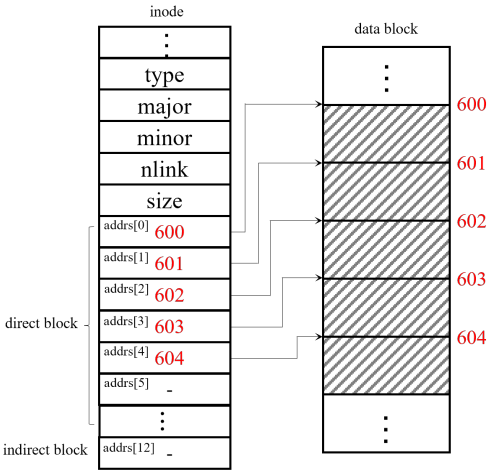
과제 #5 : Continuous Sector(CS) based Multiple Block Supporting File System in xv6

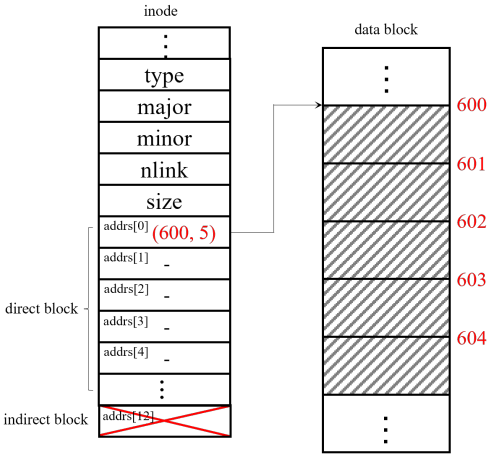
○ 과제 목표

- xv6 기존 파일 시스템의 동작 원리 이해
- xv6에 새로운 파일 시스템 추가

○ 배경 지식



[그림 1] 기존 파일 시스템의 inode 구조체 표현



[그림 2] CS 기반 파일 시스템의 inode 구조체 표현

- [그림 1]은 xv6의 파일 시스템에서 사용되는 inode 구조체의 일부를 나타냄
  - ✓ 빨간색 숫자(600, 601, ...)는 디스크의 데이터 블록 번호임
    - addrs 배열이 가리키는 블록 번호는 디스크에 파일의 내용이 실제로 저장된 데이터 블록의 번호를 의미함
    - direct 블록인 addrs[0]부터 addrs[11]에는 파일의 내용이 저장된 디스크의 데이터 블록을 가리키는 포인터 변수임
    - indirect 블록인 addrs[12]는 디스크 상 해당하는 위치를 직접 가리키는 것이 아니라, 또 다른 인덱스 블록을 가리키고 이를 거쳐서 실제 디스크의 데이터 블록을 가리키도록 하는 포인터 변수임
  - ✓ 결과적으로 xv6는 하나의 파일에서 140개의 데이터 블록을 저장 공간으로 사용할 수 있으며, 이에 따라 한 파일의 최대 크기는 140개 * 512B(Bytes) = 71,680B, 약 70KB임
- 이러한 구조는 디스크 상에 데이터를 연속적으로 할당하더라도 모든 단일 데이터 블록에 대해 매핑을 유지해야 함

○ 과제 기본 내용

- [그림 2]는 본 과제에서 xv6에 새로 추가하는 파일 시스템에서 사용되는 inode 구조체의 일부를 나타냄
  - ✓ 빨간색 숫자(600, 5)는 디스크의 데이터 블록 번호 600부터 5개의 연속된 블록을 표현하며 이것을 CS라고 정의함
  - ✓ 새로운 파일 시스템은 데이터 블록의 관리를 위해 CS를 사용함
    - CS는 (번호, 길이)로 구성된 쌍임
      - ☞ 번호: 실제 데이터가 저장된 디스크의 데이터 블록 번호
      - ☞ 길이: 번호로부터 데이터가 연속으로 할당된 블록 개수
    - (예) 데이터 블록의 크기가 512B이므로 데이터 블록 번호 600부터 파일에 2.5KB 데이터를 한 번에 저장하는 경우 CS는 (600, 5)임
  - ✓ CS 기반 파일 시스템은 기존 파일 시스템보다 연속으로 할당된 데이터 블록에 인덱싱하는 시간을 줄일 수 있고 더 많은 데이터 블록을 사용할 수 있음
- 기존 파일 시스템을 유지하면서 CS 기반 파일 시스템을 통해 새로운 파일을 할당하고 삭제할 수 있도록 구현
  - ☞ 기존 코드와의 호환성을 위해 파일 시스템을 새로운 파일 시스템으로 변경하는 것이 아니라 새로운 파일 시스템을 추가하는 것
  - ☞ CS 기반 파일 시스템을 구현하기 위해서 기존의 inode 구조체를 변경하거나 새로운 inode 구조체를 생성할 필요 없음
- 기존 파일 시스템과 CS 기반 파일 시스템에서 관리하는 파일 정보를 출력하는 함수 구현

