다.
2. Redundant : 여분의
3. 파일 시스템이 아니라 저장 시스템이다.
4. 단일 디스크 : 물리적인 특성으로 성능이 결정된다.
5. 단일 디스크의 성능을 좋게 만들기 위해 여러 개의 물리매체를 논리적으로 묶어서 사용하는 것이 RAID.
   - 여기서 말하는 성능이란?
     - 1. 속도 (read, write)
     - 2. 신뢰성 (단일 디스크에서는 고장나면 데이터가 다 날아간다.)
6. 두 가지 성능 목표 (reliability, performance)
   - Reliability via redundancy
     - Mirroring (shadowing) : 먹지처럼 디스크를 겹쳐서 사용 -> 용량이 많이 필요하다는 단점 존재
     - Parity or error-correcting codes : 복구 메커니즘을 구현
   - Performance via parallelism
     - Data striping (bit-level or block-level) : 물리적 버스를 따로 두어서 병렬 검색이 가능하다.
     - block-level이 더욱 효율적이다. (spatial locality)
     - 신뢰성은 보장할 수 없다.

B. SSD (Solid State Drive)

1. SSD의 구조
   - 많은 NAND FLASH 칩이 컨트롤러와 PCI 버스로 연결되어 있는 구조
   - 컨트롤러가 내장되어 있다.
   - 하나의 칩에는 여러 개의 (physical sector)가 존재한다.
   - 하나의 sector에는 여러 개의 block(최소 유닛)이 존재한다. (일반적으로 8개, 4KB size)
2. SSD의 block 업데이트
   - X.dat가 block을 5개 사용하고 있다고 가정하자.
   - 3번째 block의 한 글자를 수정하려면?
   - 3번째 block을 invalid 처리하고 새로운 block에 데이터를 쓴다. (덮어쓰기 X)
   - 업데이트가 많아지면 한 섹터가 모두 invalid가 생긴다.
   - 이 경우에는 block 단위로만 clear(valid 처리)가 가능하다.

3. SSD의 수명
   - Block의 클리어 횟수가 수명이다.
   - 한 block의 수명이 다하면 SSD를 통째로 쓰지 못한다.
   - 골고루 사용해서 클리어 횟수를 고르게 분포시켜야 한다. (Wear Leveling)
4. SSD의 NAND FLASH 집적도가 높으면 용량도 높다.
5. USB도 NAND FLASH를 사용하지만, 컨트롤러에 있어서 차이가 있다.
   - USB는 컨트롤러 없이(Wear leveling 없이) NAND FLASH를 사용하기 때문에 수명이 낮다.
6. HDD는 bad sector가 존재하면 컨트롤러가 해당 sector를 제외하고 사용하면 되게끔 관리해준다.

C. Multiple Disks vs. RAID
1. Disk Controller는 각 디스크마다 따로 관리한다.
2. RAID Controller는 여러 개의 디스크를 논리적으로 한 디스크처럼 관리한다.

D. RAID Levels (버전)
1. RAID 0
   - 미러링, parity 등 사용하지 않고, Data striping만 존재한다. (오직 성능만 고려, 신뢰성 X)
   - 데이터는 블록으로 여러 블록으로 쪼개져서, 각 블록은 여러 개의 디스크에 저장된다.
   - 여러 매체에 I/O load를 분산시킴으로써 I/O 성능이 좋다.
   - 고성능이 필요한 시스템에 사용된다. (Video editing, 스트리밍 서버 등)
   - 초창기에는 비트, 블록 레벨의 데이터 스트리핑 모두 가능하다. (뒤로 갈수록 정해져 있다.)
   - 한 디스크라도 고장나면 모두 쓰지 못한다. (신뢰성이 낮다.)

0	1	2	3
A0	A1	A2	A3
A4	A5	A6	A7
A8	A9	A10	A11
...	...	...	...

   -
2. RAID 1
   - 미러링 수행 (오직 신뢰성만 고려, 성능 X)
   - 같은 내용을 그대로 다른 디스크에 복사한다.
   - 실질적으로 디스크의 절반만 사용할 수 있다.
   - 수정, 삭제, 삽입을 할 때마다 두 번씩 진행해야 하므로 컨트롤러가 할 일이 많다.
     - 오버헤드, 비용이 크다.
   - 디스크 고장이 나면 백업본을 사용하면 된다. (복구 메커니즘이 따로 필요없다.)

0	1	2	3	4	5	6	7
A0	B0	C0	D0	A0	B0	C0	D0
A1	B1	C1	D1	A1	B1	C1	D1
A2	B2	C2	D2	A2	B2	C2	D2
...	...	...	...	...	...	...	...

   -

3. RAID 2

- Error-Correcting Codes (ECC)를 사용한다. (복구 메커니즘 -> 신뢰성)
- 각 데이터는 ECC 디스크에 Hamming 코드를 가지고 있다.
- 읽을 때, ECC를 통해 데이터가 정확한지 확인하고, 1비트의 에러를 수정할 수 있다.
- ECC는 요즘에 거의 모든 디스크 드라이브에 내장되어 있다. (ex. SCSI)
- 비트 레벨 데이터 스트리핑을 수행한다.

0	1	2	3	4	5	6
A0	A0	A0	A0	Px	Py	Pz
A1	A1	A1	A1	Px	Py	Pz
A2	A2	A2	A2	Px	Py	Pz
A3	A3	A3	A3	Px	Py	Pz
...	...	...	...	...	...	...

-
- 0,1,2,3 중 3장이 망가져도 복구가 가능하다. (ECC 디스크가 3장이 있기 때문에)
- ECC는 코드가 많고, 연산 비용이 낮다. (Parity bit보다는 안정적이다.)

4. RAID 3

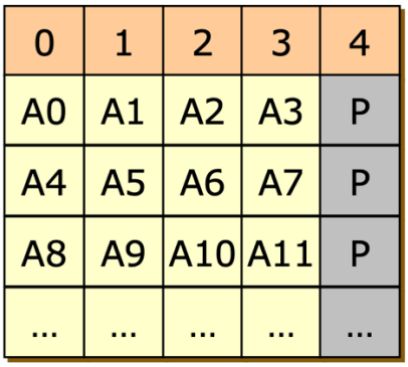
- 비트 레벨 스트리핑을 수행한다.
- Parity bit (1비트) 디스크를 가진다.
- 데이터를 쓸 때 parity bit를 생성한다.
- ECC 디스크 3장은 너무 많은 저장공간을 차지한다.
  - 연산량도 많다.
  - 따라서 parity bit를 사용한다.
- 고작 parity bit로 연산하려면 연산 비용이 높아진다. (적은 데이터 양과 많은 연산은 trade-off 관계)
- Parity bit는 하드웨어적으로 지원된다.

0	1	2	3	4
A0	A0	A0	A0	P
A1	A1	A1	A1	P
A2	A2	A2	A2	P
A3	A3	A3	A3	P
...	...	...	...	...

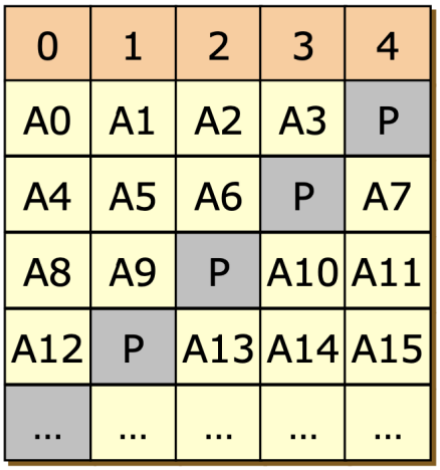
-
- 문제점 : Parity bit가 있는 디스크가 망가지면 어떻게 복구하는지?

5. RAID 4

- 블록 레벨 데이터 스트리핑을 수행한다. (spatial locality로 인해 성능이 좋다.)
- Parity는 같은 층위의 블록들이 쓰일 때 생성된다.
  - 이후 읽을 때 parity를 확인해서 오류를 감지한다.
- RAID 0처럼 읽기 성능이 좋다.
- 하지만 쓸 때마다 parity까지 갱신되어야 한다.
- Parity 데이터는 스트리핑을 하지 않는다.
  - 복원할 때 데이터가 적기 때문에 속도가 비교적 느리다.
  - 복원 가능성도 비교적 낮다.

- 여러 디스크에 동시에 write을 하는 기능도 지원되지 않아서 RAID 5보다 장점이 없다.
- 

0	1	2	3	4
A0	A1	A2	A3	P
A4	A5	A6	A7	P
A8	A9	A10	A11	P
...	...	...	...	...

6. RAID 5
   - 최소 디스크 3장이 필요하다.
   - 블록 단위 데이터 스트리핑을 수행한다.
     - Parity bit도 여러 디스크에 데이터 스트리핑
   - 같은 층위의 블록들에 데이터가 쓰일 때, Parity bit도 생성된다.
   - 멀티프로세스 시스템에서 자잘한 write을 하는 데 속도를 올릴 수 있다.
     - Parity disk에서 병목현상이 없기 때문이다.
   - 

0	1	2	3	4
A0	A1	A2	A3	P
A4	A5	A6	P	A7
A8	A9	P	A10	A11
A12	P	A13	A14	A15
...	...	...	...	...

