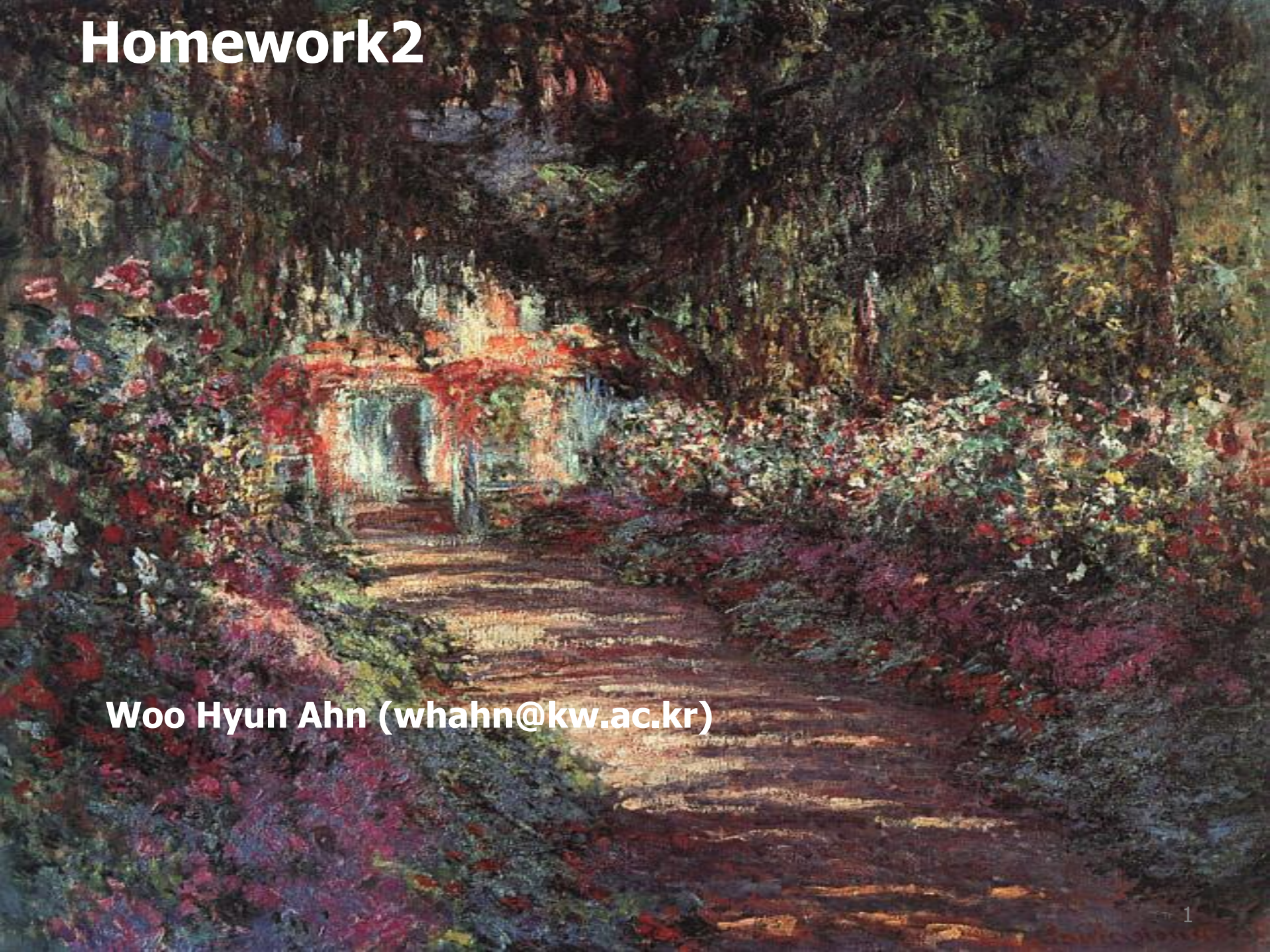
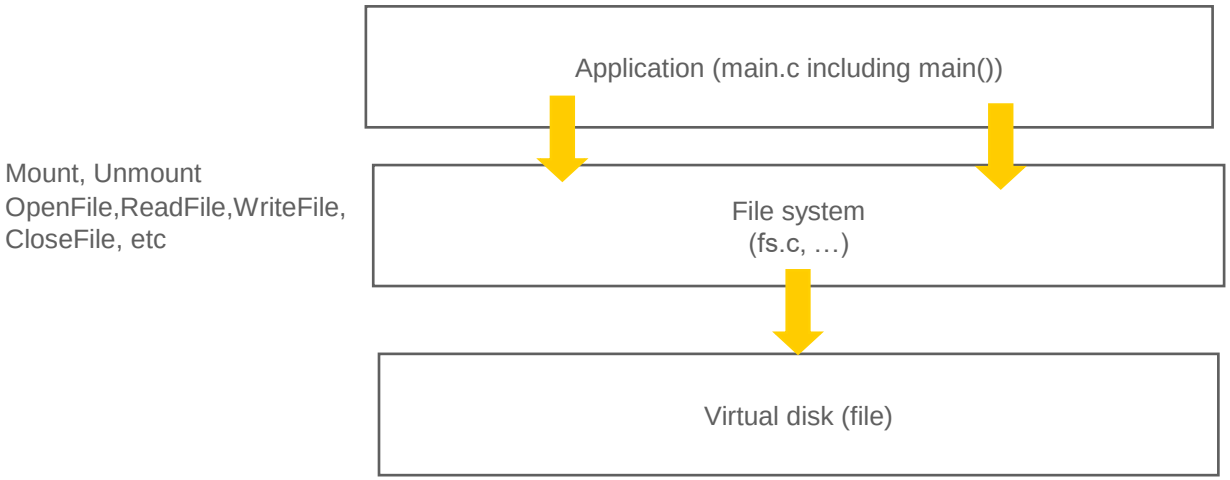
Homework2



Woo Hyun Ahn (whahn@kw.ac.kr)

Layed Architecture



File system layout

File system Info

> File system 전체의 정보를 저장함

> 한 개의 디스크 블록(block 0)에 할당

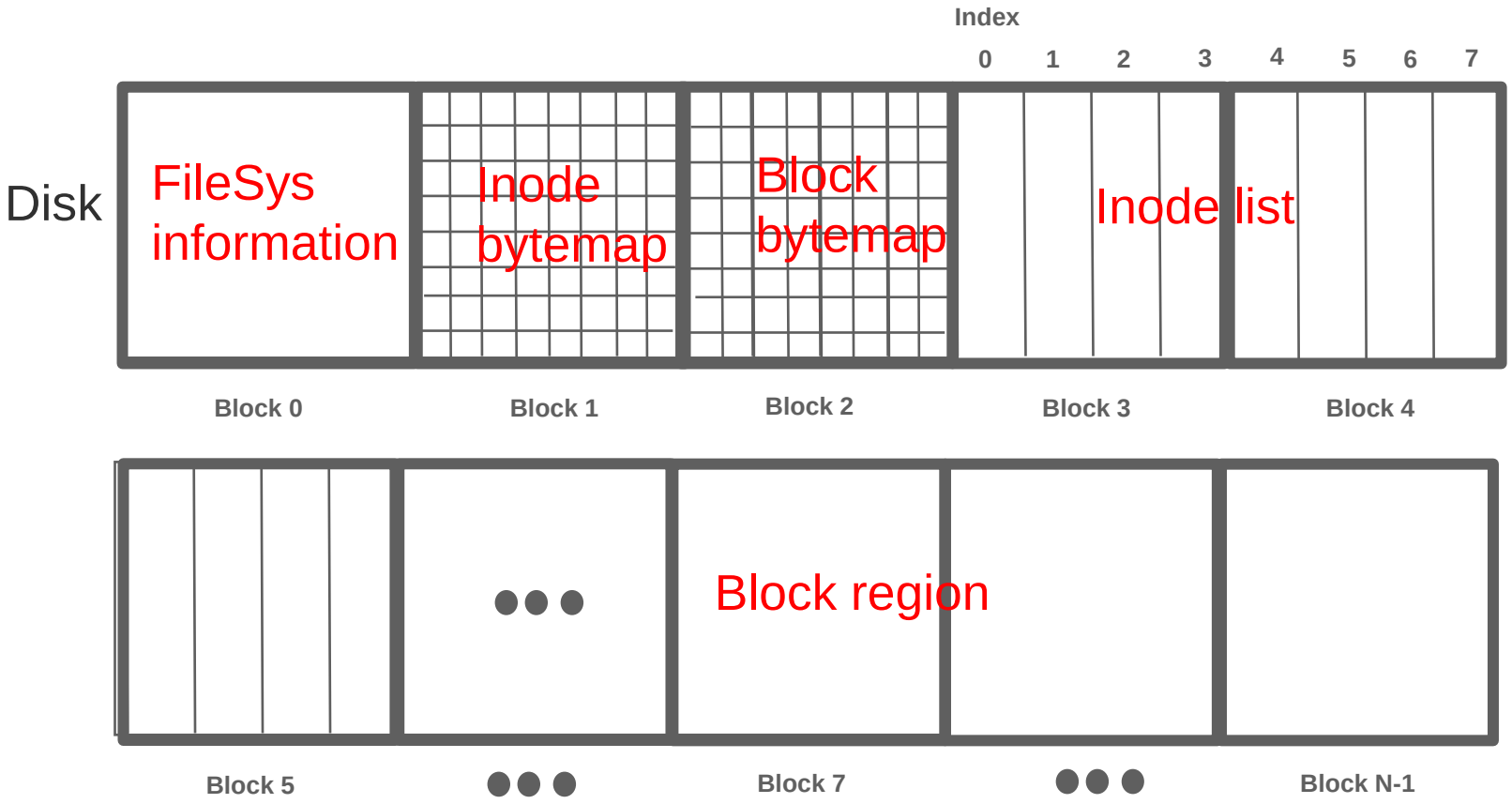
File system metadata management

> 과제1에서 구현한 API 함수를 통해 관리됨

Data Region

> directory block, file block를 저장하는 영역

Basic Disk Layout



- Block 크기: 512 bytes
- Block bytemap: block 1, Inode bytemap: block 2
- Inode list: block 3 ~ 6 (4개 blocks, 전체 Inode 개수: 64개)
  > 과제2에서 Inode list 크기 변경

File System Info

```
typedef struct _FileSysInfo {
    int blocks;                 // 디스크에 저장된 전체 블록 개수
    int rootInodeNum;           // 루트 inode의 번호
    int diskCapacity;           // 디스크 용량 (Byte 단위)
    int numAllocBlocks;         // 파일 또는 디렉토리에 할당된 블록 개수
    int numFreeBlocks;          // 할당되지 않은 블록 개수
    int numAllocInodes;          // 할당된 inode 개수
    int blockMapBlock;          // block bytemap의 시작 블록 번호
    int inodeMapBlock;          // inode bytemap의 시작 블록 번호
    int inodeListBlock;          // inode list의 시작 블록 번호
} FileSysInfo;
```

