Memory Mapped I/O

- Used to map content from kernel space to user space
    1. Automatical synchronization between two duplicates
    2. High effeciency
    3. Can be used for IPC (multiple processes share the same file)
- Manipulate file as array
    - No need to deal with file pointer (fseek, rewind)
    - No read/write (direct access/alteration)

```
void *mmap(void *start, size_t length, int prot,
int flags,int fd, off_t offset)
```

- need to include <sys/mman.h>, munmap() to cancel
    1. start: initial address for mapping region (set as ***NULL***, let kernel choose one)
    2. length: size of the mapped space (how many bytes you want to map from file)
    3. prot: protection type (***PROT_NONE*** or combination of **PROT_EXEC**, **PROT_READ**, **PROT_WRITE**).
    4. flags: Detailed control of mapping, set as **MAP_SHARED**

Example: setup mmap

```
fd = fopen("/dev/mymap","r+");
if(fd < 0){
        printf("open fail\n");
    exit(1);
}
p_map = (unsigned char* )mmap(0,\
                              PAGE_SIZE,\
                              PROT_READ | PROT_WRITE,\
                              MAP_SHARED,fd, 0);
```

- ***Note***: the permission in *fopen()* must be consistent with that in *mmap()*

Space Allocation (1)

All variable has bytes as unit

- Volume of virtual disk (Vol): total number of useable bytes
- Sector size (Bps): the minimal operable unit
- Cluster size (Bpc): size of file block
- Sector per Cluster (Spc): Bpc/Bps
- Sector for header/FAT/root directory/data region

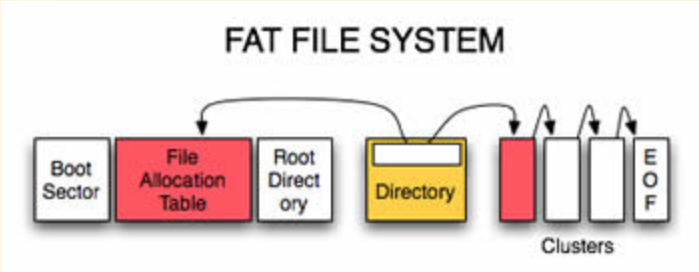
Space Allocation (2)

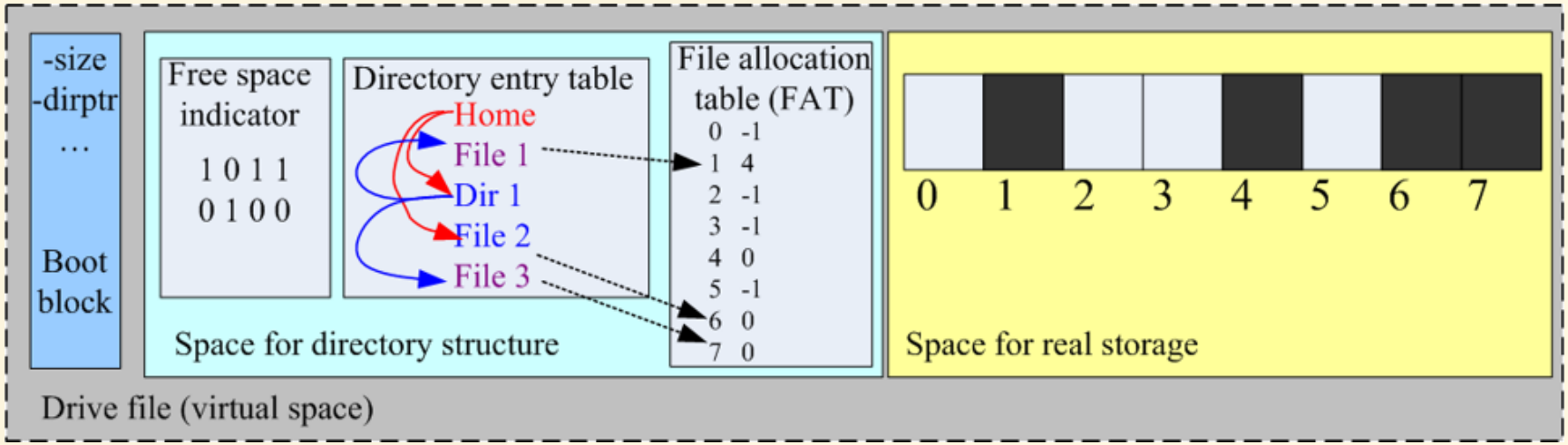
```
struct Volume{
    char name[NAME_LEN]; /*identifer for the virtualdisk*/
    char img[PATH_LEN]; /*disk file's file pointer*/
    int state;
    unsigned int used_size;
    struct VolumeHeader *vbr; /*header*/
    struct FAT *fat; /*pointer for FAT*/
    struct RootDir *root;
    struct Data_blub; /*beginning of the data region*/
};
struct VolumeHeader{
    uint16_t bps; /* 2 bytes ,bytes per sector, default 51
    uint8_t spc; /*1 byte, sector per cluster*/
    uint16_t reserved_sector; /* 2 bytes,for non-file spac
    uint16_t num_fats; /* 12 bytes, number of fat*/
    uint16_t spf; /*2 bytes, sectors used by one fat entry
    uint16_t root_entries; /*2 bytes,number of root entrie
    uint32_t total_sectors; /*4 bytes,num of sectors in vo
};
```

Space Allocation (3)

```
struct FAT{
    unsigned int total_clusters;
    /*total num of cluster in the fat, not the image*/
    unsigned int used_clusters;
    int *map; /*content of FAT table*/
};
struct Entry{
    char name[FILENAME_LENGTH]; /*8 bytes, else long name
    char ext[EXTENSION_LENGTH]; /*extension of file, 3 byt
    uint8_t attribute; /*1 byte, file or dir or system, re
    uint16_t create_time; /*2 bytes, one bytes for Hours,
    uint16_t create_date; /*2 bytes 7 bits for year from 1
    uint16_t last_access; /*2 bytes 7 bits for year from 1
    uint16_t modified_time; /*2 bytes, one bytes for Hours
    uint16_t modified_date;  /*2 bytes 7 bits for year fro
    uint16_t start_cluster; /*start in FAT*/
    uint32_t size;          /*in bytes*/
    unsigned int offset; //current offset
};
```

Space Allocation (4)





- Directory file contains a list of directory entries
- Entries contains the start block (index) in FAT

Operations (1)

- Mount:
  1. Map the disk file to memory -> disk pointer
  2. Indexing the header information by count of bytes (position of header is fixed)
  3. Get position of FAT/Root Dir/Data region from Header information

```
Volume = mmap()
Header = Volume->VolumeHeader;/*pointers are offset in nat
```

- Format: write volume metadata to the header region according to user's input or pre-defined

Operations (2)

- Creation:
  - Locate current directory file -> directory pointer
  - Traverse the FAT to find one available cluster
  - Write the entry into directory
- Write:
  - Get the entry cluster's address & in-cluster offset
  - write data to the region (using memcpy/memncpy)