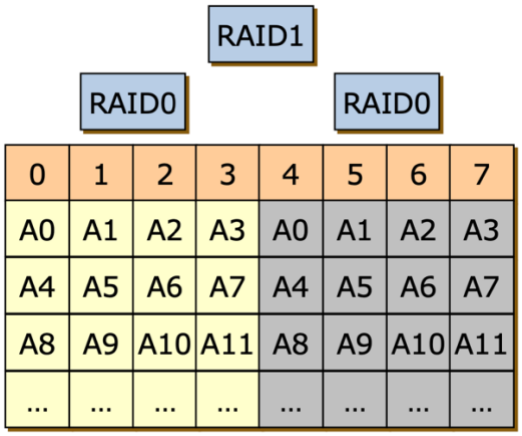
   - 3장 중 1장이 고장나도 parity bit를 활용한 odd numbering을 통해 고장난 디스크를 복원할 수 있다.
   - 속도와 신뢰성이 좋은 편이다.
     - 그런데 Parity bit는 복원력에 있어서 ECC보다 떨어진다.

7. RAID 6
   - 최소 4장의 디스크가 필요하다.
   - Parity bit가 아니라 2개의 ECC를 통해 복구를 수행한다.
     - 복구 안정성이 높아진다.
   - RAID 5의 2배만큼 복원 메커니즘을 가지고 있으므로, 2장까지는 고장나도 복원할 수 있다.
   - RAID 5에 덧붙여서 여러 디스크가 고장났을 때 복원이 가능하다.

0	1	2	3	4	5
A0	A1	A2	A3	Px	Py
A4	A5	A6	Px	Py	A7
A8	A9	Px	Py	A10	A11
A12	Px	Py	A13	A14	A15
...	...	...	...	...	...

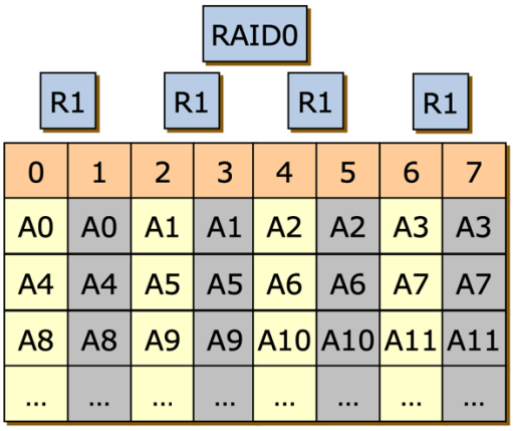
-

8. RAID 0+1
   - RAID 0(데이터 스트리핑)을 먼저 적용하고, 그 위에 RAID 1(미러링)을 적용한다.
     - RAID 0 -> 성능, RAID 1 -> 신뢰성
   - 하나의 디스크가 고장나면, 같은 RAID 0에 해당하는 디스크들도 더불어 못쓰게 된다.
     - 데이터가 여러 디스크에 스트리핑되어 있기 때문이다.
     - 사실상 RAID 1(미러링)의 효과를 누리지 못하게 된다.



-

9. RAID 1+0 (10)
   - 먼저 두 디스크씩 쌍으로 RAID 1(미러링)의 형태를 보인다.
   - 그 뒤로 RAID 0의 형태로 데이터 스트리핑을 적용한다.
   - 성능은 RAID 01과 비슷하다.
   - 신뢰성은 RAID 01보다 좋다.



-

III. I/O Systems

A. I/O request via I/O instruction (CPU에서 수행되는 명령어)

1. 명령어가 수행되는 과정에서 데이터가 이동한다. (저장공간이 필요하다.)
2. Direct I/O

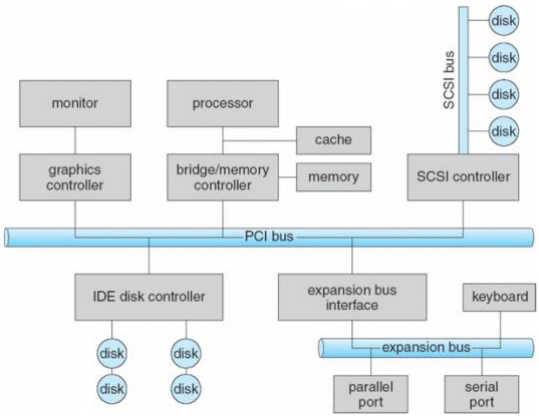
- 메모리와 I/O 디바이스가 다른 주소 체계를 가진다. (separate addresses)
- 각각의 I/O 디바이스에 따른 인스트럭션과 레지스터를 각각 가진다. (separate instructions)
  - 메모리는 따로 특별관리 (MEMR / MEMW -> LOAD / SOTRE)
  - 메모리에 대해서 빠른 처리가 가능하다.
  - 인트럭션과 레지스터의 수가 많이 필요하면, 데이터 버스가 따로 존재하기 때문에 물리적으로 구현하기 힘들다. USB는 그래서 통합 컨트롤러를 활용한다.
  - CPU의 구조가 복잡해지면 가격이 올라간다.
- IOR / IOW (IN / OUT)

3. Memory-mapped I/O
   - 메모리와 I/O 디바이스가 같은 주소 체계를 사용한다. (Integrated addresses)
   - MEMW / MEMW만 존재한다. (Integrated instructions)
     - 가장 많이 사용하는 I/O 명령이 메모리 관련이다.
     - 인스트럭션을 한 세트로 해결할 수 있기 때문에, CPU 아키텍처 복잡도가 낮다.
     - Disk에 데이터를 쓰고 싶을 땐 메모리에 쓴 뒤에 메모리를 거쳐서 디스크에 쓴다. (indirect한 방법)
     - 메모리에서 디스크 컨트롤러로 정보를 넘겨주면(linking), 컨트롤러가 입출력을 수행한다.

B. I/O method

1. Programmed I/O : CPU가 직접 입출력 데이터를 전송해서 바로바로 처리하는 것
   - CPU 오버헤드가 크다.
   - 입출력이 완료될 때까지 CPU는 다른 작업을 수행하지 못한다.
2. Interrupt : 입출력 작업이 완료되면 CPU로 보내는 신호
   - CPU는 입출력 작업이 진행되는 동안 다른 작업을 수행할 수 있다.
3. DMA (Direct Memory Access)
   - 입출력 데이터의 양이 많은 경우 CPU를 거쳐서 CPU가 I/O를 핸들링하면 CPU 오버헤드가 크다.
   - CPU가 처리해주는 과정을 컨트롤러한테 넘긴다.
   - 그래서 CPU를 거치지 않고 I/O 디바이스에서 메모리로 바로 입출력 데이터 전송이 가능하다.
   - 디스크 컨트롤러에서 DMA 컨트롤러로 요청을 보내면, DMA 컨트롤러가 CPU의 허락을 받아서 대량의 I/O 작업을 처리한다.

C. PC Bus Structure

1. 구조
   -


2. 각 디스크들은 RAID로 묶여서 사용된다.
3. PCI 버스의 대역폭이 크면 I/O 속도가 빠르다.

D. Device Controller (host adapter)

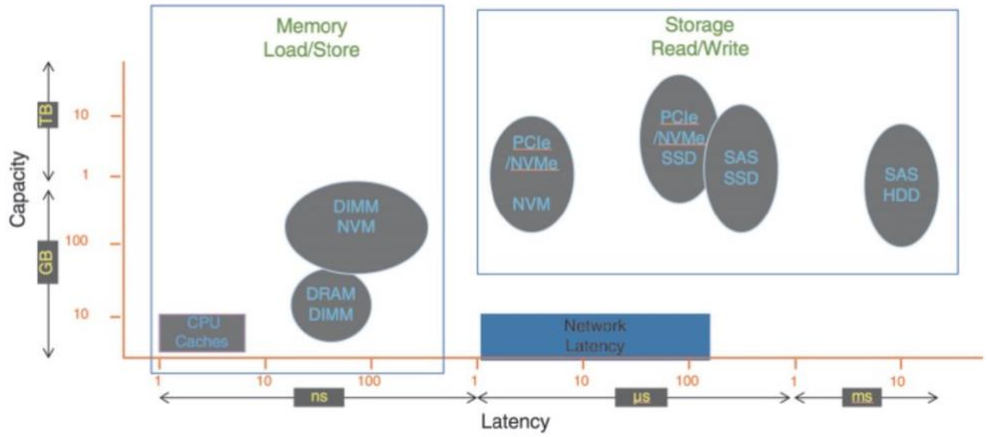
1. I/O 디바이스들은 기계적 부분과 전기적 부분으로 나뉜다.
2. 전기적 부분은 디바이스 컨트롤러로, CPU가 할 일을 더 빠르게 대신 수행한다.

- 여러 개의 디바이스를 관리할 수도 있다.

3. 컨트롤러가 맡는 일
   - 직렬 비트 스트림으로 받은 데이터를 블록 단위로 변환한다. (관리 효율성을 위해)
   - 필요한 경우 에러를 수정한다.
   - 메인 메모리에 입출력이 가능하게 한다.