- 전체 Block 개수 = Block bytemap의 byte 개수
  > Block size = 512 bytes, block 개수 = 512개
- Data region의 블록 개수= 512 – 7개 블록(block0 ~ 6) = 505
- 디스크 용량 = 512 blocks*512 bytes = 2^18 = 256 KB

Directory Entry & File Type

```
typedef enum __FileType {
    FILE_TYPE_FILE,  // regular file
    FILE_TYPE_DIR,   // directory file
    FILE_TYPE_DEV    // device file
} FileType;

#define MAX_NAME_LEN      (28)
Typedef struct  __DirEntry {
     Char name[MAX_NAME_LEN];        // file name
     int inodeNum;
} DirEntry
```

- Block 크기: 512 bytes, directory entry 크기: 32 bytes
- Directory block 당 directory entry 개수: 16개 (512/32)

Inode

```
#define NUM_OF_DIRECT_BLK_PTR   (5)
typedef struct _Inode {
    int             allocBlocks;        // 할당된 블록 개수
    int             size;               //  파일 크기(Byte 단위)
    FileType        type;                 // 파일 타입
    int   dirBlockkPtr[NUM_OF_DIRECT_BLK_PTR];    // Direct block pointers
    int   indirectBlockPtr;
} Inode;
```

- Inode 크기 : 32 bytes
- 블록 당 inode 개수: 512/32 = 16개
- 전체 Inode 개수: Inode list 블록 개수*블록 당 inode 개수
  > 16*4 = 64개

파일 시스템 초기화 동작

- void CreateFileSystem(void)
  - 파일 시스템을 포맷하고 초기화하는 동작을 수행한다.
  - 가상 디스크를 초기화, 즉 생성 후에 동작이 다음 쪽부터 설명될 동작을 하도록 구현한다.

- void OpenFileSystem(void)
  - 파일 시스템을 포맷이 아닌, 전원을 켰을 때 파일시스템 사용에 앞서 이루어지는 동작. 실제 파일시스템에서는 복잡한 동작이 이루어진다.
  - 포맷 대신 가상 디스크를 open하는 동작만 수행한다.

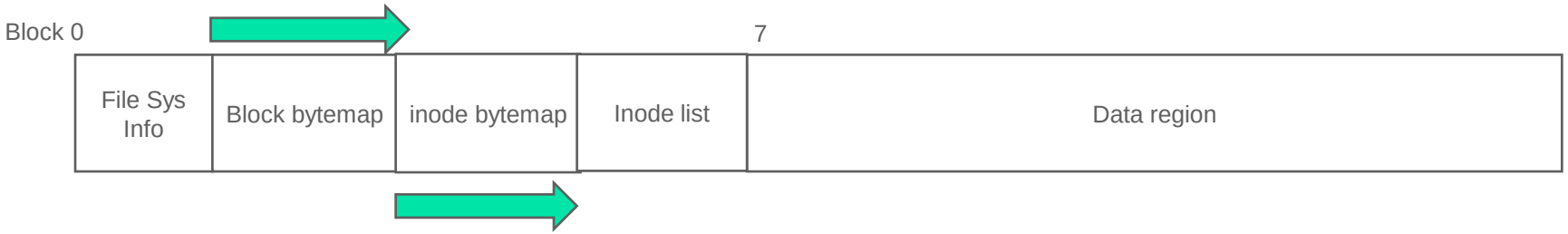
- void CloseFileSystem(void)
  - 전원을 끌 때 호출되는 함수라고 간주하면 된다.
  - 가상 디스크 파일을 close한다.

CreateFileSystem

(0) 파일시스템 초기화 단계기 때문에 **InitFileSys()**을 통해 **Block0**부터 **Block511**까지 0으로 초기화를 해야 한다(디스크 초기화).

디스크 초기화 후에, **FileSysInfo**를 설정하고, 루트 디렉토리를 생성하면 끝!!
> 루트 디렉토리부터 생성하자. 그 후에 **FileSysInfo**를 초기화하자. 원래는 반대다.

(1) **Block bytemap**의 **byte 7** 부터 검색하여 **free block** 검색. **GetFreeBlockNum** 함수를 사용한다. **Block 7**라고 가정하자.
> **Data region**의 시작이 **block 7**이기 때문에 **block 7**부터 빈 공간을 찾도록 **GetFreeBlockNum** 함수를 구현하자



(2) **Inode bytemap**의 **byte 0**부터 검색하여 **free inode** 검색. **GetFreeInodeNum** 함수를 사용한다. **Inode 0**라고 가정하자

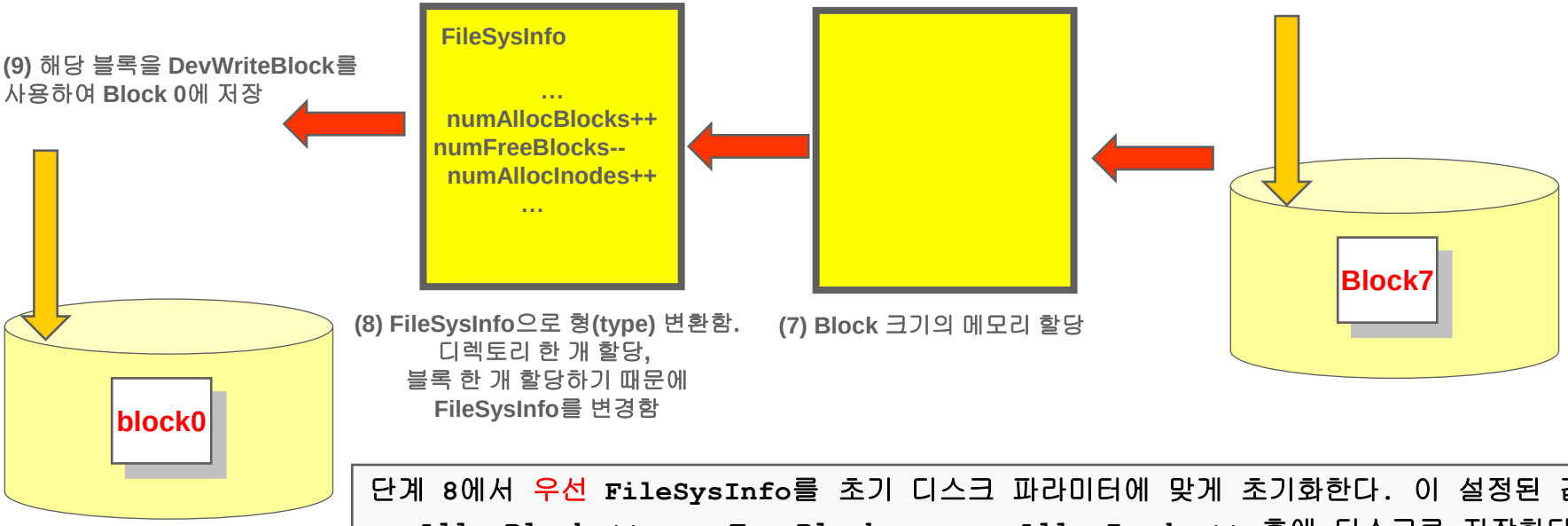
CreateFileSystem

(5) DirEntry[0]의 변수들을 설정함.
> name: “.”
> inode no: 0

(3) Block 크기의 메모리 할당

(4) DirEntry 구조체의 배열로 변환

(6) 변경된 블록을 DevWriteBlock을 사용하여 Block 7에 저장한다



(9) 해당 블록을 DevWriteBlock를 사용하여 Block 0에 저장

(8) FileSysInfo으로 형(type) 변환함.
디렉토리 한 개 할당,
블록 한 개 할당하기 때문에
FileSysInfo를 변경함

(7) Block 크기의 메모리 할당

단계 8에서 우선 `FileSysInfo`를 초기 디스크 파라미터에 맞게 초기화한다. 이 설정된 값에서 `numAllocBlocks++, numFreeBlocks--, numAllocInodes++` 후에 디스크로 저장한다.

CreateFileSystem

Block, Inode가 할당되었기 때문에 Block bytemap, inode bytemap을 변경해야겠죠?
> 그렇지 않으면, 전원 rebooting 후에 변경된 상태가 손실되겠죠

> 여러 분들은 단순히 SetBlockBytemap(int blkno)을 호출하면 Block bytemap의 byte 7이 set 됨.

> 여러 분들이 SetInodeBytemap(int inodeNum)을 호출하면 해당 inode bytemap의 byte 0이 set 됨.

CreateFileSystem

Root 디렉토리 파일에 한 개의 블록이 추가되었으니 해당 inode의 logical block 0에 대응하는 Direct block pointer[0]을 변경해야겠죠?

> Root 디렉토리의 inode를 획득한다.
GetInode(0, pInode)

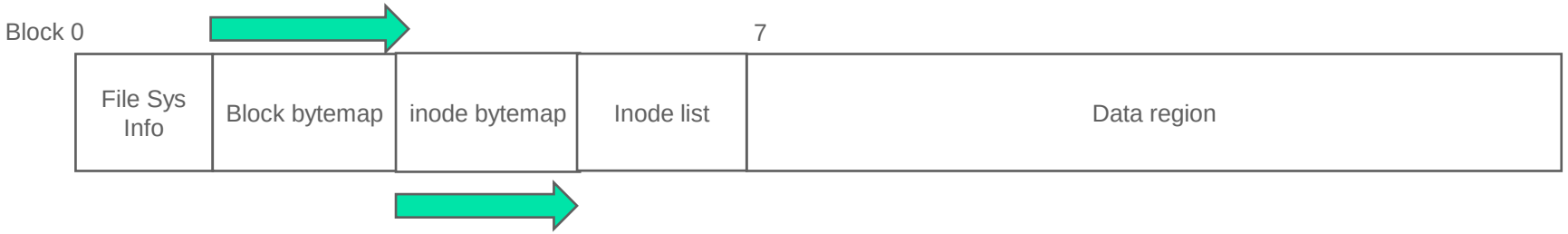
> pInode에서 direct block pointer[0]의 값을 block 7로 설정
: file type, size 등을 설정한다. File type: directory file
: File size: 512, 왜냐하면 디렉토리의 크기는 할당 받은 블록 개수 *512.

> 변경된 pInode의 값을 디스크에 저장함
SetInode(0, pInode)

디렉토리 생성(1)

MakeDir("/tmp")
> Root 디렉토리 블록에 "**tmp**" 디렉토리를 생성한다.

