에디로봇아카데미

임베디드 마스터 Lv2 과정

제 1기

2022. 06. 25

손표훈

CONTENTS

EDDI
Electronic Design
Development Institute

- 파일 디스크립터
  - 파일 디스크립터란?
  - 리눅스 추상화된 파일 시스템의 C 인터페이스
- 파일 API
  - open
  - read
  - write
  - lseek
- 파일 API응용 - 문자열 바꾸기 기능 만들기
  - 구현 전략

파일디스크립터

EDDI
Electronic Design Development Institute

➢ 파일 디스크립터란?

→ UNIX, LINUX 운영체제는 일반적인 정규파일, 물리적 입출력 장치, 소켓 디바이스(네트워크), 디렉토리, 파이프, 블록 디바이스, 캐릭터 디바이스(그래픽 관련) 등등 모든 객체는 “파일 " 로 관리 된다.
→ 프로세서가 이 “파일 " 에 접근 할 때 “파일 디스크립터 " 라는 개념을 사용한다
커널에서 프로세스에서 열린 파일의 목록을 관리하는 테이블(struct files *fd_array[])의 인덱스 값이다
→ 파일 디스크립터는 음수가 아닌 0 ~ 양의 정수 값(0 ~ OPEN_MAX)을 가진다.
→ 프로세서가 실행 중 파일을 open 시스템 콜(SW interrupt)을 호출하게 되면 커널은 해당 프로세스의 파일 디스크립터 중 사용하지 않는 가장 작은 값을 할당 해 준다
→ 프로그램이 메모리에서 프로세스로 시작 할 때 기본적으로 할당되는 파일 디스크립터 값이 있다. 이는 아래 표와 같다
아래 기본적인 파일 디스크립터는 프로세스가 시작하면 항상 열린채로 동작한다

파일 디스크립터	목적	POSIX 이름	stdio 스트림	비고
0	표준 입력	STDIN_FILENO	stdin	키보드, 마우스, 하드디스크..
1	표준 출력	STDOUT_FILENO	stdout	모니터, 하드디스크..
2	표준 에러	STDERR_FILENO	stderr	

