Linux System Programming #2

Spring, 2020

Soongsil University, Seoul, Korea

Jiman Hong

creat()

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
int creat(const char *pathname, mode_t mode);

리턴 값 : 성공 시 쓰기 전용으로 열린 파일 디스크립터, 에러 시 -1 (errno 설정)
```

- **파일을 생성할 때 사용하는 시스템호출**
- **상태 플래그를 사용하지 않는다는 점만 제외하면 open()와 동일**
  - open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
- **파일에 쓴 데이터를 다시 읽기**
  - open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);

creat()예제 1 (p.69)

<ssu_creat_1.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>


int main(void)
{
    char *fname = "ssu_test.txt";
    int fd;


    if ((fd = creat(fname, 0666)) < 0) {
        fprintf(stderr, "creat error for %s₩n", fname);
        exit(1);
    }
    else {
        printf("Success!₩nFilename : %s₩nDescriptor : %d₩n", fname, fd);
        close(fd);
    }


    exit(0);
}
```

>>> 실행 결과

```
root@localhost:/home/oslab# ./ssu_creat_1
Success!
Filename : ssu_test.txt
Descriptor : 3
```

creat()예제 2 (p.70)

<ssu_creat_2.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>


int main(void)
{
    char *fname = "ssu_test.txt";
    int fd;


    if ((fd = creat(fname, 0666)) < 0) {
        fprintf(stderr, "creat error for %s₩n", fname);
        exit(1);
    }
    else {
        close(fd);
        fd = open(fname, O_RDWR);
        printf("Succeeded!₩n<%s> is new readable and writable₩n", fname);
    }


    exit(0);
}
```

>>> 실행 결과

```
root@localhost:/home/oslab# ./ssu_creat_2
Succeeded!
<ssu_test.txt> is new readable and writable
```

lseek()

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);

리턴 값 : 성공 시 새 파일 오프셋, 에러 시 –1을 리턴하고 errno가 설정됨
```

- 오픈된 파일의 오프셋 위치(offset)를 명시적으로 변경하는 시스템호출
- 파일의 새로운 오프셋 리턴
- 파일 오프셋 위치만 변경. 별도의 I/O 없음
- read()와 write()는 파일의 현재 오프셋 위치부터 수행
- read()write() 후 새로운 오프셋 위치는 이전 오프셋 위치에 지정된 바이트 수만큼 더해진 값으로 갱신

lseek()

□ 파일의 현재 위치와 다른 위치에 I/O를 하기 위해서는 파일의 오프셋 위치를 지정 => whence <표 2-5>

플래그	내용
SEEK_SET	파일의 오프셋 위치를 파일의 처음으로 지정. 파일의 처음부터 입출력을 하고자 한다면 이 값을 지정하고 offset 인자를 0으로 지정하면 됨
SEEK_CUR	파일의 오프셋은 현재 오프셋 위치에 offset 인자를 더한 값으로 설정됨. 즉, 오프셋 설정 시의 기준 위치가 파일의 현재 오프셋이 됨. 인자로 지정한 offset은 음수이거나 양수일 수 있음
SEEK_END	파일의 오프셋은 파일 크기에 offset이 더해져서 인자로 설정됨. 오프셋 설정 시의 기준 위치가 파일의 마지막 위치가 되는 것으로 파일의 끝에 새로운 데이터를 추가하고자 한다면 이 값과 함께 offset 인자를 0으로 지정하면 됨. 인자로 지정한 offset은 음수이거나 양수일 수 있음

lseek()

□ 프로그래밍 Tip

- 파일의 오프셋은 그 파일의 실제 크기보다 클 수 있음
  - write()에서 초과된 오프셋만큼 파일의 크기가 커짐
  - 파일의 홀(hole)lseek()가 성공적으로 호출되면, 새로운 파일 오프셋 리턴
- 현재 오프셋 위치를 알려면 현재 오프셋 위치로부터 0바이트를 지정

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
// 어떤 파일이 lseek()가 호출 가능한지 아닌지를 확인하기 위해 사용하기도 함
```

- Linux에서 FIFO나 파이프 파일인 경우 -1 리턴/errno : EPIPE
- 기존 파일 끝에 데이타 추가 방법

1)
```
filedes=open(filename, O_RDWR);
lseek(filedes, (off_t)0, SEEK_END);
write(filedes, buf, BUFSIZE);
```

2)
```
filedes=open(filname, O_RDWR | O_APPEND);
write(filedes, buf, BUFSIZE
```

- 파일의 크기 확인 방법