(1) Block bytemap의 **byte 7**부터 검색하여 **free block** 검색. **GetFreeBlockNum** 함수를 사용한다. **Block 8**라고 가정하자

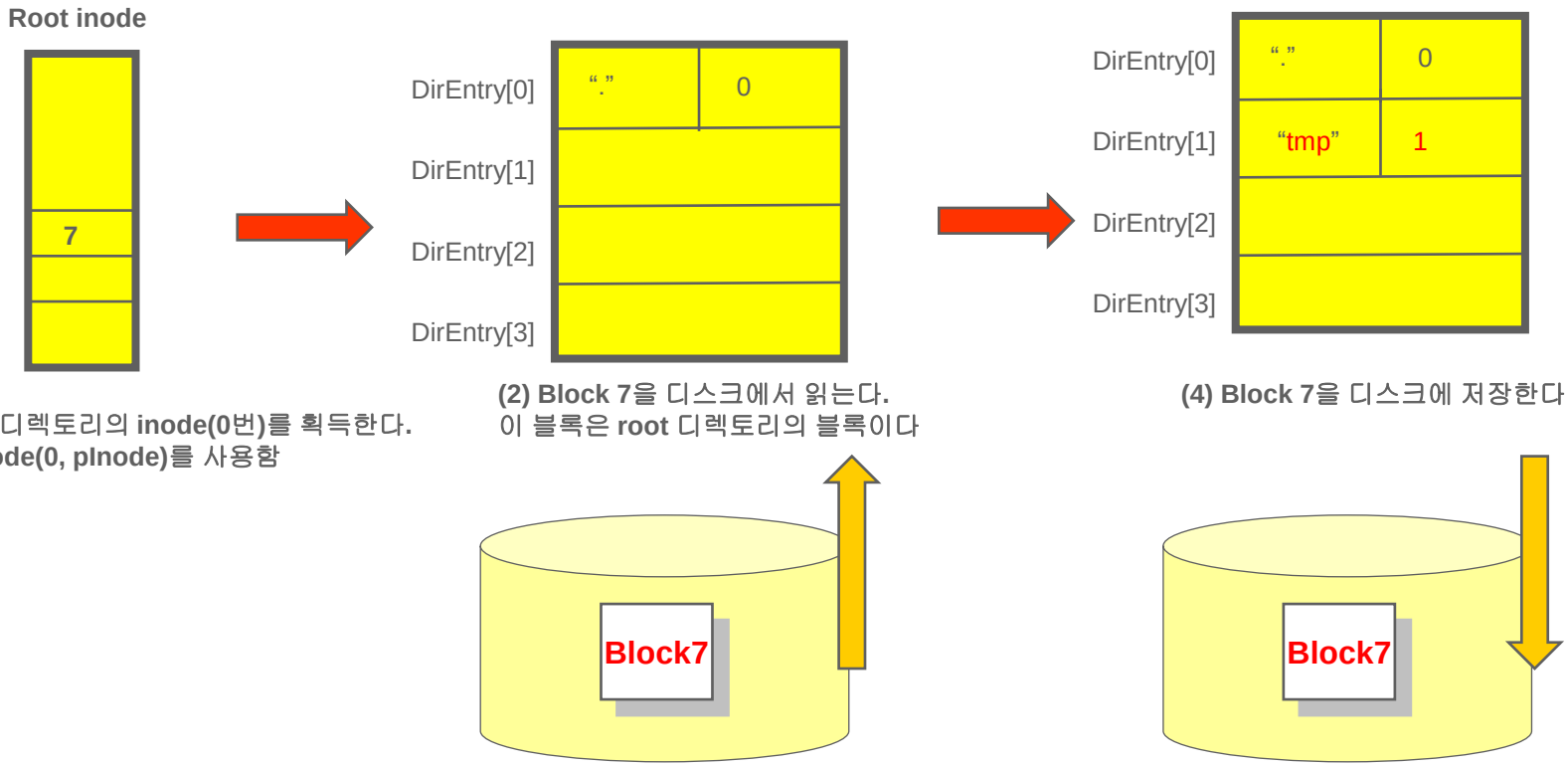


(2) Inode bytemap의 **byte 0**부터 검색하여 **free inode** 검색. **GetFreeInodeNum** 함수를 사용한다. **Inode 1**라고 가정하자

디렉토리 생성(2)

```
MakeDir("/tmp")
   > Root 디렉토리 블록에 "tmp" 디렉토리를 생성한다.
```

(3) DirEntry[1]의 변수들을 설정함.
> name: "tmp"
> inode no: 1



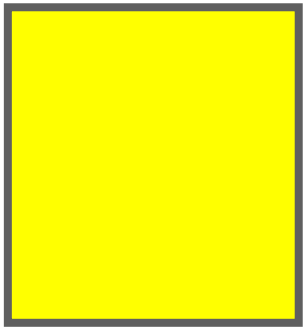
(1) Root 디렉토리의 inode(0번)를 획득한다.
GetInode(0, pInode)를 사용함

(2) Block 7을 디스크에서 읽는다.
이 블록은 root 디렉토리의 블록이다

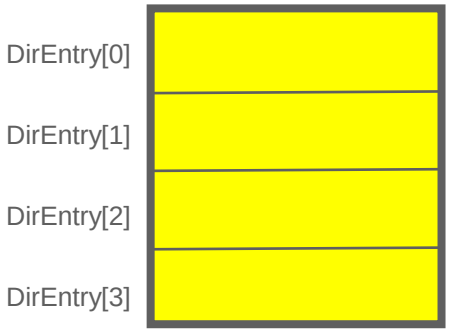
(4) Block 7을 디스크에 저장한다

디렉토리 생성(3)

MakeDir("/tmp")
- **> tmp** 디렉토리의 블록을 생성하고 설정한다
- **> Block 8(단계1에서 할당)**을 디렉토리 블록으로 사용함

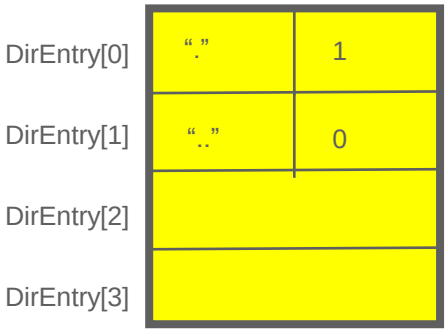


(1) Block 크기의 메모리 할당

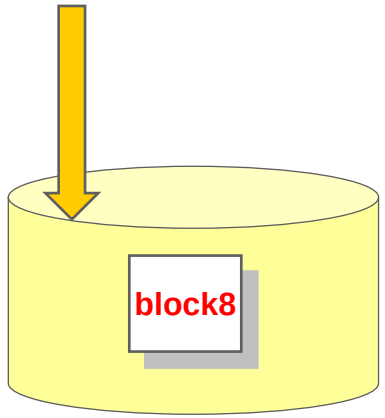


(2) DirEntry 구조체의 배열로 변환

(3) DirEntry[0]의 변수들을 설정함.
- > name: ".", inode no:1
- > name: "..", inode no:0



(4) 변경된 블록을 DevWriteBlock을 사용하여 Block 8에 저장한다



디렉토리 생성(4)

```
MakeDir("/tmp")
   > tmp 파일의 inode를 변경하고 디스크에 저장함
```

> tmp 디렉토리의 inode(단계 2에서 inode 1번임)를 획득한다.
GetInode(1, pInode)

> pInode에서 direct block pointer[0]의 값을 block 8로 설정

> 변경된 pInode의 값을 디스크에 저장함
SetInode(1, pInode)

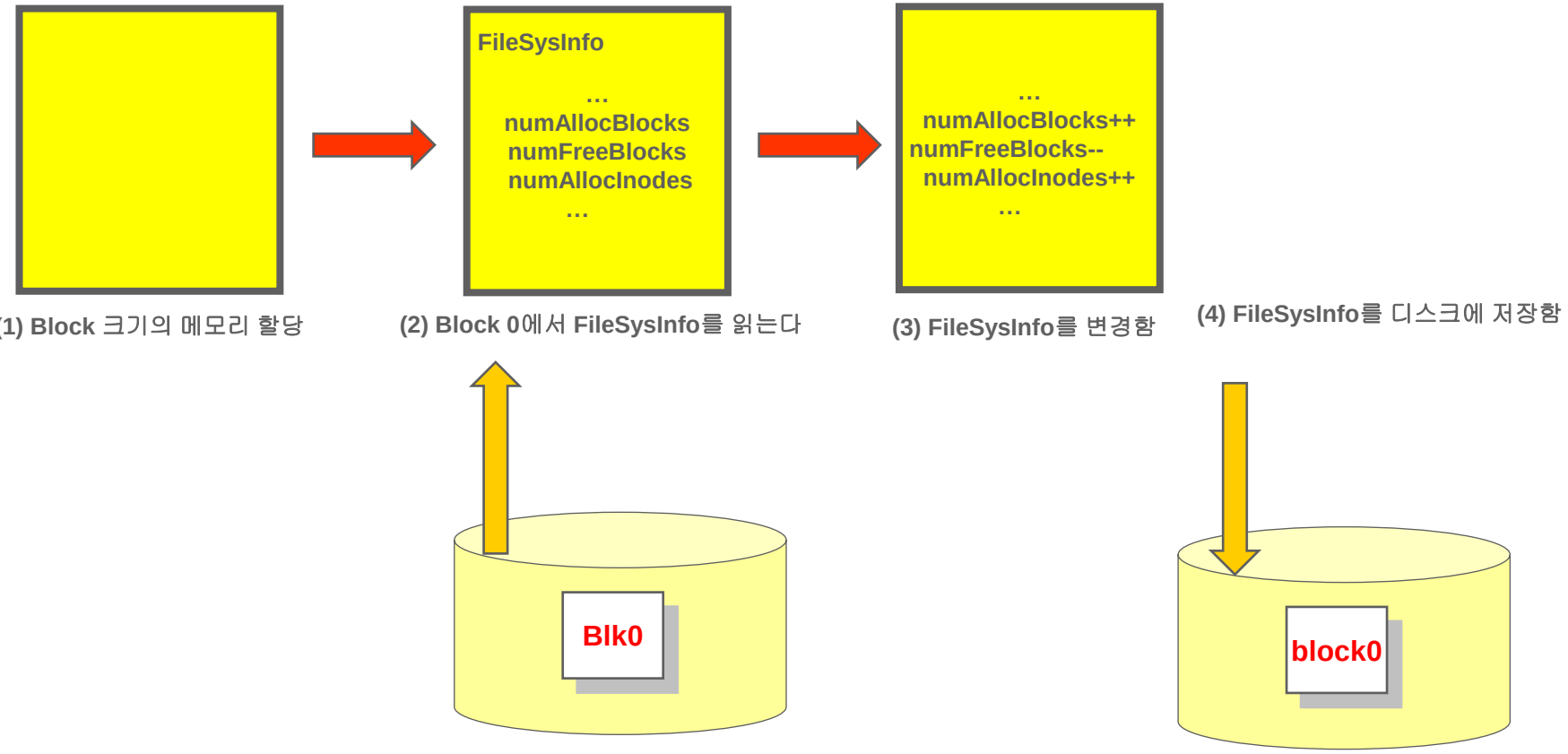
```
MakeDir("/tmp")
   > block bytemap, inode bytemap을 업데이트하자
```