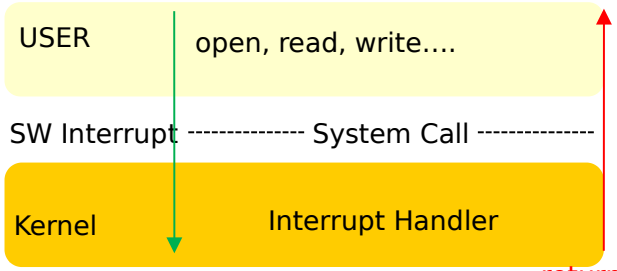
→ 각 프로세스 별로 커널은 open file table을 가지고 있다

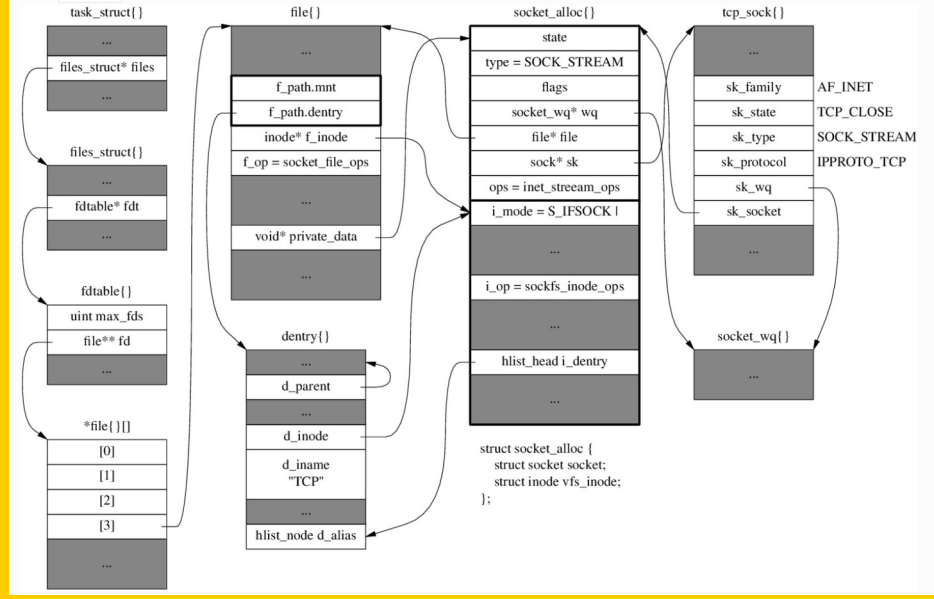
파일디스크립터

EDDI
Electronic Design Development Institute

➢ 파일 디스크립터란?

→ 시스템 콜이 호출되면 인터럽트 핸들러 실행
→ 커널에서 우측의 그림과 같이 연결리스트를 생성한다
(우측은 socket file의 예시)





출처 : http://chenshuo.com/notes/kernel/data-structures/

리눅스 추상화된 파일 시스템의 C 인터페이스

EDDI
Electronic Design Development Institute

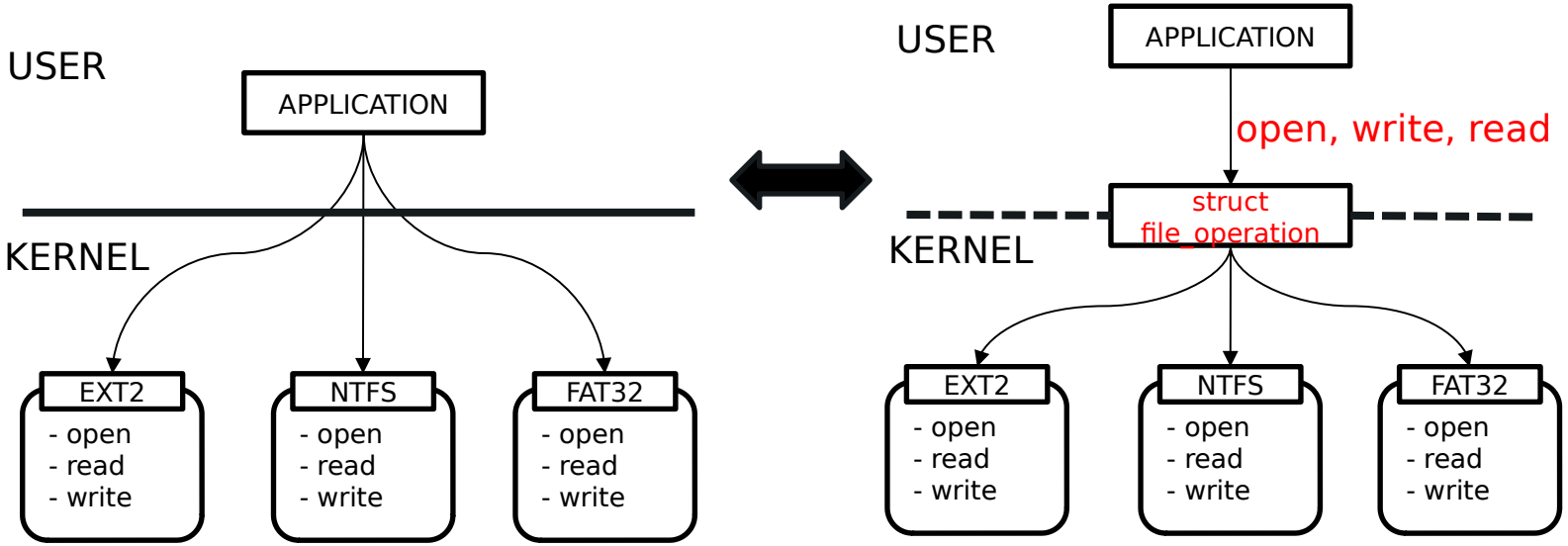
- file struct 내부의 file operation
  - → file struct 내부에 file operatio이라는 구조체에 파일을 관리하는 API 함수 포인터를 볼 수 있다
  - → 이 인터페이스 구조는 리눅스에서 지원하는 여러 종류의 파일 시스템을 관리 할 수 있게 한다
  - → 이 와 같은 구조를 VFS : Virtual File System이라 한다