○ 과제 구현 내용

1. CS 기반 파일 시스템 구현
   - 기존 파일 시스템에서 관리되는 파일과의 구분을 위해 stat.h에 CS 기반 파일 타입(T_CS) 추가
     - ☞ 추후 코드 구현 시 기존 파일 시스템을 위한 코드와 CS 기반 파일 시스템을 위한 코드를 구분하기 위해 새로 정의한 타입 T_CS을 사용

stat.h

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device
#define T_CS   4   // Continuous Sector based File
```

- CS 기반 파일을 생성하기 위해 fcntl.h에 플래그(O_CS) 추가

☞ open() 함수 호출 시 추가한 플래그를 사용하기 위한 루틴 처리 필요

fcntl.h

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_CS  0x020
```

- CS 기반 파일을 위한 데이터 할당 및 삭제 메커니즘 구현

☞ inode의 direct 블록(4B)을 번호 영역(3B)과 길이 영역(1B)으로 나누어 사용

- 기존 파일 시스템의 direct 블록(4B)에는 디스크의 데이터 블록 번호가 저장됨
- CS 기반 파일 시스템의 direct 블록(4B)에서 상위 3B는 할당되는 데이터 블록의 시작 번호를, 하위 1B는 연속으로 할당되는 데이터 블록의 개수를 저장함

☞ CS 기반 파일을 생성하면 CS를 사용하여 데이터 블록을 관리함

- 파일에 데이터 쓰기 작업을 수행하다가 아래의 경우에 비어있는 다음 direct 블록을 할당받아 쓰기 작업 수행
  - 하나의 direct 블록에서 관리하는 길이 영역의 크기(1B)를 초과한 경우
  - 데이터 블록의 연속적인 할당이 중단된 경우 (예) 다른 파일에 데이터 쓰기 작업 수행
- 아래 '과제 출력 결과 예시' 참고

☞ CS 기반 파일을 삭제하면 데이터 블록의 시작 번호부터 길이만큼 기존에 할당되었던 데이터 블록을 모두 해제

☞ CS 기반 파일 시스템에서 indirect 블록은 사용하지 않음

☞ CS 기반 파일 시스템에서는 파일의 데이터 할당 및 삭제만을 고려함 (디렉토리는 고려하지 않음)

☞ 파일에 내용을 쓰고 저장 후 다시 내용을 수정하는 경우는 고려하지 않음

☞ 할당할 수 있는 데이터 블록이나 direct 블록의 범위를 초과할 경우 범위 내까지만 데이터 할당 후 에러 메시지 출력

2. 파일 정보 출력 함수 구현

- 파일 디스크립터(fd)가 인자로 주어지면 파일의 정보를 출력하는 함수 void printinfo(int fd) 구현

- 함수에서 출력되는 파일의 정보는 파일의 inode 번호, 타입, 크기, direct 블록에 저장된 내용임

✓ 파일 타입이 기존 파일 시스템에서 관리하는 파일인 경우

- 파일 타입은 "FILE"로 출력하고, direct 블록 정보는 사용중인 direct 블록에 저장된 내용만 출력

✓ 파일 타입이 CS 기반 파일 시스템에서 관리하는 파일인 경우

- 파일 타입은 "CS"로 출력하고, direct 블록 정보는 사용중인 direct 블록에 저장된 내용과 (번호, 길이) 정보를 함께 출력

- 아래 '과제 출력 결과 예시' 참고

○ 과제 출력 결과 예시

- 과제 구현 결과를 테스트하기 위한 파일(test.c)이 제공됨

✓ CS 기반 파일 시스템과 파일 시스템 정보 출력 함수를 제대로 구현했을 경우 예상 출력 결과는 [그림 3]~[그림 5]와 같음

✓ [그림 3]의 파일 정보 출력 형식을 준수할 것 (출력 형식 미준수 시 50% 감점)

✓ 제출 시 반드시 test.c 파일 원본을 Makefile에 실행 파일로 포함하여 제출할 것 (test.c 수정본 제출 또는 Makefile에 test.c 파일 미포함 시 채점 불가)

- 테스트 케이스 (1). CS 기반 파일에 연속적으로 데이터를 쓰는 경우

✓ 테스트 프로그램 동작 순서

1. open() 함수를 호출하여 CS 기반 파일을 생성
2. CS 기반 파일에 데이터 130KB(1024B씩 130번) 쓰기 작업 수행
3. 모든 쓰기 작업이 완료되면 printinfo()를 호출하여 CS 기반 파일 정보 출력