> SetBlockBytemap(8) // block 8을 tmp의 디렉토리 블록으로 할당했기 때문에

- SetInodeBytemap(1) // inode 1을 tmp의 inode로 할당했기 때문에

디렉토리 생성(5)

```
MakeDir("/tmp")
   > FileSysInfo를 디스크에서 읽고 변경한다.
   > 1개 블록 할당, 한 개 파일 할당에 대한 정보 변경
```



(1) Block 크기의 메모리 할당

(2) Block 0에서 FileSysInfo를 읽는다

(3) FileSysInfo를 변경함

(4) FileSysInfo를 디스크에 저장함

파일 생성

int OpenFile(char* szFileName, OpenFlag flag)

> flag가 OPEN_FLAG_CREATE이며, 파일이 존재하지 않으면 생성

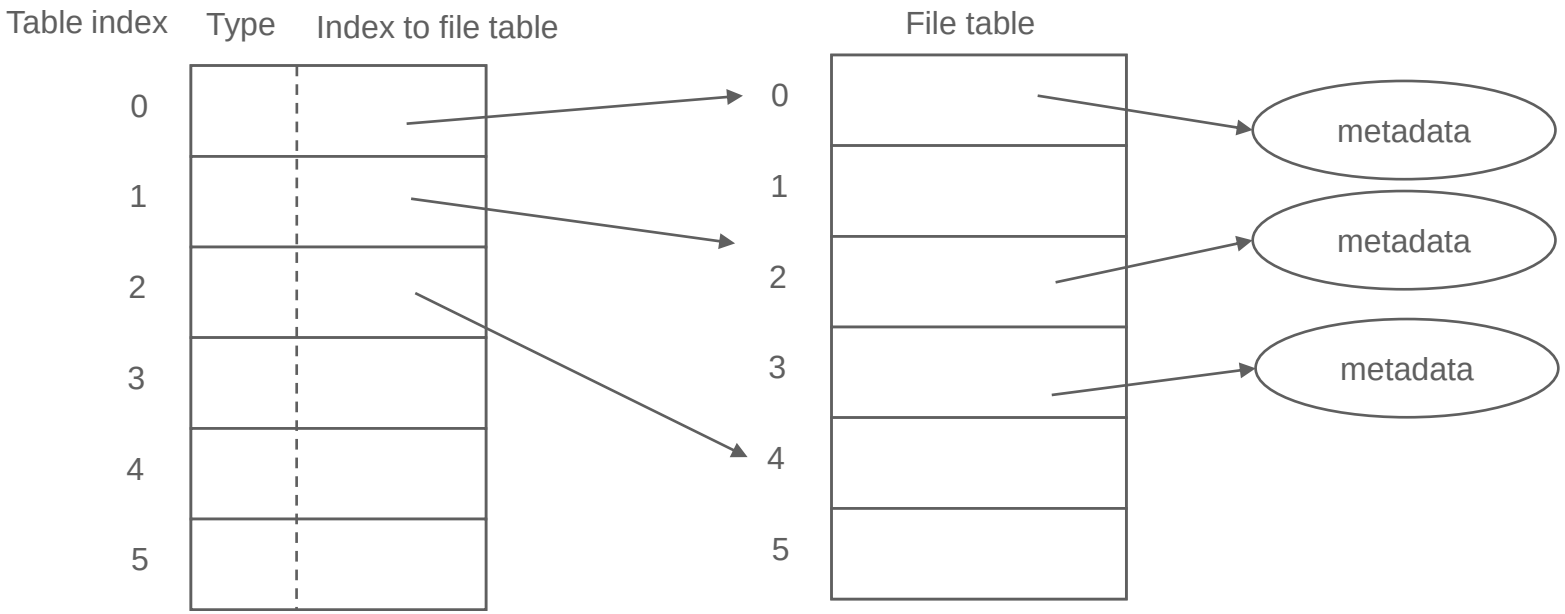
File descriptor table

> open file의 descriptor를 관리하는 table

> file 객체를 포인트함

File table

> 해당 파일의 metadata (e.g., inode 번호, file offset)를 저장한다.



File Descriptor Table and File Object

```
// File descriptor table
typedef struct  __FileDescTable {
    int         numUsedDescEntry;          // 사용 중인 entry의 개수
    DescEntry   pEntry[MAX_DESC_ENTRY];    // MAX_DESC_ENTRY의 Descriptor entry
} FileDescTable


// File descriptor table의 descriptor entry
typedef struct _DescEntry {
    BOOL  bUsed;              // 해당 entry가 사용 중인지를 표시함
    File* pFile;              // open file의 File object pointer
} DescEntry;
```

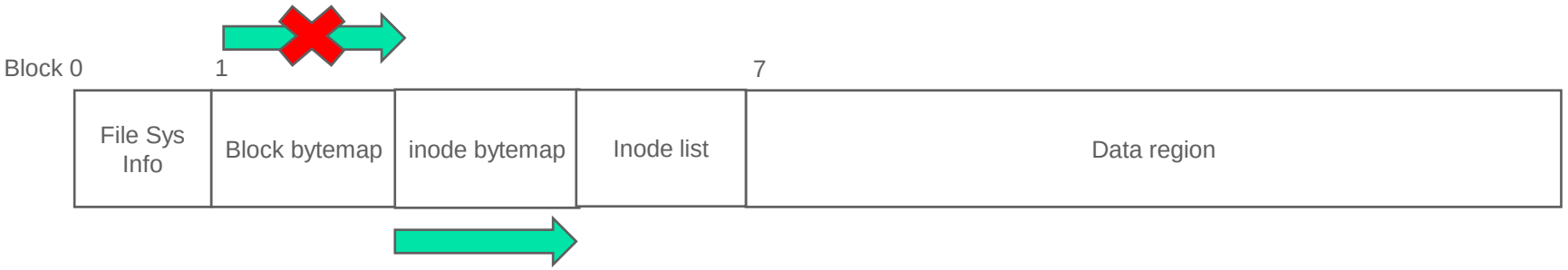
File Descriptor Table and File Object

```
typedef struct __OpenFlag {
    OPEN_FLAG_READONLY,
    OPEN_FLAG_WRITEONLY,
    OPEN_FLAG_READWRITE
} OpenFlag;

typedef struct _File {
    BOOL     bUsed;
    OpenFlag flag;                // open mode
    int      inodeNum;           //  해당 파일의 inode 번호
    int      fileOffset;          // 파일 내에서 최근까지 read/write한 위치(position)
} File;

Typedef struct __FileTable {
    int  numUsedFile;
    File pFile[MAX_FILE_NUM];
}
```

파일 생성(1)

```
OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > a.c를 위한 inode를 할당 받는다.
   > "/" → "tmp" 디렉토리 순으로 tmp의 디렉토리 블록을 찾아야 한다
```

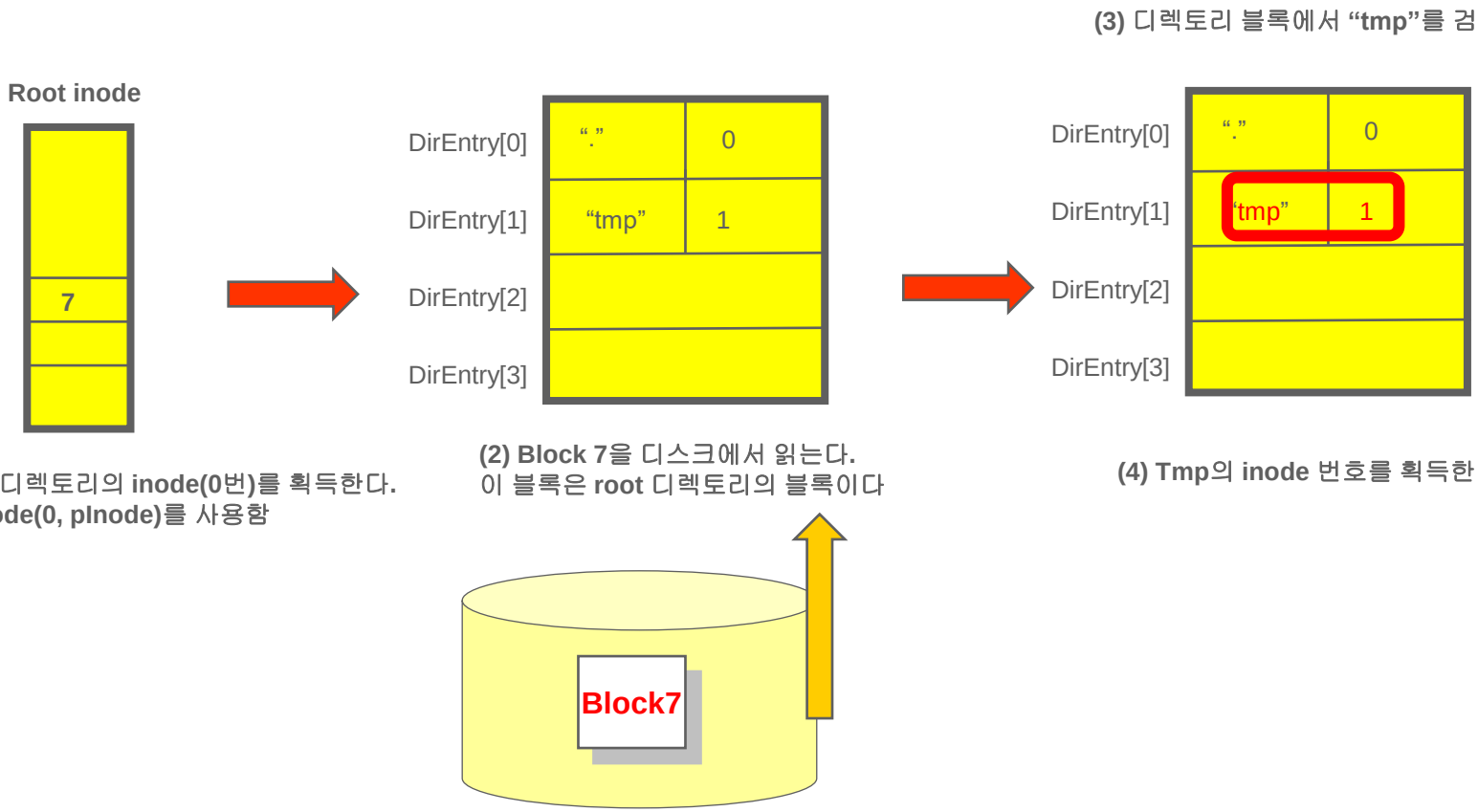
Block bytemap에서 빈 블록을 할당하지 않는다. Directory 생성할 때와의 차이점