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached value */
        const struct file_operations    *f_op;

        /*
         * Protects f_ep, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
```

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, struct io_comp_batch *,
            unsigned int flags);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
            loff_t, size_t, unsigned int);
    loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                   struct file *file_out, loff_t pos_out,
                   loff_t len, unsigned int remap_flags);
    int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

리눅스 추상화된 파일 시스템의 C 인터페이스

- file struct 내부의 file operation
  - → VFS와 같이 인터페이스 구조를 사용할 때와 안 할 때의 차이를 보자



- → 인터페이스가 없을 때 사용자가 각 파일 시스템에 맞게 함수를 호출해줘야 한다
- → 인터페이스가 있으면 일관된 함수 호출을 통하여 각 파일 시스템에 접근 할 수 있다

파일 API

EDDI
Electronic Design Development Institute

- open
  - → 파일을 생성, 수정하거나 파일에 데이터를 읽고 쓸 때 사용
  - → 함수의 원형은 아래와 같다

```
int open(const char *pathname, int flags, mode_t mode)
```

  - → 첫 번째 파라메타인 pathname은 파일이 위치한 경로
  - → 두 번째 flags는 파일을 어떤 방식으로 열것인지 정한다

Flags	**설명**
O_RDONLY	읽기 전용
O_WRONLY	쓰기 전용
O_RDWR	읽기/쓰기 혼용
O_CREAT	생성
O_APPEND	이어쓰기 모드로 쓰기 작업 시 파일의 끝에 쓰기 수행
O_TRUNC	잘라내기
O_NONBLOCKING	논블로킹 작업
O_SYNC	쓰기 작업시 바로 물리적 메모리에 위치한 파일에 쓰여짐
O_EXCL	파일이 존재하는지 확인(존재하면 -1)

파일 API

EDDI
Electronic Design Development Institute

- open
  - → 파일을 생성, 수정하거나 파일에 데이터를 읽고 쓸 때 사용
  - → 함수의 원형은 아래와 같다

int open(const char *pathname, int flags, mode_t mode)

  - → 마지막으로 mode는 파일에 대한 권한을 설정해준다
  - → 이 권한은 3자리씩 8진수로 표현한다
  - → 리눅스 파일 정보를 보면 아래와 같은 정보를 알 수 있다

파일형식	권한정보	링크수	소유자	소유그룹	용량	생성날짜	파일이름
d	rwxr-xr-x	3	root	root	4096	June 11	test.txt

☒ 파일형식 : “d”-디렉토리, “l”-링크파일, “-”-일반파일

☒ 권한 정보는 아래와 같다

예를 들어 권한이 rwxr-xr-x라면 (rwx)(r-x)(r-x)로 r = 4, w = 2, x = 1이므로 이를 숫자로 변환하면 다음과 같다
(rwx) = 4+2+1 = 7, (r-x) = 4+0+1 = 5이므로 따라서 해당 파일의 open mode는 755가 된다