E. I/O Performance & Network Latency

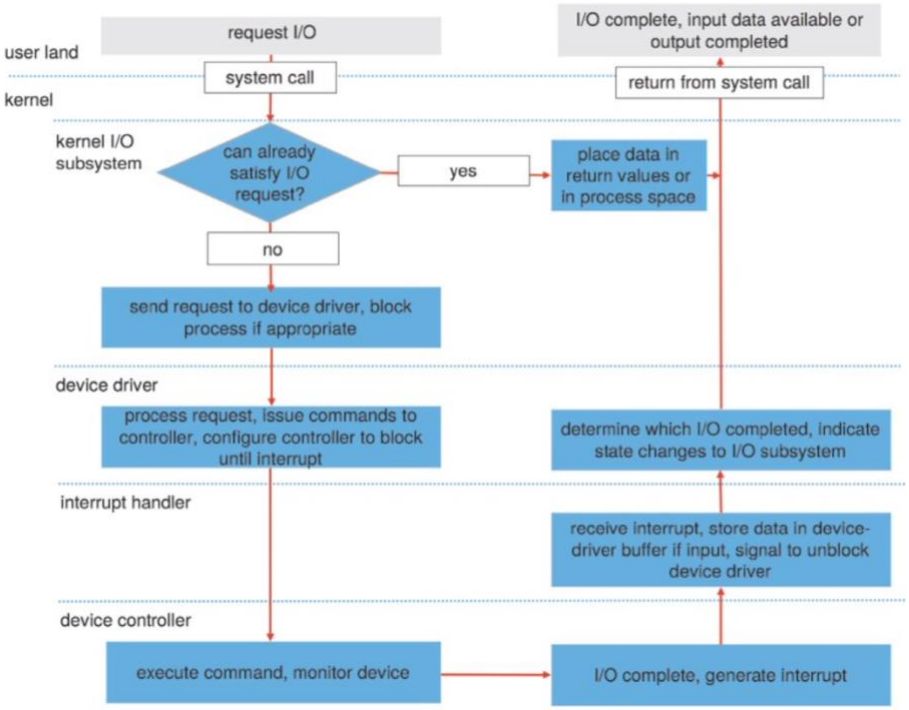
1. 메모리 & storage 관점
2. Capacity (용량)
   - CPU 캐시 < DRAM < NVM < HDD < SSD
3. Network Latency (지연시간)
   - CPU 캐시 < DRAM < NVM < SSD < HDD



F. Kernel I/O Structure

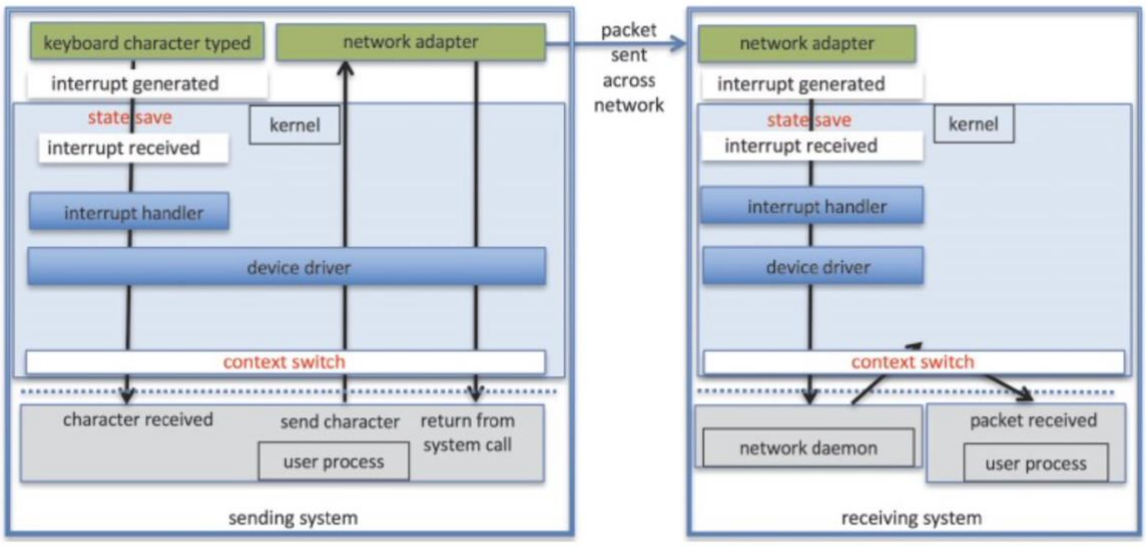
1. 구조 (SW -> HW)
   - kernel (HW 이전까지는 모두 SW)
   - kernel I/O subsystem
     - I/O 요청이 빈번하게 일어나게 되면, 중복되는 I/O 명령이 발생할 수 있다.
     - 중복 명령을 모두 수행하게 되면 불필요한 로드가 발생한다.
     - 따라서 이미 요청한 명령이 완료되었는지 체크하는 역할을 한다.
   - 각종 I/O 디바이스 드라이버
     - application에서 내려온 어떤 형태의 데이터를 컨트롤러에 적합한 데이터로 변환해서 컨트롤러로 전송
     - 컨트롤러로부터 받은 데이터를 처리해서 어느
   - 각종 I/O 디바이스 컨트롤러 (여기서부터 HW)
     - I/O 디바이스에 작업 수행을 명령한다.
   - 각종 I/O 디바이스

G. Life Cycle of An I/O Request



1. 유저단에서 I/O 요청을 하면, 커널로 시스템 콜을 보낸다.
2. 커널 I/O 서브시스템에서 이미 수행한 I/O 요청인지 확인한다.
   - 이미 완료된 요청이라면, 해당 데이터를 리턴값으로 유저단으로 전송한다.
   - 완료되지 않은 I/O 요청이라면, 필요한 디바이스의 드라이버로 요청을 전송하고, 필요하다면 프로세스를 block시킨다.
3. 디바이스 드라이버에서는 요청을 처리하고, 명령을 컨트롤러로 전송한다.
   - 컨트롤러가 해석할 수 있는 형태로 처리해서 보낸다.
4. 디바이스 컨트롤러에서는 명령을 실행하고, 디바이스를 관리한다. (모니터링)
5. 디바이스 컨트롤러는 요청이 완료되면 인터럽트를 발생시킨다.
6. 인터럽트 핸들러에서는 인터럽트를 받아서, 인터럽트가 입력일 경우(키보드, 마우스 등)에는 디바이스 드라이버 버퍼에 데이터를 저장하고, block 상태를 해제하는 신호를 보낸다.
7. 디바이스 드라이버에서는 I/O가 완료되었는지 결정하고, I/O 서브시스템한테 상태 변화를 알린다.
8. 서브시스템은 커널을 거쳐서 유저단에 리턴값을 전송한다.

H. Intercomputer Communications



1. 채팅 애플리케이션 같은 경우는 I/O가 매우 많이 일어난다.
2. 키보드로 입력이 발생하면 인터럽트가 발생한다.
   - 키보드나 마우스 같은 입력의 경우 programmed I/O (CPU가 바로바로 처리)방식을 사용한다.
3. 커널에서 인터럽트를 받으면, 먼저 프로세스의 상태를 저장하고 인터럽트 핸들러와 디바이스 드라이버를 지나서 context switch를 수행한다.
4. CPU에서 커널을 거쳐서 입력값을 network adapter(I/O 디바이스)로 보낸다.
5. Network adapter에서 패킷 형태로 연결된 네트워크를 따라서 수신자의 시스템으로 보낸다.
6. 수신자의 시스템 내에서 network adapter가 패킷을 받으면, 인터럽트를 발생시킨다.
7. 송신자의 시스템과 같이 인터럽트를 발생시키고, 인터럽트 핸들러와 디바이스 드라이버를 거쳐서 context switch를 발생시킨다.
8. 이후 network daemon을 거쳐서 수신자의 프로세스에 패킷이 도달한다.

I. Device-Functionality Progression

1. application code, kernel code, device-driver code, device-controller code (hardware), device code (hardware)
   - device-controller code (hardware) – SSD 컨트롤러
   - device code (hardware) – 그래픽 카드 (Parallel computing 가능)
   - HW단 코드는 수정하기가 어렵다.
   - HW단 코드는 전기신호로 작동하기 때문에 실행 속도가 빠르다.
2. HW단으로 갈수록 증가하는 것
   - 생성 시간 (코드 변환에 시간이 걸린다.)
   - 효율성
   - 비용
   - 추상화 정도 (HW -> 전기신호 -> 사용자가 몰라도 동작한다.)
3. SW단으로 갈수록 증가하는 것
   - 유연성
   - SW로 코드를 짜는 것이 오히려 시간이 적게 든다. (HW단 코드는 복잡하다.)