Inode bytemap의 byte 0부터 검색하여 free inode 검색. GetFreeInodeNum 함수를 사용한다. Inode 2라고 가정하자

파일 생성(2)

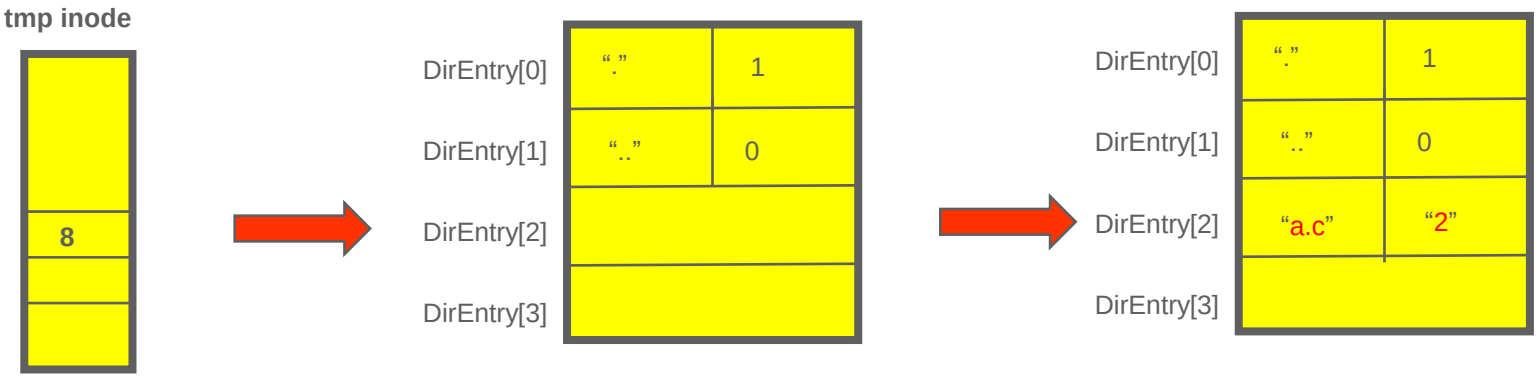
```
OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > "/" → "tmp" 디렉토리 순으로 tmp의 디렉토리 블록을 찾아야 한다
```



파일 생성(3)

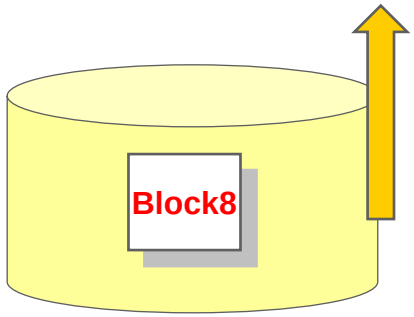
```
OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > tmp의 inode를 디스크에서 읽고, tmp의 디렉토리 블록을 획득함
   > tmp의 디렉토리 블록에 "a.c"를 추가함
```

(3) 디렉토리 블록에 "a.c"와 inode 번호를 추가

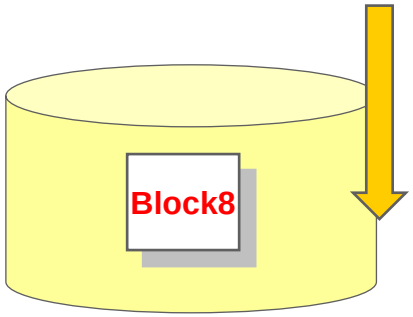


(1) tmp 디렉토리의 inode(1번)를 획득한다.
GetInode(1, pInode)를 사용함.
tmp의 디렉토리 블록은 8.

(2) Block 8을 디스크에서 읽는다.
Block 8은 tmp의 디렉토리 블록이다



(4) Tmp의 디렉토리 블록을 디스크에 저장



파일 생성(4)

```
OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > a.c 파일의 inode를 변경하고 디스크에 저장함
```

> a.c의 inode(inode 2번임)를 획득한다.
GetInode(2, pInode)

> pInode에서 a.c의 특성들을 설정한다. (e.g., file type, size 등등)
: file type: regular file, file size: 0 (데이터가 저장되지 않았기 때문)

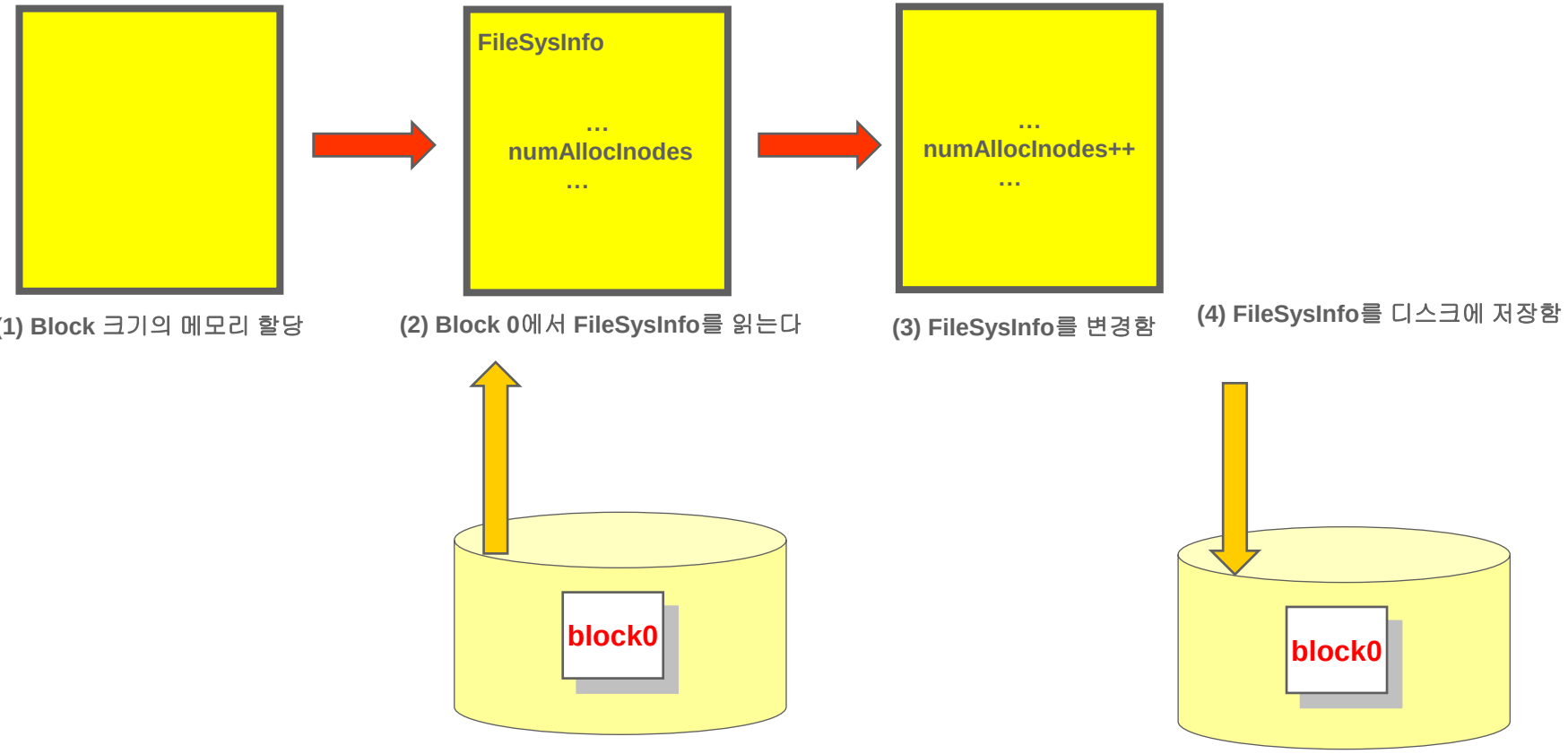
> 변경된 pInode의 값을 디스크에 저장함
SetInode(2, pInode)

```
OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > inode bytemap을 업데이트하자
```

> SetInodeBytemap(2) // inode 2을 a.c의 inode로 할당했기 때문에

파일 생성(5)

```
OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > FileSysInfo를 디스크에서 읽고 변경한다.
   > 한 개 파일 할당에 대한 정보 변경
```



(1) Block 크기의 메모리 할당

(2) Block 0에서 FileSysInfo를 읽는다

(3) FileSysInfo를 변경함

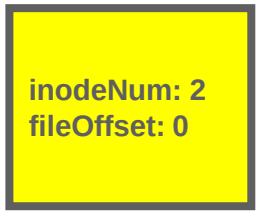
(4) FileSysInfo를 디스크에 저장함

파일 생성(6)

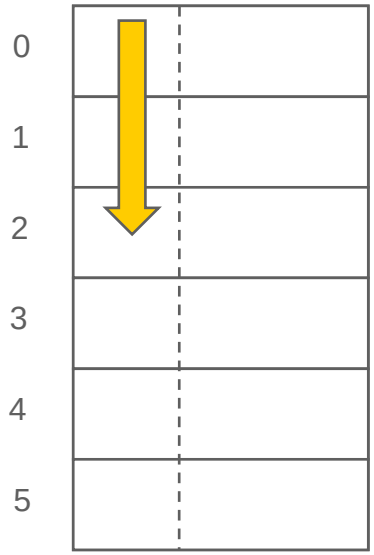
int OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)

> File descriptor table과 file object를 설정하자

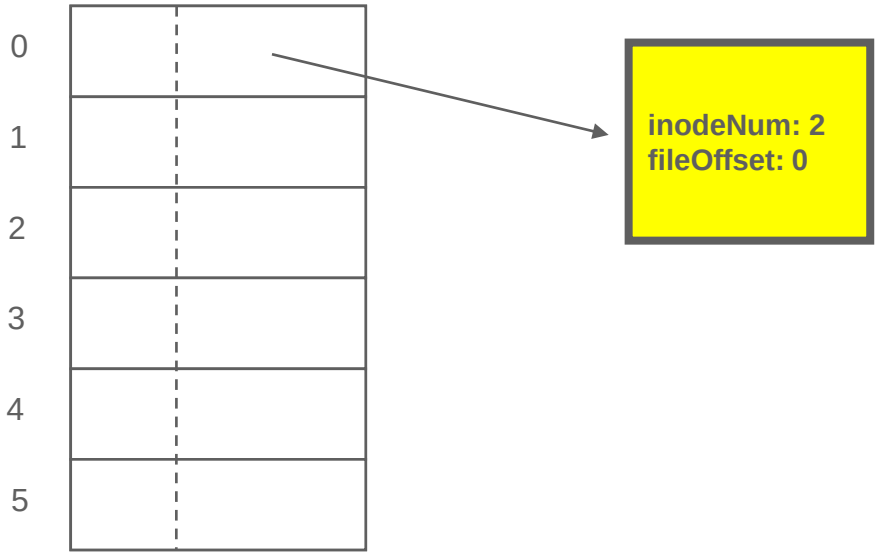
(2) Malloc으로 File 객체를 할당한다.