```
합계 40
drwxrwxr-x 2 son son  4096  7월  1 11:45 .
drwxrwxr-x 3 son son  4096  6월 27 20:01 ..
-rw-rw-r-- 1 son son    72  6월 30 22:35 file_test.txt
-rwxrwxr-x 1 son son 17224  6월 30 20:18 str_swap
-rw-rw-r-- 1 son son  4028  6월 30 20:47 str_swap.c
-rw-rw-r-- 1 son son  3994  6월 30 02:14 str_swap_fail.c
son@son-ThinkPad-X1-Carbon-Gen-8:~/proj/eddi/academy/EmbeddedMasterLv2/SPH/linux/system_programming/file_system$
```

☒ 가장 중요한 open의 반환 값은 앞에서 살펴 봤듯이 file 구조체 포인터 배열(fd_array)의 인덱스 값이다 즉, 파일 디스크립터 값이 된다

파일 API

EDDI
Electronic Design Development Institute

- read
  - → 파일의 내용을 원하는 바이트 만큼 읽어올 수 있게 한다

  size_t read(int fd, void *buff, size_t nbytes)

  - → 첫 번째 파라메타는 파일디스크립터이고 open 시스템콜을 통해 얻는다
  - → 두 번째는 파일에서 읽은 내용을 저장할 버퍼이다
  - → 세 번째는 읽어오고자 하는 내용의 바이트 사이즈이다
  - → 여기서 중요한 내용이 하나 더 있는데 seek_pointer라는 파일에는 파일의 내용을 가리키는 커서가 있다
  - → 이 커서는 read후에 위치가 변경되는데 그 위치는 nbytes의 다음 위치에 위치한다

  예를 들어 파일의 내용이 test apple이라 하고 nbyte = 5였다면 read 시스템 콜 이후 seek_pointe는 'a'를 가리키고 있다
  즉 현재 커서의 위치가 'a'에 있다

  - → 반환 값은 읽은 내용의 byte수가 된다. 실패는 -1, 파일의 맨 끝에 커서가 위치한 상태에서 읽으면 0을 반환
- write
  - → 파일에 원하는 내용을 원하는 바이트 만큼 쓸 수 있게 한다

  size_t write(int fd, void *buff, size_t nbytes)

  - → 첫 번째 파라메타는 파일디스크립터이고 open 시스템콜을 통해 얻는다
  - → 두 번째는 파일에 쓸 내용을 저장한 버퍼이다
  - → 세 번째는 쓰고자 하는 내용의 바이트 사이즈이다
  - → read와 마찬가지로 write 시스템 콜이 완료 후 seek_pointer는 nbyte에 위치한다
  - → 반환 값은 쓴 내용의 byte수, 실패는 -1

파일 API

EDDI
Electronic Design Development Institute

- lseek
  - → seek_pointer를 원하는 위치로 설정하는 함수 이다

off_t lseek(int fd, off_t offset, int whence)

  - → fd : 파일 디스크립터 값, offset : 기준점으로 부터 이동할 거리, whence : 기준점
  - → whence는 다음과 같은 값을 가진다

(1) SEEK_SET : 파일의 맨 앞
(2) SEEK_CUR : 현재 seek 포인터
(3) SEEK_END : 파일의 맨 끝

  - → lseek를 통해 파일 전체의 크기 또는 현재 seek_pointer까지의 파일 크기를 알 수 있다
  - → 예를 들어 아래와 같이 test.txt의 내용이 들어 있다하면

test apple test water

seek_pointer
offset = 0

파일 API

EDDI
Electronic Design Development Institute

➢ lseek

off_t lseek(int fd, off_t offset, int whence)

→ 예를 들어 아래와 같이 test.txt의 내용이 들어 있다하면

test apple test water

seek_pointer
offset = 0

→ 파일의 전체 크기를 lseek를 통해 알고자 한다면 lseek(fd, 0, SEEK_END) 를 통해 알 수 있다

test apple test water

offset = 0 seek_pointer

→ 문자는 21byte이지만 SEEK_END를 통해 맨 마지막 까지 계산되므로 lseek를 통해 반환된 값은 22가 된다

파일 API

EDDI
Electronic Design Development Institute

➢ lseek

off_t lseek(int fd, off_t offset, int whence)

→ read, write 후 seek_pointer의 위치가 바뀐다

→ 예를 들어 아래와 같이 test.txt의 내용이 들어 있고, open(fd, buf, 문자열 전체 사이) 시스템 콜 후 seek_pointer가 다음과 같이 위치하게 된다

test success!

seek_pointer

→ 위 상태에서 lseek(fd, 0, SEEK_CUR)를 하게 되면 14가 return 된다