```
off_t filesize;
filesize = lseek(filedes, (off_t)0, SEEK_END);
```

lseek()예제 1 (p.74)

<ssu_test.txt>

```
Linux System Programming!
Unix System Programming!
Linux Mania
Unix Mania
```

<ssu_lseek_1.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

int main(void)
{
    char *fname = "ssu_test.txt";
    off_t fsize;
    int fd;

    if ((fd = open(fname, O_RDONLY)) < 0) {
        fprintf(stderr, "open error for %s₩n", fname);
        exit(1);
    }

    if ((fsize = lseek(fd, 0, SEEK_END)) < 0) {
        fprintf(stderr, "lseek error₩n");
        exit(1);
    }

    printf("The size of <%s> is %ld bytes.₩n", fname, fsize);

    exit(0);
}
```

>>> 실행 결과

```
root@localhost:/home/oslab# ./ssu_lseek_1
The size of <ssu_test.txt> is 74 bytes.
```

lseek()예제 2 (p.75)

<ssu_lseek_2.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define CREAT_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

char buf1[] = "1234567890";
char buf2[] = "ABCDEFGHIJ";

int main(void)
{
    char *fname = "ssu_hole.txt";
    int fd;

    if ((fd = creat(fname, CREAT_MODE)) < 0) {
        fprintf(stderr, "creat error for %s₩n", fname);
        exit(1);
    }

    if (write(fd, buf1, 12) != 12) {
        fprintf(stderr, "buf1 write error₩n");
        exit(1);
    }

    if (lseek(fd, 15000, SEEK_SET) < 0) {
        fprintf(stderr, "lseek error₩n");
        exit(1);
    }

    if (write(fd, buf2, 12) != 12) {
        fprintf(stderr, "buf2 write error₩n");
        exit(1);
    }

    exit(0);
}
```

실행 결과 [Ubuntu, Fedora]

```
root@localhost:/home/oslab# ./ssu_lseek_2
root@localhost:/home/oslab# ls -l ssu_hole.txt
-rw-r--r-- 1 root root 15012 Jan 3 03:53 ssu_hole.txt
root@localhost:/home/oslab# od -c ssu_hole.txt
0000000 1 2 3 4 5 6 7 8 9 0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0
0000020 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0
*
0035220 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 A B C D E F G H
0035240 I J ₩0 ₩0
0035244
```

실행 결과 [macOS]

```
ssu-ui-MacBook-Air:oslab root# od -c ssu_hole.txt
0000000 1 2 3 4 5 6 7 8 9 0 ₩0 A ₩0 ₩0 ₩0 ₩0
0000020 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0
*
0035220 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 ₩0 A B C D E F G H
0035240 I J ₩0 ₩0
0035244
```

read()

```
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t nbytes);