(3) File 객체의 변수들을 설정한다.
a.c의 inode 번호와 file offset을 설정함



(1) File descriptor table의 index 0부터 시작하여,
빈 entry(bUsed = false)를 찾는다. 찾았다.
Index 0이라 가정하자

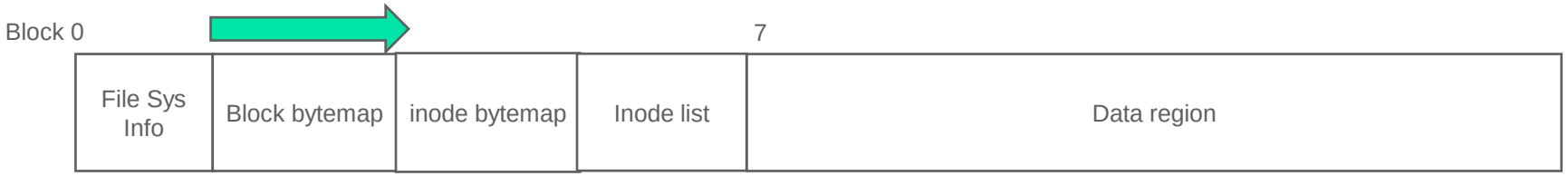


(4) Entry 0의 bUsed를 True로 설정하고,
file object를 포인트 한다. 마지막으로, Index 값을 반환한다

파일 쓰지(1)

```
WriteFile(fd, pBuf, BLOCK_SIZE)
   > Block bytemap에서 빈 블록을 할당 받는다.
```

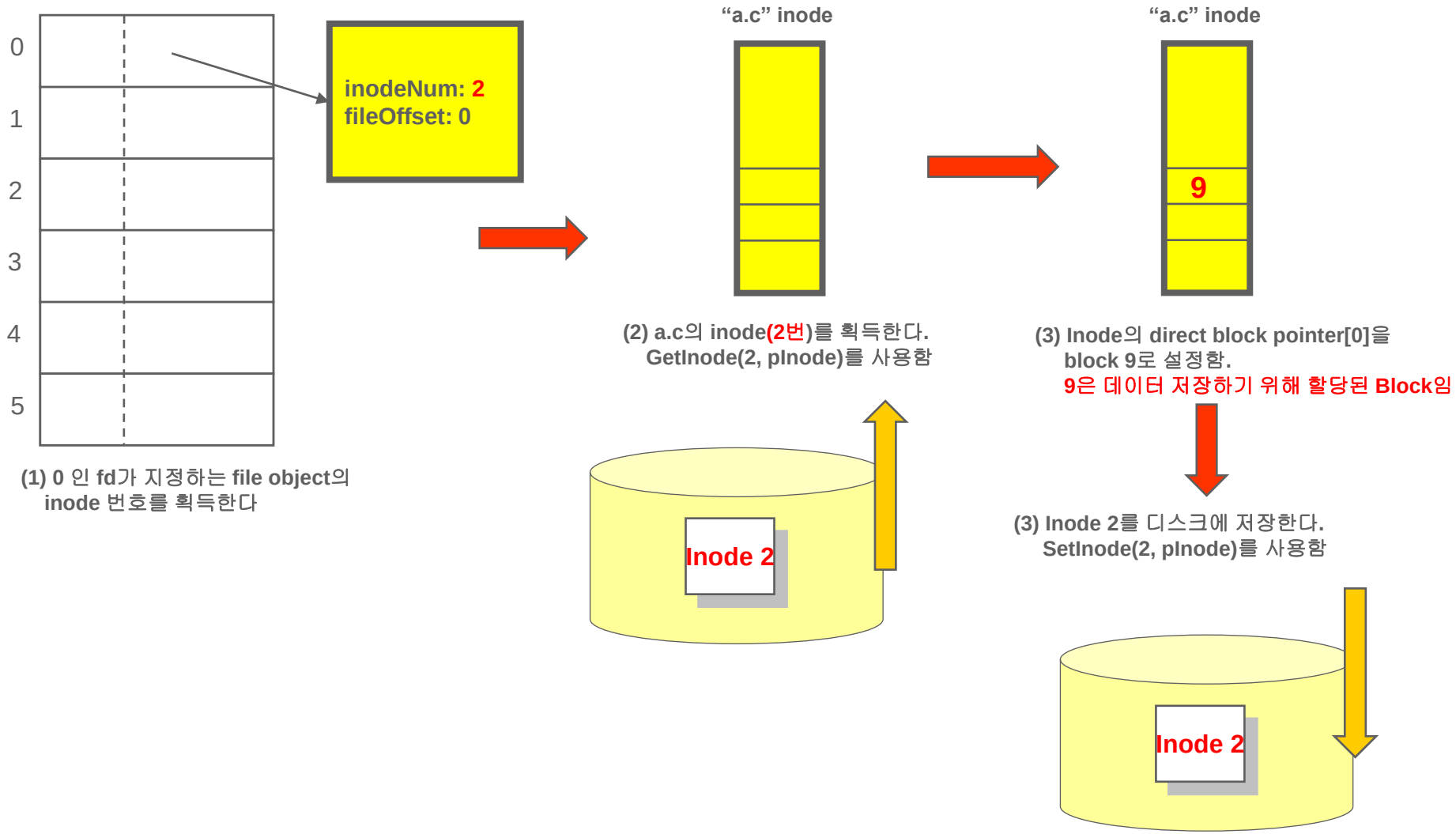
Block bytemap의 byte 7부터 검색하여 free block 검색. GetFreeBlockNum 함수를 사용한다. Block 9라고 가정하자



파일 쓰기(2)

int WriteFile(fd, pBuf, BLOCK_SIZE):

> a.c의 inode를 획득한다.



(1) 0 인 fd가 지정하는 file object의 inode 번호를 획득한다

(2) a.c의 inode(2번)를 획득한다. GetInode(2, pInode)를 사용함

(3) Inode의 direct block pointer[0]을 block 9로 설정함.
9은 데이터 저장하기 위해 할당된 Block임

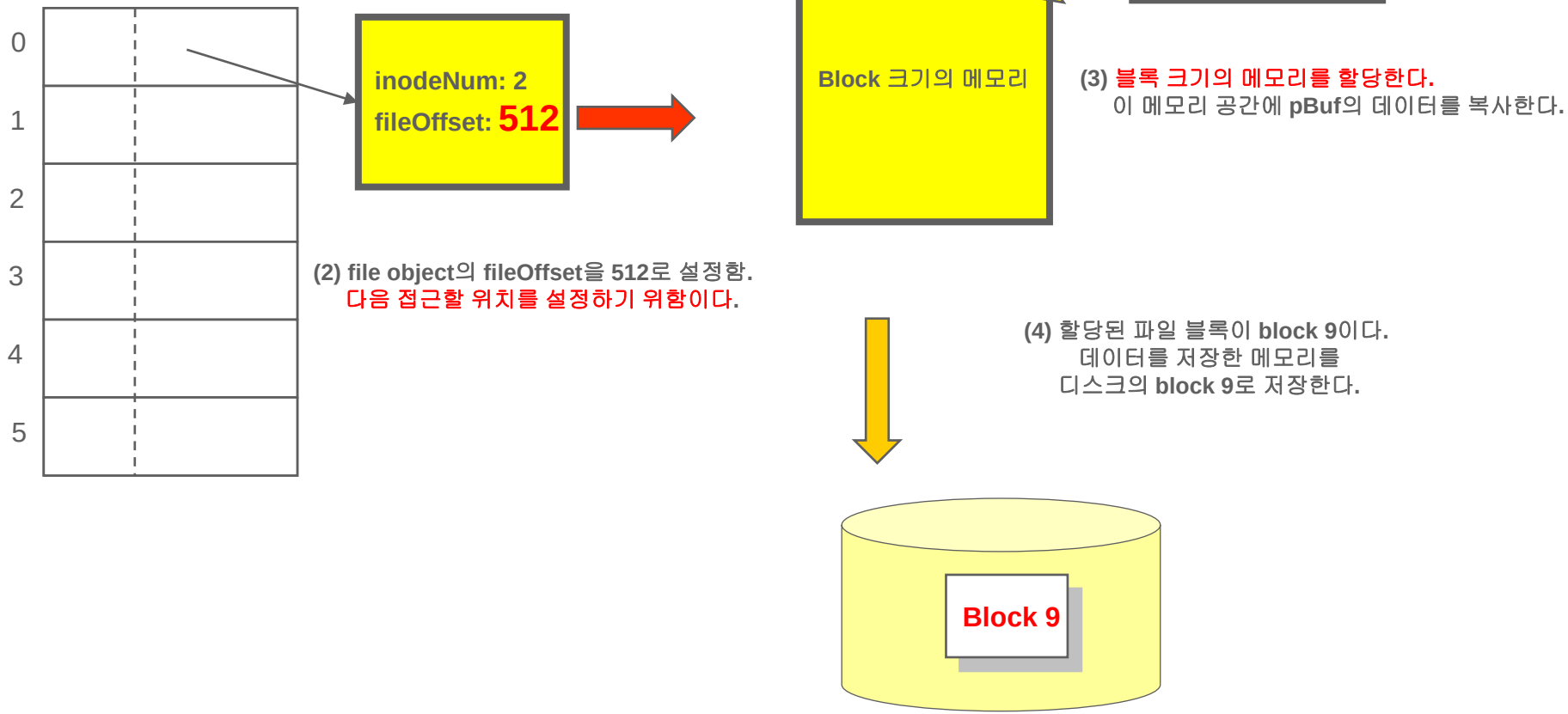
(3) Inode 2를 디스크에 저장한다. SetInode(2, pInode)를 사용함

파일 쓰기(3)

int WriteFile(fd, pBuf, BLOCK_SIZE):

> FileSysInfo를 업데이트하고, 데이터를 저장하자.