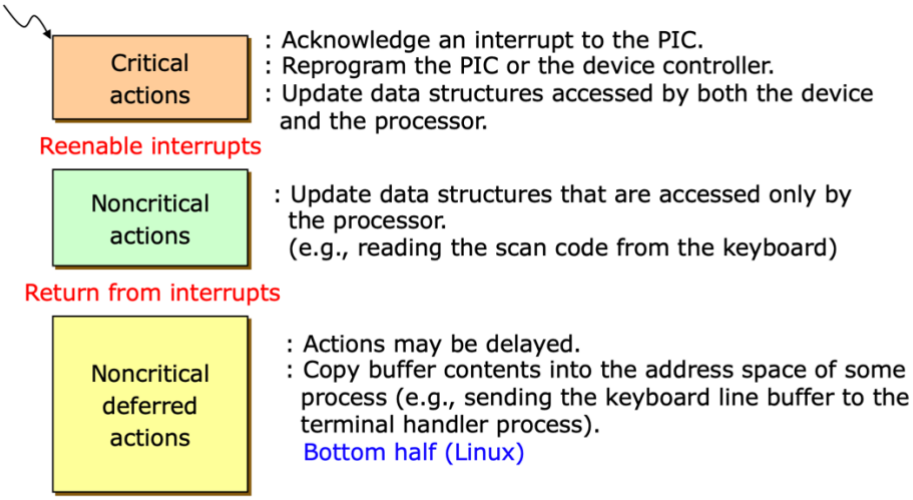
J. I/O Software의 목적

1. Device independence
   - 프로그램은 사전에 디바이스에 대해 자세히 몰라도 어느 I/O 디바이스에도 접근 가능하다.
2. Uniform naming
   - 파일이나 디바이스의 이름은 문자열이나 정수여야 한다.
3. Error handling
   - 최대한 하드웨어와 물리적으로 가까운 위치에서 에러를 관리해야 한다.
4. Synchronous vs. asynchronous
   - Blocked transfers : 데이터를 받기 전까지는 다른 작업을 수행하지 않고 대기
   - Interrupt-driven : 인터럽트를 받기 전까지 다른 작업을 수행하다가 인터럽트를 받으면 데이터를 받아서 처리
5. Buffering
   - 디바이스로부터 나온 데이터를 application단에 바로 저장할 수 없는 경우, 중간에 버퍼를 이용해서 저장하고 전달한다.
6. Sharable vs. dedicated devices
   - Sharable : 여러 프로세스가 동시에 공유할 수 있는 디바이스
   - Dedicated : 정해진 프로세스만 사용할 수 있는 디바이스
   - Disks vs. tape drives

- 공유할 수 없는 디바이스는 데드락과 같은 문제가 발생한다.

7. Layers 구조 (상단부터 하단 순서)
   - User-level I/O SW
   - Kernel level
     - Device-independent I/O SW
     - Device drivers
     - Interrupt handlers
   - Hardware

K. I/O Software의 Layer 구조



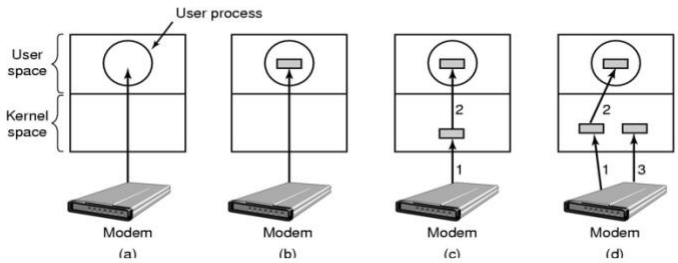
1. Layers 구조 (상단부터 하단 순서)
   - User-level I/O SW
   - Kernel level
     - Device-independent I/O SW
     - Device drivers
     - Interrupt handlers
   - Hardware
2. Interrupt Handlers (심각성에 따른 분류)
   - Critical actions
     - PIC(Program Interrupt Controller)에게 인터럽트를 인지시키고, PIC 혹은 디바이스 컨트롤러를 다시 재구성하여 다음에 발생할 인터럽트를 처리할 수 있도록 한다.
     - 디바이스와 프로세서가 접근하는 데이터 구조를 갱신한다.
     - Reenable interrupts : 중단된 인터럽트를 재개할 수 있도록 한다.
   - Noncritical actions
     - 프로세서에 의해서만 접근되는 데이터 구조를 갱신한다.
     - Return from interrupts : 인터럽트 처리가 끝나고 돌아온다.
   - Noncritical deferred actions
     - 실행이 지연될 수 있다.
     - 버퍼 내용을 어떤 프로세스의 주소 공간에 복사한다.
     - Bottom half (Linux) : 인터럽트의 처리를 지연시켜 복잡한 작업을 나눠서 처리하는 방법이다.
3. Device Drivers
   - I/O 디바이스를 통제하기 위한 디바이스 특화된 코드는 device-independent I/O SW와 인터럽트 핸들러

사이에서 상호작용한다.

- 디바이스 드라이버가 OS의 남은 부분과 상호작용하는 표준 인터페이스와 잘 정의된 모델을 정의할 필요가 있다.
- 구현 관점
  - 정적으로 커널과 linked되어 있는 방법
  - 부팅하는 동안 시스템 안에 선택적으로 로드되는 방법
  - 실행하면서 동적으로 시스템 안에 로드되는 방법

4. Device-Independent I/O Software (서브시스템)

- 유닉스는 모든 I/O 디바이스를 파일로 관리한다.
  - 유닉스에서는 소켓도 파일로 관리되고, 리턴값인 정수를 파일 디스크립터로 사용해서 소켓을 관리한다.
  - I/O 디바이스는 시스템 콜을 통해 접근된다.
  - 파일명은 각 디바이스와 관련이 있다.
- 주요 디바이스 번호는 적절한 드라이버에 위치한다.
  - 중요하지 않은 디바이스 번호는 드라이버로 파라미터로 넘어간다.
- 일반적인 파일에 대한 protection 룰은 I/O 디바이스에도 적용된다.
- Buffering
  - ✓ (a) Unbuffered
  - ✓ (b) Buffered in user space
  - ✓ (c) Buffered in the kernel space
  - ✓ (d) Double buffering in the kernel

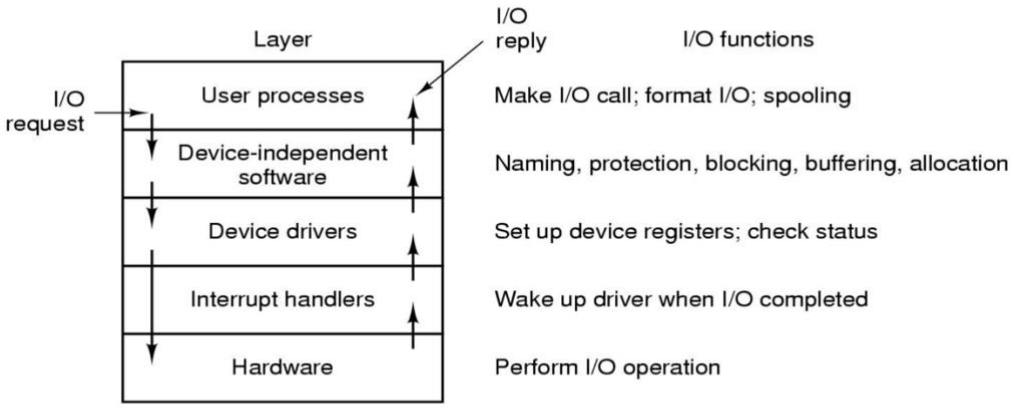


-
  - 디바이스마다 다를 수 있다.
  - 버퍼는 변수 하나를 의미한다. (ex. 큐)
  - Unbuffered (버퍼 사용 X)
  - Buffered in user space (유저단에 버퍼 존재)
  - Buffered in the kernel space (커널에 있는 버퍼를 거쳐서 유저 버퍼에 복사)
  - Double buffering in the kernel (커널에 백업용 버퍼가 하나 추가)
- Error reporting (서브시스템에서 에러 받으면 어떻게 처리하는지?)
  - 많은 에러들이 디바이스 특화되어 있고, 그에 따라 적절한 드라이버에 의해 핸들링을 수행해야 한다.
  - 프로그래밍 에러와 실제 I/O 에러로 나뉜다.
  - Returning the system call with an error code : 크게 중요한 에러가 아닐 경우 에러 코드와 함께 보내주면서, 애플리케이션단에서 해결
  - Retrying a certain number of times : 특정 횟수만큼 재시도 (ex. ARQ : 재전송 프로토콜)
  - Ignoring the error : 에러 무시
  - Killing the calling process : 강제 종료
  - Terminating the system : 시스템 종료 (ex. watch dog : 시스템이 멈춘 상태에서 타이머 작동 -> 특정 시간이 되면 재부팅 수행)

- 특화된 디바이스에 대해서 적용하는 법
  - 프로세스들이 직접적으로 디바이스의 특정한 파일에 open()을 수행하도록 요구된다.
    - open()에 실패하면 재시도
  - 특화된 디바이스는 전용 메커니즘이 서브시스템에 구현된다.
- 디바이스에 상관없는 블록 사이즈
  - 어느 정도 합의점이 존재한다.
  - 여러 섹터를 하나의 논리적인 블록으로 간주한다. (abstraction)
  - 상위 계층에서는 추상화된 디바이스(모두 같은 블록 사이즈 사용)에만 접근한다.