리턴 값: 읽은 바이트 수, 파일의 끝에 도달 한 경우 0, 에러 시 –1을 리턴하고 errno가 설정
```

□ 오픈된 파일에서 데이터를 읽을 때 사용 시스템호출

- 읽기 전용 또는 읽기·쓰기 혼용 접근 모드으로 오픈된 파일(filedes)로부터 지정된 메모리(buf)로 인자 nbytes만큼의 데이터를 읽음
- 성공 시 : 실제로 읽은 바이트 수가 리턴 => 파일의 현재 오프셋 위치는 이전 오프셋 위치에 실제 읽은 바이트 수만큼이 더해진 값으로 갱신
  - nbytes가 0이면 0을 리턴
  - 리턴 값인 파일을 실제로 읽은 바이트 수는 읽기를 요청한 바이트 수보다 적을 수 있음
  - 예) 현재 오프셋 위치에서 남은 바이트가 50 바이트 밖에 안 되는데 100 바이트의 읽기 요청을 하게 되면 50 바이트만큼의 데이터만 메모리로 읽고, 리턴 값도 50이 된다
  - read()로 파일의 끝을 읽으려 한 경우에는 0이 리턴 => 한 파일에 대해 파일 읽기를 연속해서 시도하면 리턴 값이 0이 되어 파일의 끝에 도달 => read()를 사용할 때는 리턴 값 자주 확인 필요

read()

일반 파일에 요청된 수만큼의 바이트들을 읽기 전에 파일의 끝에 도달했을 때
터미널 파일에서 자료를 읽을 때
네트워크에서 자료를 읽을 때
파이프나 FIFO에서 자료를 읽을 때
레코드 기반 장치에서 자료를 읽을 때
요청된 크기의 자료 중 일부만 읽은 상황에서 시그널에 의해 연산이 가로채였을 때

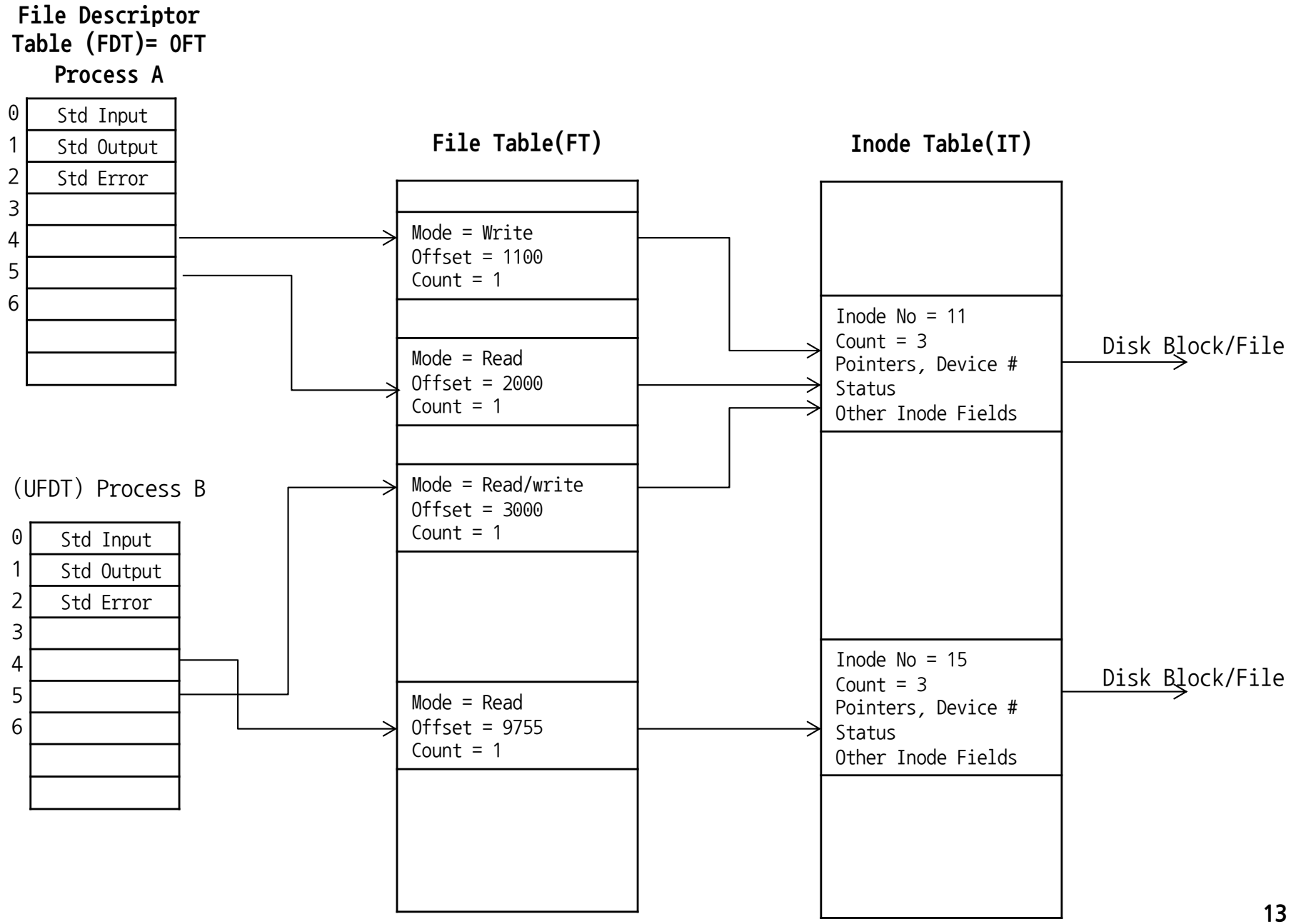
□ 참고

- 한 프로세스가 read()/write를 호출하게 되면, 커널은 호출이 수행되는 동안 해당 파일의 i-node에 락을 걸고 해당 파일에 접근하려는 다른 모든 연산을 블록(접근을 허용하지 않음)시킴.