```
buf = t
buf = e
buf = s
buf = t
buf =
buf = s
buf = u
buf = c
buf = c
buf = e
buf = s
buf = s
buf = !
buf =

current seek_pointer = 14
file size = 14
son@son-ThinkPad-X1-Carbon-Gen-8:~/proj/eddi/academy/E4DS-Linux-System-Programming/basic_file$
```

파일 API

EDDI
Electronic Design Development Institute

➢ lseek

off_t lseek(int fd, off_t offset, int whence)

→ read, write 후 seek_pointer의 위치가 바뀐다

→ 예를 들어 아래와 같이 test.txt의 내용이 들어 있고, write(fd, “test”, 4) 시스템 콜을 하게 되면 seek_pointer는 아래와 같이 위치하게 된다

test

seek_pointer

Q. write 시스템 콜을 하게 되면 자동으로 seek_pointer의 위치가 다음 내용으로 변경되는 건가요?

```
write(fd, buf, 4);
printf("buf = %s\n", buf);
printf("current seek_pointer = %d\n", (int)lseek(fd, (off_t)0, SEEK_CUR));
printf("write data = sucess\n");
write(fd, "success", 7);
printf("current seek_pointer = %d\n", (int)lseek(fd, (off_t)0, SEEK_CUR));
```

```
son@son-ThinkPad-X1-Carbon-Gen-8:~/proj/eddi/academy/E4DS-Linux-System-Programming/basic_file$ ./file4
buf = test
current seek_pointer = 4
write data = sucess
current seek_pointer = 11
file size = 12
son@son-ThinkPad-X1-Carbon-Gen-8:~/proj/eddi/academy/E4DS-Linux-System-Programming/basic_file$ cat file_test.txt
testsuccess
```

위 예상된 출력은 “tessuccess”이지만 실제 실행 결과는 “testsuccess”가 나왔습니다
test 쓰기 후 현재 0 ~ SEEK_CUR까지 값이 4가 나왔으므로 마지막 ‘t’에 위치한 것으로 생각했습니다.

파일 API응용 - 문자열 바꾸기 기능 만들기

EDDI
Electronic Design Development Institute

➢ 구현전략

☒ “파일 명” “문자열1” “문자열2”를 입력하여 파일에서 문자열1을 문자열2로 바꾼다

1. 커맨드라인 인수를 통해 “파일명“ “문자열1” “문자열2”를 입력 받는다
2. 파일명에 따라 open 시스템 콜을 호출하여 파일을 RDWR로 연다
3. 문자열 1의 크기를 구한다
4. lseek를 통해 파일 전체의 크기를 얻는다
5. 파일에서 문자열1을 찾는다
   (1) 파일을 스캔할 수 있는 스캔사이즈를 문자열1의 사이즈로 한다
   (2) 아래와 같이 파일을 스캔한다

   (3) lseek를 통해 파일의 맨 처음 부터 1byte씩 seek_pointer의 위치를 변경하고 seek_pointer의 위치를 버퍼에 저장한다
   (4) 스캔사이즈 만큼 read를 통해 파일의 내용을 읽는다
   (5) 문자열1과 파일의 내용이 일치하는지 확인한다
   (6) 일치하면 교체시작 불일치하면 (3) 부터 반복
6. 문자열1과 파일의 내용이 일치한 경우
   (1) 파일의 문자열1 다음 내용을 버퍼에 복사한다
   (2) 문자열1을 문자열2로 교체한다
   (3) 문자열 복사한 문자열1의 다음 내용을 문자열2 다음에 붙여 넣는다
7. 3 ~ 6을 파일 전체 크기만큼 반복 한다