L. User-Space I/O Software

1. 라이브러리 형태로 제공된다.
2. Spooling
   - 멀티프로그래밍 시스템에서 특화된 I/O 디바이스를 핸들링하는 방법이다.
   - daemon과 spooling directory에 의해 구현된다.
   - 프린터(대표적으로 느린 I/O 디바이스), network file transfers, USENET news, mails, 등
   - I/O 출력할 때 필요한 데이터를 메모리에 쓰지 않고 파일로 만든다.
     - 어차피 파일로 만들어서 디스크를 사용하는 것이 프린터의 속도보다 빨라서 괜찮다.
3. I/O Systems Layers 총체적 관점(구조)



-
- Request가 디바이스로 갈 땐, 인터럽트 핸들러를 거치지 않는다.

IV. File system interface

A. 파일시스템의 개념

1. 메인 메모리와 다르게 long-term 정보를 저장하는 장소 (요구조건)
   - 상당히 큰 크기의 정보를 저장할 수 있어야 한다.
   - 파일 오픈 정보는 프로세스가 종료될 때까지 유지되어야 한다.
   - 다수의 프로세스는 해당 정보에 동시에(병렬적으로) 해당 정보에 접근할 수 있어야 한다.
2. File – secondary storage(HDD, SSD 등)에 저장할 때 추상화를 제공하는 방법이다.
3. Directories – 파일들을 논리적인 구조로 정리하는 방법을 말한다.
4. Sharing – 여러 프로세스, 사람, 기계들이 파일 데이터를 공유할 수 있도록 해준다. (서버급, 리눅스에서는 여러 사용자의 권한이 매우 중요하다.)
5. Protection – 권한이 없는 접근을 막는다.

B. File

1. Secondary storage에 저장된 정보들을 관련된 것들끼리 묶어서 이름지어놓은 것이다.

2. OS는 파일을 통해 저장된 정보들의 논리적인 구조를 제공한다.
3. 파일 구조
   - Flat : byte sequence
     - 단순한 형태
     - 데이터가 바이트의 연속 형태
   - Structured
     - Lines – 데이터가 라인(레코드)로 구분된다.
     - Fixed length records : 각 레코드가 정해진 크기로 저장된다.
     - Variable length records : 각 레코드의 길이가 가변적이다.
4. File Attributes
   - OS -> File system -> file마다 정보 존재 (PCB와 비슷)
   -

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

5. File Operations
   - Unix/Linux system call을 통해서 구현된다. (C언어 기준)
   - creat, open, close, read, write
   - lseek – 파일 디스크립터를 원하는 임의의 위치로 옮기는 것
   - stat – 파일 상태값
   - chmod – 모드 변환
   - chown – ownership 변경
   - flock – 잠그는 것
   - fcntl – 파일 디스크립터, 속성에 대한 조작
6. File Types
   - 파일의 존재를 누가 알 수 있는가?
     - 파일 시스템 (디바이스, 디렉토리, 심볼릭 링크 등)
     - OS 혹은 런타임 라이브러리 (실행가능파일, dll, 소스코드, object 코드, text 등)
     - Application programs (jpg, mpg, mp3 등) – 그림판, 미디어플레이어 등 OS에서 제공해주기도 한다.
   - Encoding file types
     - Windows – 확장자 개념 사용 (in name) -> .com, .exe, .bat, .dll, .jpg, .avi, .mp3
     - Unix – 확장자의 개념이 아니라 컴파일러와 사용자 간의 약속 -> magic numbers, initial characters
7. File Access
   - 파일 시스템마다 다른 접근 방법을 정의한다.
   - Sequential access
     - 바이트 단위로 순서대로 읽는 방법
   - Direct access (= random access)
     - block이나 byte 단위로 random access
   - Record access

- 파일은 불변 혹은 가변 길이의 레코드의 배열 형태다. sequentially 혹은 randomly하게 읽고 쓴다.

- Index access
  - 파일 시스템은 파일 안에 각 레코드의 특정 필드마다 인덱스를 포함한다.
  - 시스템은 인덱스를 통해 레코드를 찾는다.
  - 인덱스 파일을 따로 저장해서 관련된 파일의 원하는 레코드를 찾을 수 있다. (인덱스 파일을 저장하는 데 용량이 더 들지만, 성능이 더욱 좋다.) -> 용량이 큰 파일을 모두 찾을 필요가 없기 때문에

C. Directories

1. 사용자들을 위해 파일들을 구조적으로 정리하는 방법을 제공해준다.
2. 파일 시스템을 위해 디스크에 저장된 물리적 파일의 위치를 논리적 파일 조직으로 매핑하는 편리한 인터페이스를 제공한다. (convenient naming interface)
3. Hierarchical Directory System
   - 대부분의 파일 시스템은 멀티레벨 디렉토리를 지원한다.
   - 대부분의 파일 시스템은 현재 디렉토리의 개념을 지원한다.
     - 현재 디렉토리 기준으로 relative names
     - 루트 디렉토리부터 시작하는 absolute names
4. Directory Internals
   - 디렉토리는 일반적으로 특별한 메타데이터를 포함하는 파일이다.
   - list of (file name, file attributes) -> 정렬이 큰 의미가 없다. (random access)
   - file attributes
     - 크기(용량), protection, 생성 시간, 접근 시간, 디스크 상의 위치(파티션 정보까지 포함)
5. Directory Operations
   - 디렉토리는 파일 안에 구현된다.
     - 디렉토리를 조작하기 위해서 file operations를 사용한다.
   - C언어 런타임 라이브러리가 디렉토리를 읽기 위한 높은 수준의 추상화를 제공한다.
     - 구조체, 함수 제공
     - opendir, readdir, seekdir, closedir
   - 디렉토리 관련 시스템 콜
     - rename, link(symbolic link 관련), unlink
6. Pathname Translation
   - open("/a/b/c", ...)
     - "/" – delimiter (Unix/Linux)
     - "₩" – delimiter (Windows) -> 스트링 안에 적을 땐 역슬래시를 두 개 적어야 인식한다.
     - a 디렉토리의 b를 찾아서 b의 위치를 얻는다.
     - b 디렉토리의 c를 찾아서 c의 위치를 얻는다.
     - C 파일을 연다.
     - 모든 과정에서 permission check가 수행된다.
   - 시스템은 디렉토리 경로를 따라가는 것에 많은 시간을 쓴다.
     - open이 read/write가 구분되는 이유
     - OS는 성능 강화를 위해 자주 방문하는 디렉토리(prefix lookups)를 캐싱한다.
7. Mounting
   - Windows : letter를 활용 (C:/ , D:/ 등)
   - Unix/Linux : 기존의 디렉토리 활용 (mount point)

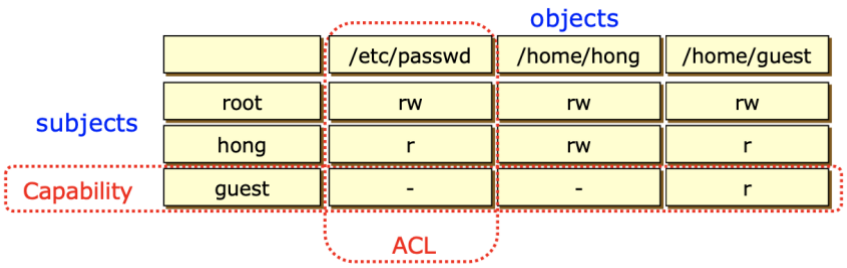
    - 시스템 전체에서 root 경로는 1개만 존재한다.

D. File Sharing

1. Remote File Systems (NAS 등)
2. 클라이언트-서버 모델은 클라이언트가 서버로부터 원격의 파일 시스템에 접근할 수 있게 해준다.
    - 서버는 여러 클라이언트한테 제공할 수 있다.
    - 클라이언트와 클라이언트를 사용하고 있는 사용자 식별은 복잡하다.
    - NFS(Network File System)은 Unix의 클라이언트-서버 파일 쉐어링 프로토콜의 표준이다.
    - CIFS(Common Internel File System)은 Windows 프로토콜의 표준이다.

E. Protection

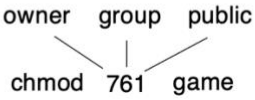
1. Access Control Lists (ACLs)
    - 각 오브젝트(파일)마다 주체와 허용된 액션들의 리스트를 가지고 있다.
    - 효율이 좋아서 사용한다.
        - 계정보다는 파일이 훨씬 많을 확률이 높다.
2. Capabilities
    - 각 주체(사용자, 계정)마다 접근할 수 있는 오브젝트와 허용된 액션들의 리스트를 가지고 있다.
    - 각 주체는 그룹으로도 가능하다.



    -
3. ACLs vs. Capabilities
    - 테이블을 어떻게 읽을 것인지에 대한 점이 다르다.
    - Capabilities
        - key처럼 쉽게 전달이 가능하다.
        - 공유가 쉽다.
    - ACLs
        - 관리하기가 쉽다.
        - 객체 중심 -> 권한을 부여하고 취소하는 것이 쉽다.
        - Capability를 취소하려면, 해당 capability를 가지는 모든 주체를 탐색해야 한다.
    - ACLs는 오브젝트가 많이 공유되는 경우에 ACL 사이즈가 커질 수 있다.
        - Groups를 이용해서 간단히 나타낼 수 있다.
        - 그룹 멤버를 변경함으로써 해당 그룹 안의 모든 객체에 접근할 수 있다.
4. Access Lists in Unix/Linux
    - read, write, execute 접근 모드가 있다.
    - 유저에는 3가지 클래스가 있다. (user / group / others)
    - 8진수로 나타내며, 2진수로 변환해서 RWX 각각에 대한 권한을 나타낸다.