(1) Block이 할당되었기 때문에 **FileSysInfo**를 업데이트한다



파일 읽기

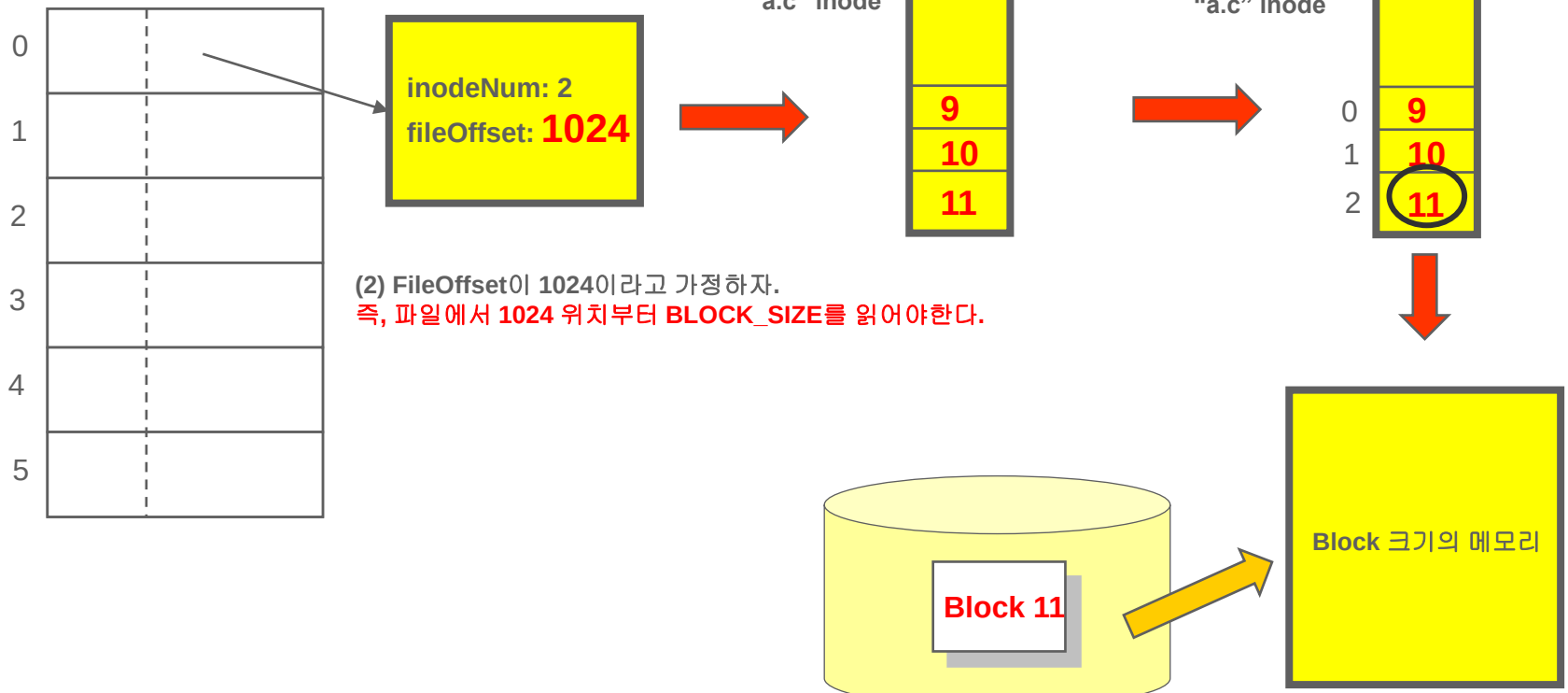
int ReadFile(fd, pBuf, BLOCK_SIZE):

(3) fileOffset에 대한 logical block no를 획득한다.
> 1024에 대한 Logical block no는 2이다

(2) a.c의 inode(2번)를 획득한다.
GetInode(2, pInode)를 사용함

(4) Logical block no에 대응하는 block pointer를 획득한다.

(1) 0인 fd에서 file object를 확인한다.



(2) FileOffset이 1024이라고 가정하자.
즉, 파일에서 1024 위치부터 BLOCK_SIZE를 읽어야한다.

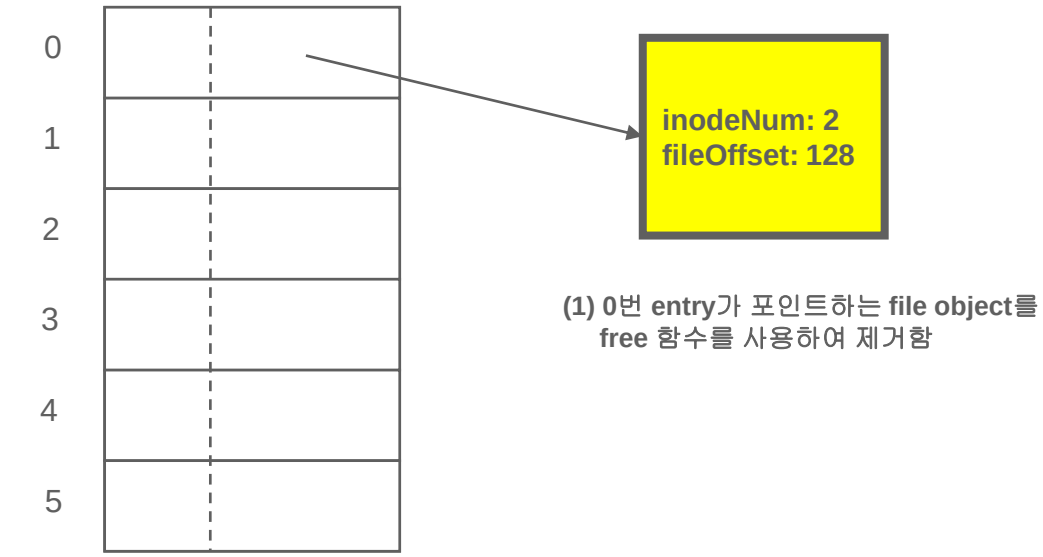
(5) 블록 크기의 메모리를 할당하고,
이 메모리 공간으로 Block 11을 읽는다

(6) 단계 5의 메모리의 내용을 pBuf로 복사해준다.
> 잊지 말자!! file offset은 read/write 크기만큼 증가시킨다.
> ReadFile은 읽은 데이터 크기를 반환한다

파일 닫기

CloseFile(0)

> 0번 descriptor를 닫고자 한다면, file object를 free하고, 0번 entry의 bUsed를 False로 설정



API 구현 범위

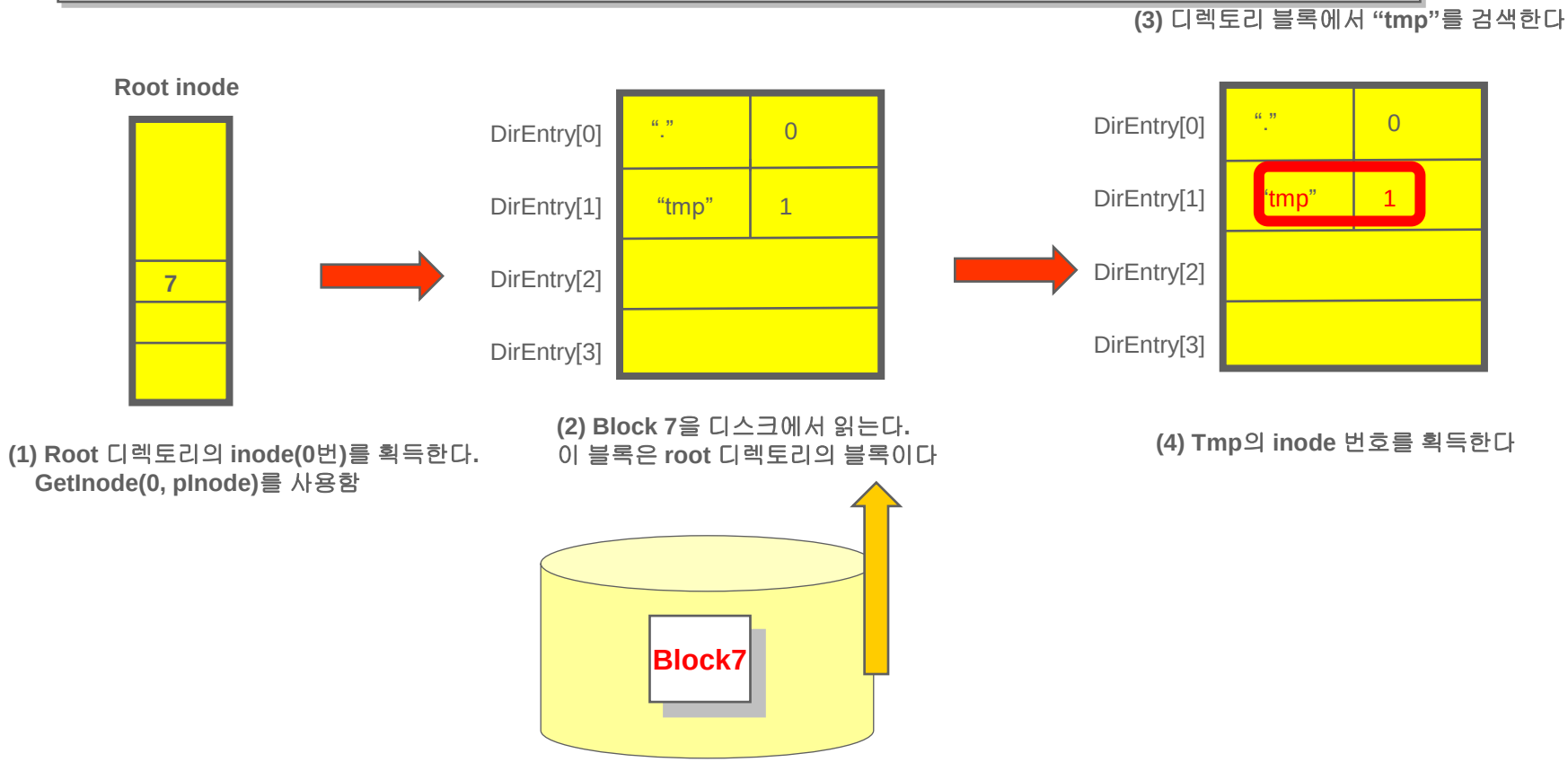
다음 첨부 파일의 API를 구현함.

- OpenFile, MakeDir 함수의 역동작으로 RemoveFile, RemoveDir을 구현할 수 있다. 스스로 구현해보도록 합시다.