  - read()가 즉시 리턴하지 못하고 블록 되는 경우 그 사이 다른 프로세스가 먼저 자신의 데이터를 쓸 수 있으며, 이럴 경우 데이터의 일관성이 깨질 수 있음
  - inode = index node

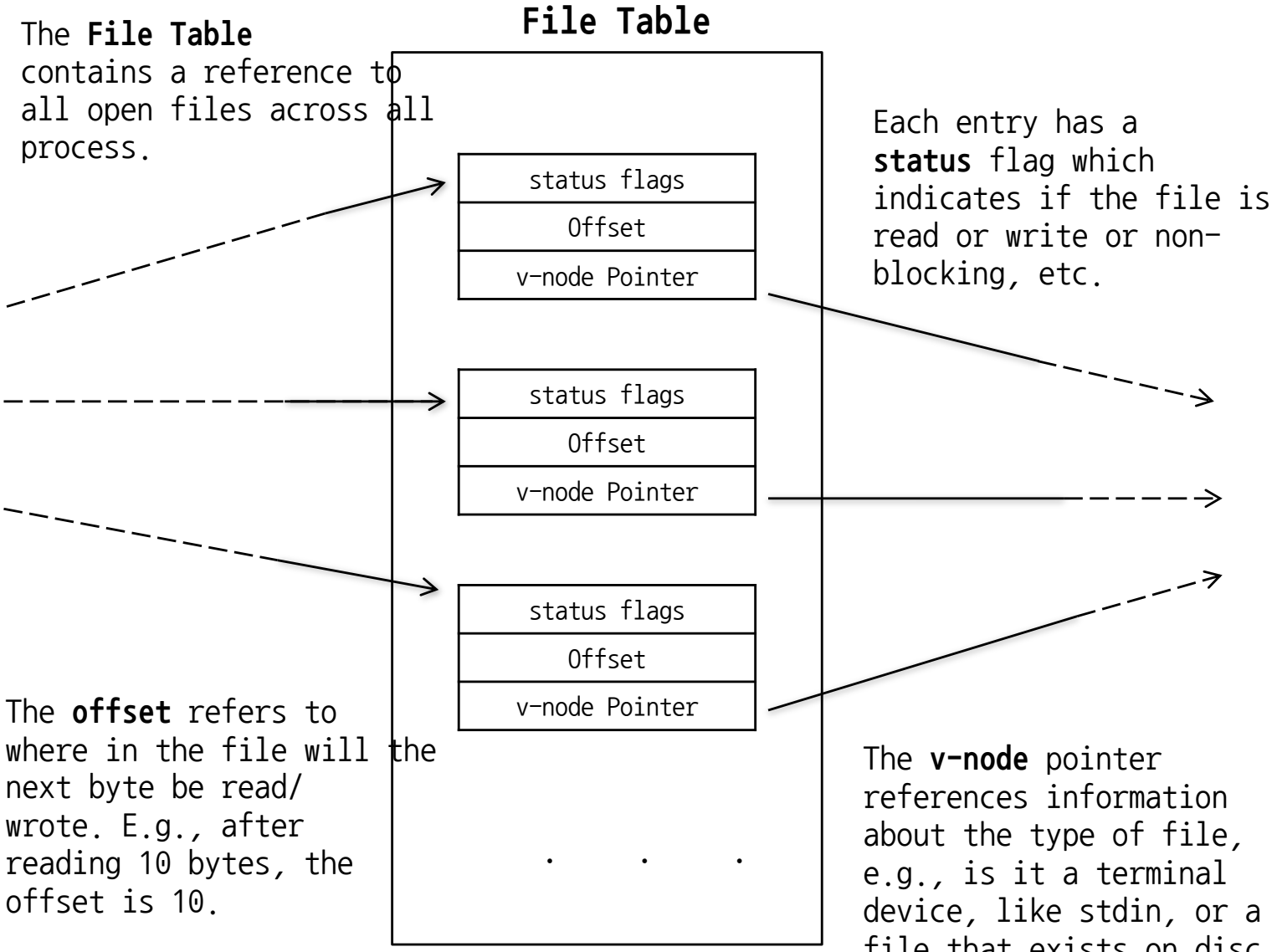
FDT vs FT vs IT

- 파일디스크립터 테이블
  - 로컬구조 : 모든 프로세스가 한개씩
  - 기본적으로 0, 1, 2 항목은 프로세스 생성 시 자동 오픈
  - FT 한 항목 포인터
- 파일테이블
  - 전역구조 -> 모든 프로세스가 오픈한 파일에 대한 참조
  - 파일디스크립터 테이블에서 포인팅(참조)되는 수 (ref. count), 공유 목적
  - (1) 상태 정보 access mode (read/write/non-blocking)
    - 프로세스간 또는 프로세스 내에서 공유(fork()/dup())