	RWX		
a) **owner access**	7	⇒	1 1 1
b) **group access** 6	RWX ⇒	1 1 0	
c) **public access** 1	RWX ⇒	0 0 1	

- For a particular file (say *game*) or subdirectory, define an appropriate access



- $ ls - l
  - ls 명령어 -> 이름만 나옴
  - -l (optional) -> 파일에 대한 권한 정보, 그룹, 용량, 생성날짜 등이 나옴
  - 첫 글자가 "-" : 일반 파일
  - 첫 글자 "d" : 디렉토리

5. Access Lists in Windows
   - 파일 속성에서 확인 및 권한 설정 가능
6. Memory-Mapped File
   - 디스크에 있는 파일을 메모리 안의 버퍼에 매핑한다.
   - mmap() 시스템 콜 사용
   - read, write 없이 바로 I/O를 수행한다.
   - stack과 heap 사이에 공간을 할당한다. (start address, length)를 활용한다.

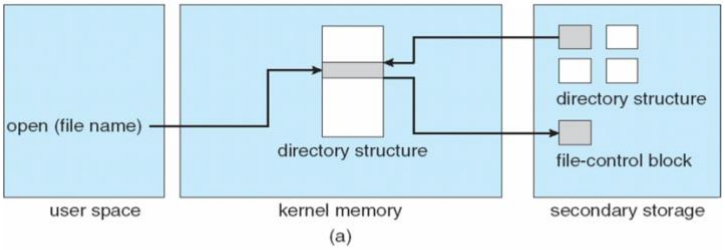
V. File system implementation

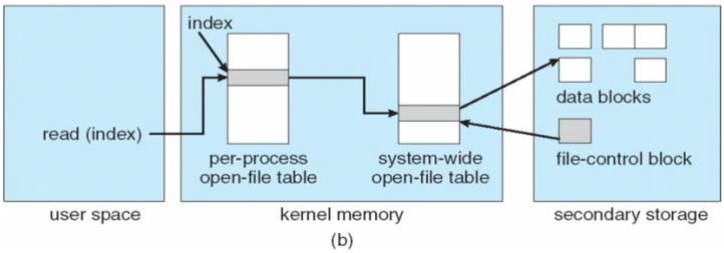
A. User's view vs. Implementer's view

1. 유저 관점
   - 파일의 이름은 어떻게 지어지는지?
   - 유저 입장에서 어떤 연산(액션)을 수행할 수 있는지?
   - 디렉토리 트리가 어떻게 생겼는지?
2. 구현하는 사람 관점
   - 파일과 디렉토리는 어떻게 저장되는지?
   - 디스크 공간은 어떻게 관리되는지?
   - 어떻게 효율적이고 안정적으로 관리하는지?

B. File System Implementation

1. 파일 시스템이 데이터를 관리하는 구조
2. In-memory structure
   - In-memory partition table
     - 디스크의 파티션에 대한 정보를 담고있는 테이블
     - 메모리에 저장
   - In-memory directory structure
     - 메모리에 올라와있는 디렉토리 구조
     - 실행 중인 프로세스가 데이터에 접근할 때 활용
   - System-wide open file table
   - Per-process open file table



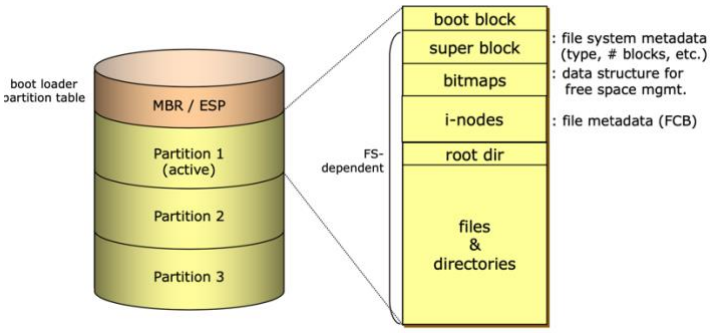


-
- open(file_name) : 시스템 콜
  - file descriptor를 반환(인덱스, 벡터)
  - 커널을 통해서 secondary storage
- read(index)
  - per-process open-file table은 각 프로세스마다 PCB에 존재한다.
  - open()을 통해 반환된 인덱스 값을 통해 해당하는 데이터 블록에 접근한다.
- directory cache
  - 자주 사용되는 경로 캐싱
- buffer cache
  - 실제 파일 내용 중 자주 사용하는 부분 캐싱
- Virtual File System (VFS)
  - 하나의 파일 시스템 하나로 모든 걸 해결할 수 없다.
  - 클래스 상속 개념으로 공통적인 기능만 VFS에 넣어두고, 필요한 파일 시스템들을 따로 구현한다.
- Layered File System (상위 레벨부터 나열)
  - application programs
  - logical file system -> 물리적인 매체(디스크)를 프로그램 상에서 추상화된 상태로 이용
  - file-organization module
  - basic file system
  - I/O control
  - devices

3. On-disk structure
- Boot Control Block
  - 부팅과 관련된 정보를 담고있다.
  - UFS에서는 Boot block, NTFS에서는 Boot sector
- Volume Control Block
  - 디스크 전체의 볼륨에 대한 메타데이터를 담고있다.
  - Super block(UFS), Master file table(NTFS)
- Directory structure
  - 파일 및 디렉토리 계층구조를 담고있다.
- File Control Block (FCB)
  - 각 파일에 대한 정보를 담고있다.

- i-node(UFS), Master file table 안에(NTFS)

4. On-disk structure 한눈에 보기



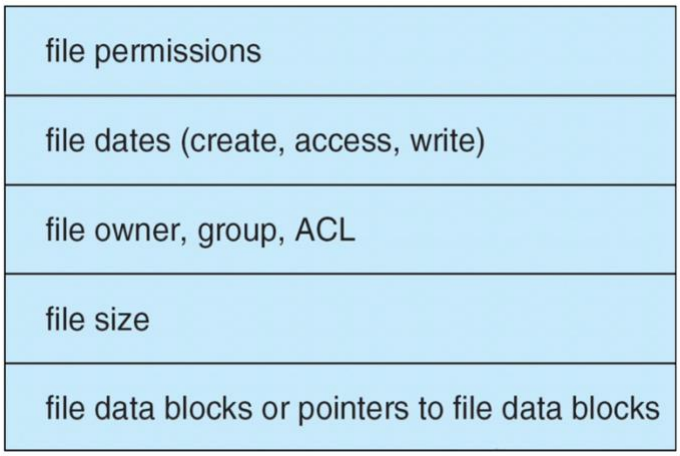
-
- MBR (Master Boot Record) <- Boot loader partition table
  - 설치되어 있는 부트로더들의 정보를 가지고 있다.
  - 부트로더에 따른 인터페이스를 제공한다.
  - 제일 상위 플래터, 제일 바깥 트랙, 시작 주소(sector 0번)부터 위치한다.
- 파티션별로 존재하는 block 내용
  - boot block : 해당 파티션에 OS가 깔려있다면, 부트로더가 필요한 데이터들을 저장하는 블록
  - super block : 파티션 전체에 대한 개괄적인 정보 (file system metadata)
    - 어떤 파일 시스템 사용하는지?
    - 데이터 타입, 블록 개수, 용량, 사용하고 있는 용량, 블록 개수 등)
  - bitmaps : 전체 파티션 중 얼마나 사용하고 있는지?
    - 해당 블록이 free 상태인지?
  - i-nodes : file metadata (FCB)
    - 실제 파일이 어떤 것이고, 어느 경로에 있는지?
    - 통상 4KB 사이즈
  - root dir : 시작 주소 (절대 지점)
    - header 역할
  - files & directories : root dir을 활용해서 주소 연산을 통해 파일을 담는다.
    - payload 역할

5. Disk Block
- platter -> cylinder -> track -> sector
- arm-seek : 최대 병목
- 데이터를 쓸 때 arm이 최대한 적게 움직이는 게 성능면에서 좋다.
  - 같은 cylinder에 존재하는 트랙에 데이터를 분산한다. (contiguous)

6. Free-Space Management
- Bit vector, Bit map (n blocks)
  - bit[i] = 1 -> 블록 free 상태
  - bit[i] = 0 -> 사용 중인 블록 (블록의 일부만 사용하더라도)
- Bitmap은 추가적인 저장공간이 필요하다.
  - block size = 2^12 bytes = 4KB
  - disk size = 2^30 bytes (1GB)
  - n = 2^30 / 2^12 = 2^18 bits (32KB) -> 필요한 비트맵 블록 개수
- Contiguous file에 접근하기 쉽다.
- Free space 리스트를 유지하는 방법

    - Linked list (free list)
    - Grouping
    - Counting
    - Space maps (ZFS)
    - 메모리 할당 방식과 비슷하다.