Microsoft Word
97 - 2003 문서

파일 열기(1)

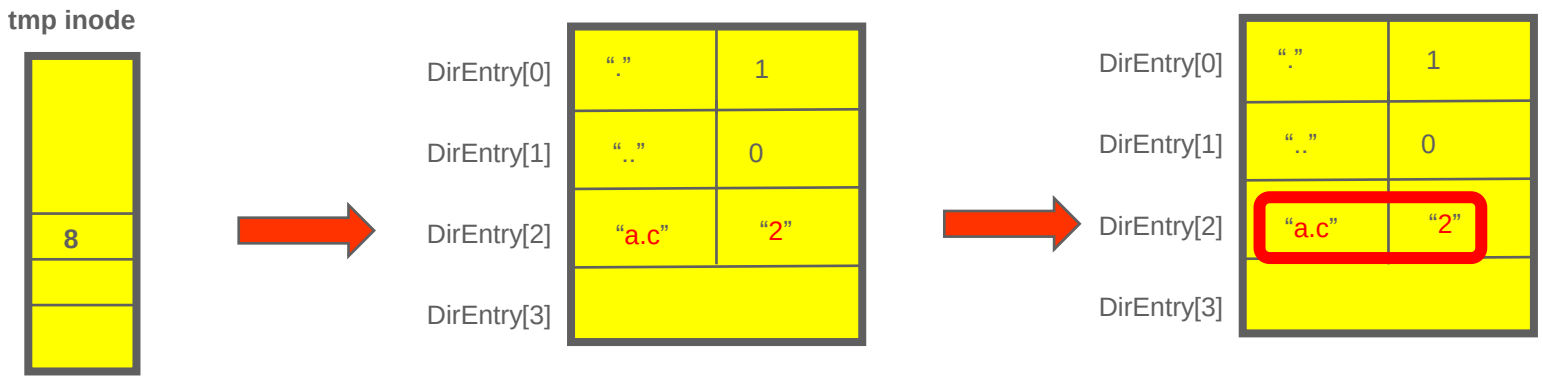
```
OpeneFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > 이미 생성된 a.c에 대한 열기 동작을 수행함. 단, 해당 파일이 존재하지 않으면 -1을 반환
   > "/" → "tmp" 디렉토리 순으로 tmp의 디렉토리 블록을 찾아야 한다
```



파일 열기(2)

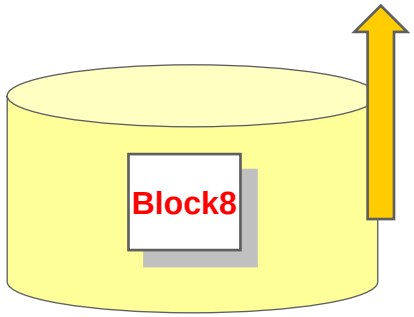
```
OpeneFile("/tmp/a.c", OPEN_FLAG_CREATE)
   > tmp의 inode를 디스크에서 읽고, tmp의 디렉토리 블록을 획득함
   > tmp의 디렉토리 블록에 파일 이름으로 "a.c"의 directory entry 검색
```

(3) 디렉토리 블록에 "a.c"를 검색함



(1) tmp 디렉토리의 inode(1번)를 획득한다.
GetInode(1, pInode)를 사용함

(2) Block 8을 디스크에서 읽는다.
Block 8은 tmp의 디렉토리 블록이다



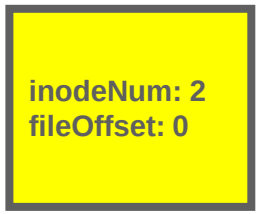
파일 열기(3)

int OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)

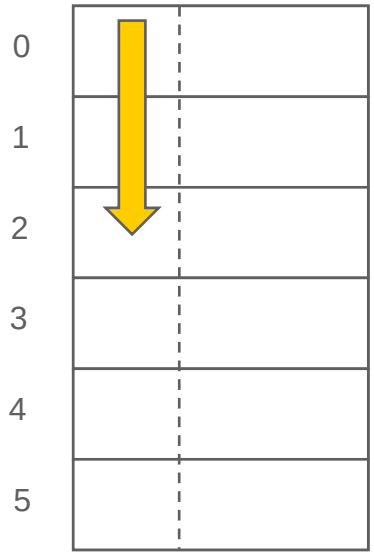
> File descriptor table과 file object를 설정하자

> File descriptor table 동작은 파일 생성의 동작과 동일함.

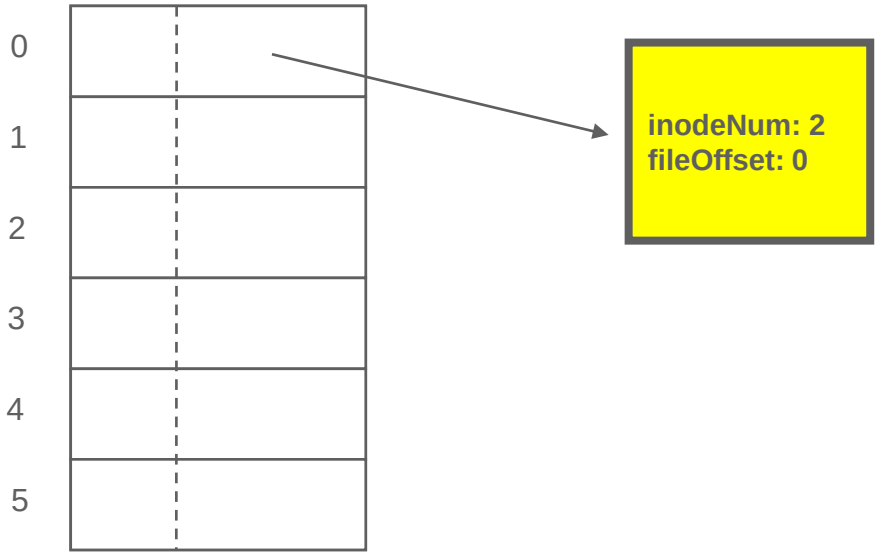
(2) Malloc으로 File 객체를 할당한다.



(3) File 객체의 변수들을 설정한다.
a.c의 inode 번호와 file offset을 설정함



(1) File descriptor table의 index 0부터 시작하여,
빈 entry(bUsed = false)를 찾는다. 찾았다.
Index 0이라 가정하자



(4) Entry 0의 bUsed를 True로 설정하고,
file object를 포인트 한다. 마지막으로, Index 값을 반환한다

디렉토리 열기, 읽기 및 닫기(Linux)

```
Struct dirent* dentry; /* dirent structure */
DIR* dirp; /* directory pointer (descriptor) */

if ((dirp = opendir("/home/park")) == NULL) {
   perror("opendir");
   exit(1);
}
while ((dentry = readdir(dirp)) != NULL)
{
    printf("name:%s, type:%d\n",
        dentry->name, dentry->type);
}
if (closedir(dirp) < 0) {
   perror("closedir");
   exit(2);
}
```

디렉토리 열기와 읽기

Directory* OpenDirectory(char* name)

> 디렉토리를 open하는 함수. 리눅스의 opendir() 동일함

> 성공하면 Directory의 주소를 반환, 실패하면 NULL 반환

> 구현 방법: name의 디렉토리를 검색하고, Directory 구조체를 동적으로 메모리로 할당. 할당된 Directory에 inode를 저장한 후 메모리 주소를 반환함.

```
typedef struct __Directory {
    int       inodeNum;       // 해당 디렉토리의 Inode 번호
} Directory;
```

디렉토리 열기와 읽기

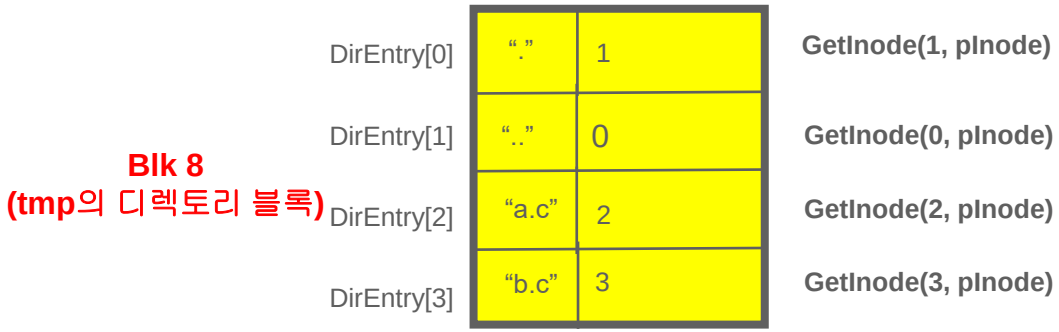
FileInfo* ReadDirectory(Directory* pDir)

> 디렉토리에 저장된 파일의 정보를 획득함. 리눅스의 readdir() 동일함

> 디렉토리에 포함된 File의 정보를 순차적으로 획득함. 성공하면 FileInfo의 주소를 반환. 더 이상 획득한 파일 정보가 없으면, NULL 반환

> 구현 방법: pDir에 저장된 inode 번호를 사용하여 디렉토리 블록을 검색. 해당 디렉토리 블록에 저장된 각 파일의 정보를 FileInfo의 구조체에 저장하여 반환함. 이때, FileInfo 구조체를 동적 메모리로 할당하고, 이 주소를 반환함.

```
typedef struct _FileInfo { // 리눅스에서 direnrty와 유사함
    char*     name[16];    // 파일 이름
    FileType filetype;     // file type
    int inodeNum;          // inode 번호
    int numBlocks;         // file을 구성하는 블록 개수
    int size;              // byte 단위의 파일 크기
} FileInfo;
```



디렉토리 닫기(Linux)

int CloseDirectory(Directory* pDir)

> 열린 디렉토리를 닫는 함수. 리눅스의 closedir() 동일함

> 구현 방법: pDir가 가리키는 메모리를 해제함. 성공하면 0, 실패하면 -1을 반환함. 단, 실패하는 경우는 테스트케이스에 포함하지 않을 계획.

```
FileInfo* pFileInfo;
Directory* pDir;

If ((pDir = OpenDirectory("/home/park")) == NULL) {
   perror("OpenDirectory");
   exit(1);
}
While ((pFileInfo = ReadDirectory(pDir)) != NULL)
{
    printf("name:%s, type:%d\n",
        pFileName->name, pFileInfo->fileType);
}
If (CloseDirectory(dirp) < 0) {
   perror("CloseDirectory");
   exit(2);
}
```

과제 유의 사항

- RemoveDir(char* pathname);
  - 빈 디렉토리만 삭제할 수 있음. 즉, 디렉토리에 한 개의 파일 (또는 디렉토리)가 있다면 삭제를 할 수 없다.
  - 만일 비어 있지 않는 디렉토리를 삭제하고자 할 때는 -1을 반환함.
  - Linux의 rmdir()와 동일한 동작을 수행한다.

- MakeDir("/dir1/dir2/dir3/dir4/…/dirN")
  - 절대 패스의 깊이에 제한을 두지 않음.
  - dirN을 생성할 때 dir4가 없다면 dir4를 생성하지 말고 실패해야 함. 즉, 새로운 디렉토리는 반드시 부모 디렉토리가 존재할 때만 생성될 수 있음.

- 리눅스 system call을 절대 사용하면 안됨
  - open, read, write, close, stat, fopen, fwrite, fwrite 등 파일/디렉토리 관련 시스템 콜을 절대 사용할 수 없음. 사용 시 100% 감점.
  - printf, malloc, free 등 가장 기본적인 함수는 사용 가능함.