✓ 테스트 프로그램 출력 결과

- CS 기반 파일 시스템에서 하나의 direct 블록에서 관리하는 길이 영역의 크기는 1B이므로 최대 저장할 수 있는 블록의 개수는 256임
- 데이터 130KB는 512B * 260이므로 2개의 direct 블록을 사용하는 것을 확인할 수 있음

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ./test
FILE NAME: test_cs
INODE NUM: 20
FILE TYPE: CS
FILE SIZE: 133120 Bytes
DIRECT BLOCK INFO:
[0] 182015 (num: 710, length: 255)
[1] 247045 (num: 965, length: 5)

$
```

[그림 3] 테스트 케이스 (1) 수행 후 출력 결과

- 테스트 케이스 (2). CS 기반 파일에 불연속적으로 데이터를 쓰는 경우 (test.c 파일 39, 40번째 주석 제거)

✓ 테스트 프로그램 동작 순서

1. open() 함수를 호출하여 CS 기반 파일을 생성
2. CS 기반 파일에 데이터 130KB(1024B씩 130번) 쓰기 작업 수행
3. 51번째에 기존 파일 시스템에서 관리하는 일반 파일을 생성하여 데이터 2KB(1024B씩 2번) 쓰기 작업 수행
4. 모든 쓰기 작업이 완료되면 printinfo()를 호출하여 일반 파일 정보 출력
5. 일반 파일의 데이터 쓰기 작업 완료 후 CS 기반 파일에 남은 데이터 쓰기 작업 수행
6. 모든 쓰기 작업이 완료되면 printinfo()를 호출하여 CS 기반 파일 정보 출력

✓ 테스트 프로그램 출력 결과

- CS 기반 파일(test_cs)에 쓰기 작업 수행 중 일반 파일(test_norm)을 생성하여 연속적인 데이터 쓰기를 중단함

☞ 일반 파일에 2KB를 쓰면서 4개(2KB = 512B * 4)의 direct 블록이 사용되는 것을 확인할 수 있음

☞ CS 기반 파일에 1부터 51번째까지 쓰기 작업이 연속적으로 수행되었으므로 길이는 102(51KB = 512B * 102), 일반 파일에 쓰기 작업 수행 후 새로운 direct 블록을 할당받아 52부터 130번째까지 쓰기 작업이 연속적으로 수행되었으므로 길이는 158(79KB = 512B * 158)임

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ./test
FILE NAME: test_norm
INODE NUM: 21
FILE TYPE: FILE
FILE SIZE: 2048 Bytes
DIRECT BLOCK INFO:
[0] 812
[1] 813
[2] 814
[3] 815

FILE NAME: test_cs
INODE NUM: 20
FILE TYPE: CS
FILE SIZE: 133120 Bytes
DIRECT BLOCK INFO:
[0] 181862 (num: 710, length: 102)
[1] 209054 (num: 816, length: 158)
```

```
$
```

[그림 4] 테스트 케이스 (2) 수행 결과 1

- 일반 파일과 CS 기반 파일에 모두 쓰기 작업이 정상적으로 수행되는 것을 확인

☞ ls, wc 명령어 사용하여 파일 크기를 확인했을 때 일반 파일과 CS 기반 파일 모두 쓰기 작업이 정상적으로 수행됨

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
test           2 3 19824
cat            2 4 16432
echo           2 5 15284
forktest       2 6 9596
grep           2 7 18648
init           2 8 15868
kill           2 9 15312
ln             2 10 15168
ls             2 11 17920
mkdir          2 12 15408
rm             2 13 15392
sh             2 14 28032
stressfs       2 15 16300
usertests      2 16 67408
wc             2 17 17164
zombie         2 18 14980
console        3 19 0
test_cs        4 20 133120
test_norm      2 21 2048
$ wc test_cs
0 1 133120 test_cs
$ wc test_norm
0 1 2048 test_norm
$
```

[그림 5] 테스트 케이스 (2) 수행 결과 2

○ 과제 제출 마감

- 2022년 12월 12일 (월) 23시 59분 59초까지 구글 클래스룸으로 제출
- 보고서 (hwp, doc, docx 등으로 작성) 및 소스코드 (Makefile도 반드시 제출)
- 1일 지연 제출마다 30% 감점. 4일 지연 제출 시 0점 처리 (이하 모든 설계 과제 동일하게 적용)

○ 필수 구현

- 1

○ 배점 기준

- 1: 40점, 2: 60점