7. File Control Block (FCB)



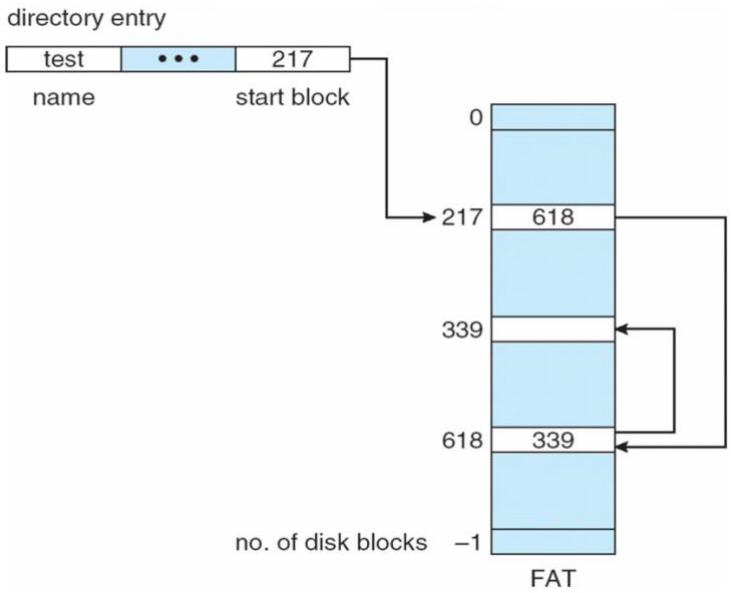
-
- 권한
- 생성, 접근, 수정 날짜
- 만든 사람, 그룹, ACL
- 파일 크기
- 실제 데이터 블록이나, 해당 블록을 가리키는 포인터

8. Directory Implementation
    - 메타데이터가 위치한 곳 (FCB)
    - In the directory entry
        - FCB에 모든 정보 다 담는 방법
        - 파일 독립성이 보장되지 않는다.
        - Direction을 통해 필요하지 않은 정보까지도 access한다.
        - FCB 사이즈가 너무 커져서 사용하지 않는다.
    - In the separate data structure (Unix / Linux)
        - ex. i-node
        - directory entry마다 해당 디렉토리의 정보를 담고있는 구조체를 가리키는 포인터를 저장한다.
        - indirect access
    - Hybrid approach (Windows)
        - 자주 쓰는 정보는 FCB에 저장한다.
        - 불필요한 indirection을 방지한다.

9. Allocation Methods
    - free space : 데이터를 지우는 것은 큰 문제가 되지 않지만, write은 큰 문제가 된다.
    - Contiguous allocation
        - 중간에 업데이트만 없다면 굉장히 좋다. (ex. CD-ROMS)
        - arm seek이 최소화된다. (HDD에서 강점)
        - 디렉토리 엔트리가 간단하다.
        - Extra fragmentation(hole)이 발생한다.
        - Dynamic storage allocation(First fit, best fit)이 필요하다.

- 조각모음 작업이 필요할 수 있다.
- 파일 사이즈는 수정이 계속되면서 점점 커질 수 있다. -> 저장 위치를 다른 위치로 옮겨야될 수도 있다.

- Linked allocation
  - 포인터 개념을 통해 데이터가 흩뿌려져있다.
    - random access가 가능한 매체는 문제가 없다. (SSD, FLASH 메모리)
    - HDD, magnetic tape 등에서는 문제가 된다.
  - 디렉토리 엔트리가 간단하다.
  - External fragmentation이 발생하지 않는다.
  - free block 개수만큼 파일의 용량을 수정할 수 있다.
  - 하나의 블록만 망가져도 그 블록 뒤에 연결된 모든 블록들은 회수(반환)하지도 못하고 가비지 메모리가 된다. (신뢰성 매우 안좋다.)
  - 포인터 변수를 저장해야돼서 추가로 저장공간을 사용한다.
  - arm-seek이 엄청나게 많아져서 성능이 안좋다.
  - Free-Allocation Table (FAT)

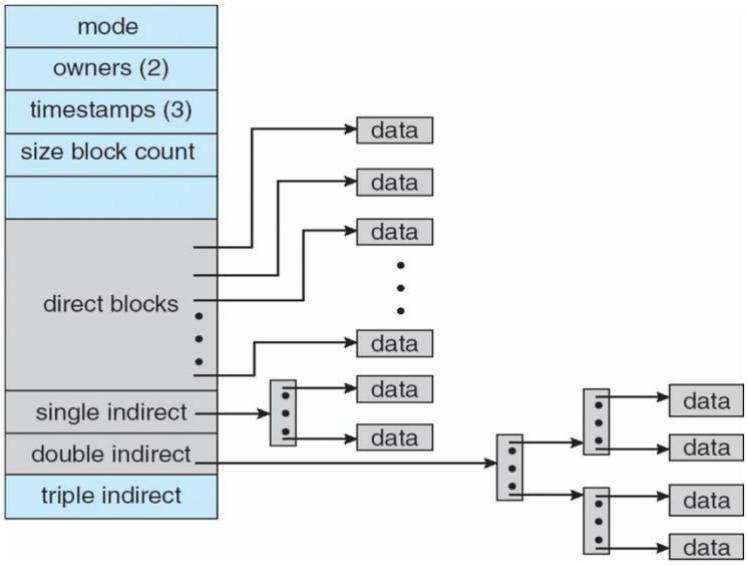


  -
    - 엔트리 테이블의 백업본을 여러 개 만들어놓는다.

- Indexed allocation
  - 각 파일은 인덱스 블록을 가지고 있다.
  - 인덱스 블록은 파일의 데이터 블록들이 어디에 있는지에 대한 인덱스 정보를 가지고 있다.
  - Direct access를 지원한다.
  - External fragmentation(홀)이 발생하지 않는다.
  - i-nodes는 메모리 안에 있어야 한다. (파일이 열릴 때)
  - 인덱스 블록을 추가로 가지고 있기 때문에 저장공간 오버헤드가 발생한다.
    - Linked scheme – 여러 개의 인덱스 블록을 연결
    - Multilevel index blocks
    - **Combined scheme (UNIX UFS)** – 12개의 direct blocks, (single, double, triple) indirect block

C. Indexed Allocation

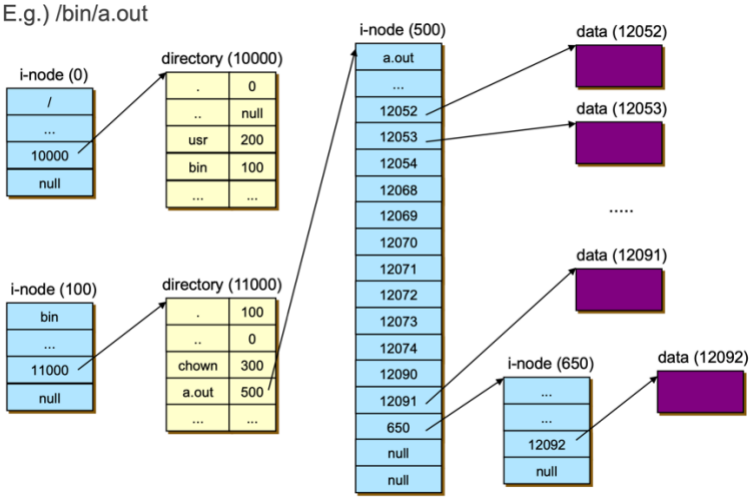
1. Combined scheme : UFS (4KB block size, 32 bits addresses)
2. i-nodes 구조



-
- direct blocks – 12개의 데이터 블록 가리킨다.
  - 블록 하나당 4KB -> 총 48KB까지 접근 가능하다.
- single indirect
  - 중간에 인덱스 블록을 하나 추가한다.(4KB)
  - 한 인덱스 엔트리 = 4B (정수)
  - 1000개의 인덱스 엔트리를 가리킨다.
  - 따라서 1000개의 블록 주소를 나열 가능하다. (4KB * 1000)
  - 4MB 접근 가능
- double direct
  - 같은 원리로 single direct 구조에 인덱스 블록을 한 계층 더 추가한다.
  - 1000 * 1000 * 4KB = 4GB
- triple direct
  - 같은 원리로 인덱스 블록 한 계층 더 추가한다.
  - 1000 * 1000 * 1000 * 4KB = 4TB
  - 상위부터 순차적으로 적용하는데, 아직은 triple direct까지는 필요가 없어서 쓰지 않는다.

3. UFS Structure

- E.g.) /bin/a.out



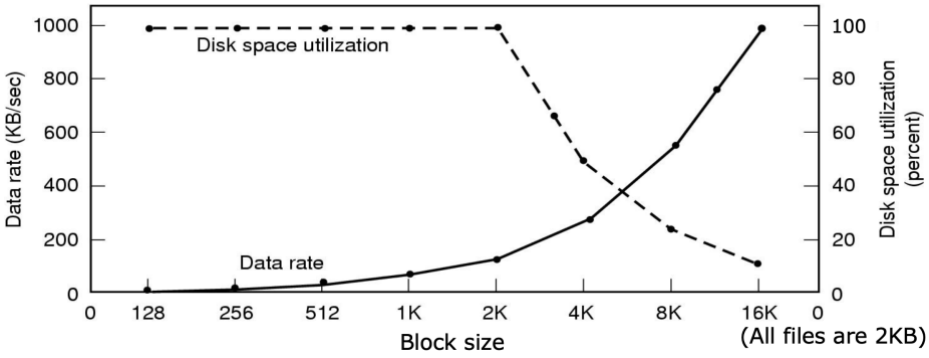
-
- 0번 i-node
  - root directory에 대한 i-node
  - 10,000번째 블록 포인팅

- 10,000번 블록
  - 하위 디렉토리에 대한 인덱스 정보들을 담고 있다.
  - bin -> 100번 블록
- 100번 i-node
  - bin directory에 대한 i-node
  - 11,000번 데이터 블록 포인팅
- 11,000번 블록
  - a.out -> 500번 블록
- 500번 i-node
  - 12개의 direct blocks, single indirect block까지 유효한 상태
  - a.out은 총 13개의 데이터 블록을 사용하는 파일이다.
- UNIX 기반 -> 다수의 유저가 사용할 수 있다.
  - 여러 명이 동시에 a.out에 접근한다면?
  - 같은 경로에 접근이 너무 많아진다.
  - 여러 번 접근하는 경로 및 파일은 버퍼에 캐싱해서 사용한다.
  - 경로만이라도 캐싱되어 있으면 좋다.
  - 동일한 디렉토리 -> spatial locality

D. Block Size에 따른 Performance 및 Efficiency

1. Disk block size vs. file system block size
   - 중간 파일 사이즈는 유닉스에서 대략 1KB다.

2.



   - Disk space utilization : internal fragmentation이 없는지
     - 블록 사이즈가 작을수록 internal fragmentation이 덜 발생한다.
   - Data rate (성능) : 성능이 활용률보다 훨씬 중요하다.
   - 그래서 두 그래프가 만나는 접점보다 작은 블록 사이즈인 4KB를 대부분 사용하는 것이다.

E. Read-Ahead

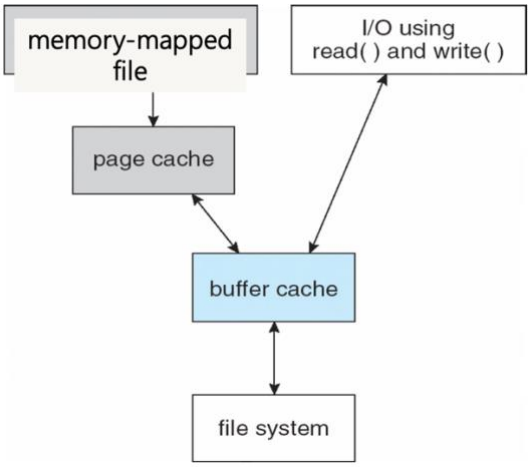
1. 파일 시스템은 프로세스가 다음 프로세스가 다음에 요청할 블록을 예측한다.
   - 프로세스가 한 디렉토리의 한 파일, 혹은 한 파일의 특정 부분만 읽어도 다음 블록을 미리 읽어서 가져온다.
2. 미리 가져온 데이터 블록은 캐시에 저장된다.
3. 연속적으로 접근되는 파일들에 있어서 효과적이다.
4. 파일 시스템은 블록들이 디스크에 흩뿌려지는 것을 방지한다. (할당, 주기적인 재구조화 작업을 통해)

F. Buffer Cache

1. 애플리케이션은 파일을 읽거나 쓸 때 많은 locality를 보인다.
2. 메모리의 캐시 파일 블록들은 버퍼 캐시(혹은 디스크 캐시)에서 locality를 담고있다.
   - 캐시는 시스템 전체(모든 사용자, 모든 프로세스)에서 사용할 수 있다.

- 캐시로부터 데이터를 읽는 것은 디스크가 메모리처럼 사용될 수 있게 한다.
- 4MB 캐시라도 있으면 엄청 효율적이다.

3. 문제점
   - 버퍼 캐시는 Virtual Memory와 경쟁적이다.
   - 버퍼 캐시 역시 제한된 용량을 가지고 있기 때문에 replacement algorithm이 필요하다. (victim 선정)
4. Unified Buffer Cache
   - Demand paging에서의 페이지 캐시와 버퍼 캐시를 통합하여 관리하는 방식



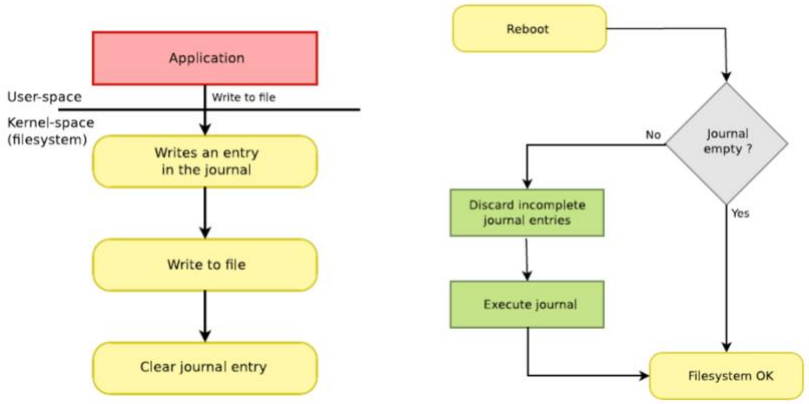
   -
5. Caching Writes
   - 캐시 일관성 관련 이슈 존재
   - Synchronous writes : 너무 느리다. (그때그때 writes 수행)
   - Asynchronous writes (write-behind, write-back)
     - 아직 I/O가 완료되진 않은 블록들의 큐를 유지해야 한다.
     - 주기적으로 큐를 비워야 한다. (디스크에 반영)
     - 신뢰성 문제 : 메타 데이터는 synchronous writes가 필요하다. (작은 파일, 메타데이터에 write하는 경우)
     - 즉, 메모리에 캐싱해놓고, 유휴 상태일 때 디스크에 쓰는 것이다.

G. Reliability
1. File System consistency <- 일관성이 유지됐는지 확인하는 기능을 OS가 제공한다
   - ex. scandisc (Windows), fsck (UNIX)
2. 캐싱된 블락들이 디스크에 쓰여지지 않으면 파일 시스템은 일관성이 무너진 채로 존재한다.
3. 일관성이 깨진 블록 중 i-node, directory, 혹은 free list를 담고 있는 블록이라면 치명적이다.

H. Log Structured File Systems
1. Journaling file systems와 같은 의미
2. 파일 시스템 체크(fsck)는 시간이 많이 소요된다.
   - 시스템이 고장나면 파일 시스템을 천천히 다시 시작한다.
3. 파일 혹은 디렉토리의 변경된 내용을 로그를 통해 기록한다.
4. 고장이 발생하면, 로그(저널)은 부분적으로 완료된 작업(inconsistency를 유발한 작업)을 취소(undo)할 수 있다.
5. 순서 정리
   - read, write 명령이 OS까지 내려온다. (디스크에 실제로 I/O하기 전)
   - 디스크(비휘발성)에 로그를 남긴다. (write-through 방식으로 그때 그때 수행)
   - I/O 작업을 완료하면 해당 로그를 삭제한다.

- 하나라도 로그가 남아있다면 consistency 문제 발생

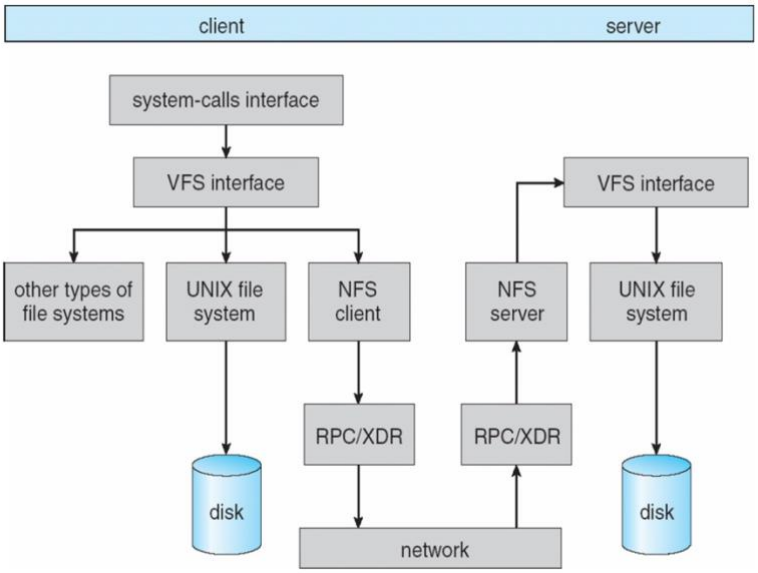
6. 예시
   - IBM JFS for AIX, Linux (UFS를 저널링으로 확장)
   - ext3 for Linux
   - NTFS for Windows
7. 순서도



-

I. Remote File Systems

1. Network File Systems (NFS)
2. 프로토콜을 통해서 파일 시스템을 관리한다.
3. VFS interface -> NFS client -> network -> NFS server -> VFS interface -> UNIX file system -> disk 순으로 I/O 작업이 진행된다.



-