  - (2) offset into the file
  - (3) IT 한 항목 포인터
- 아이노드 테이블 (disk IT vs In-Core IT) / vnode table

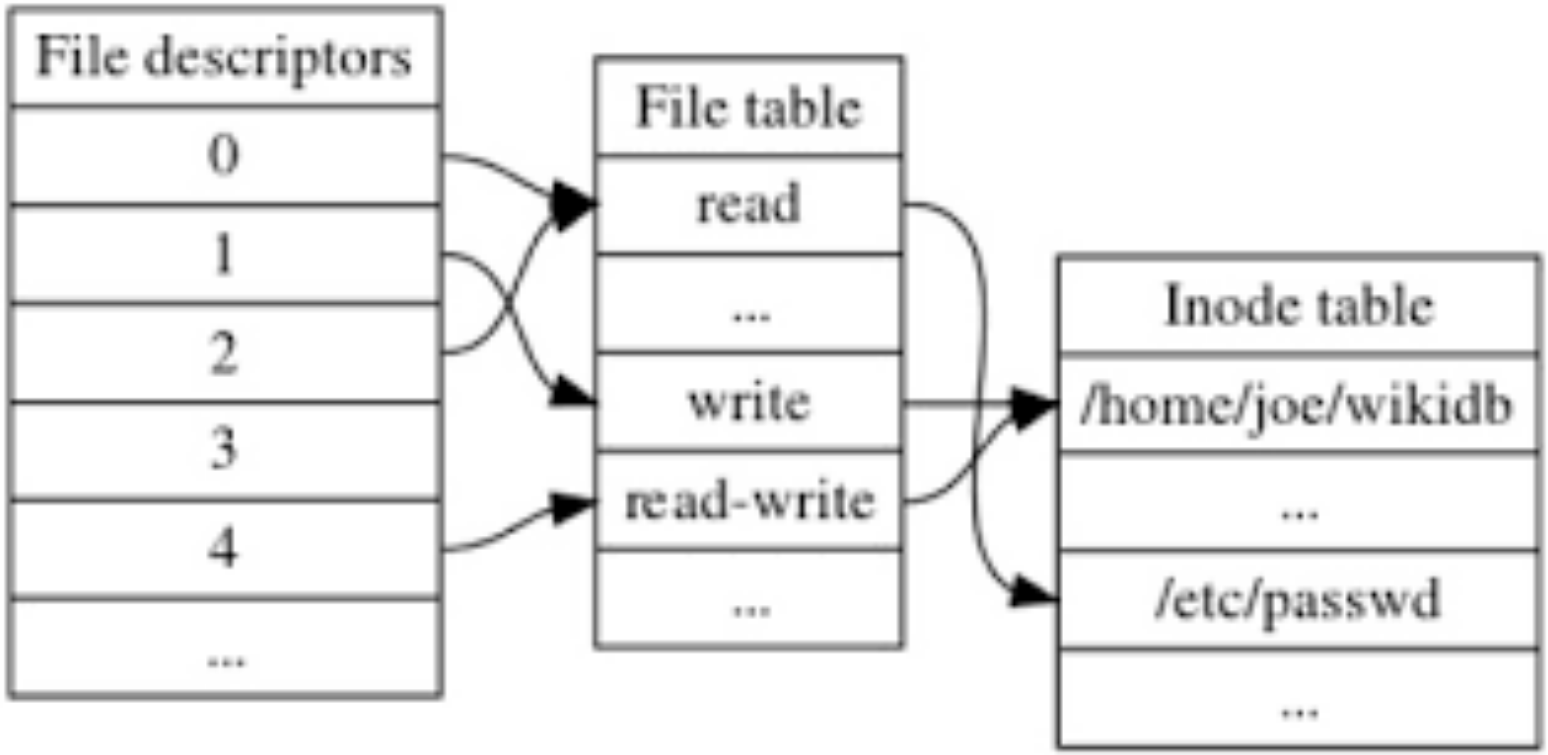
File Related Table in the OLD UNIX



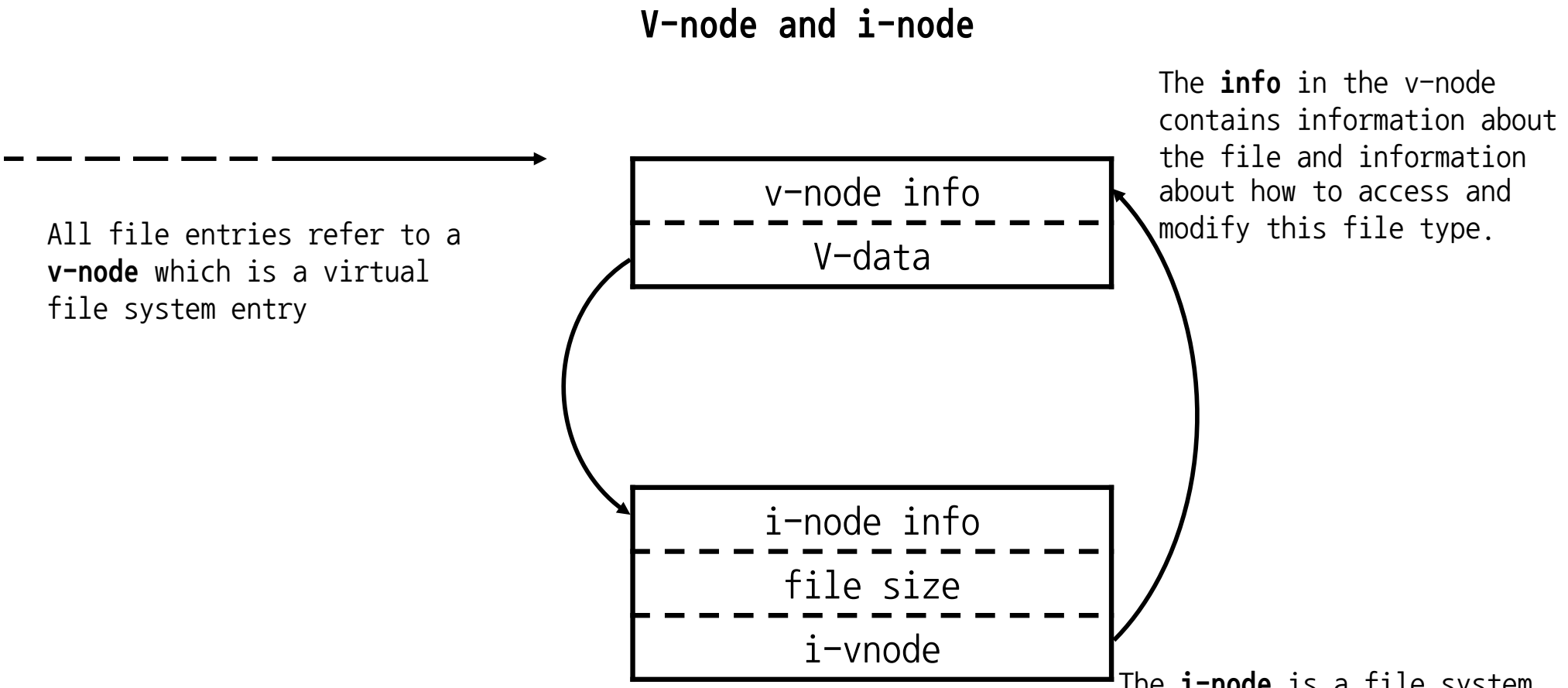
FT


File Table
The File Table contains a reference to all open files across all process.
status flags
Offset
v-node Pointer
status flags
Offset
v-node Pointer
status flags
Offset
v-node Pointer
Each entry has a status flag which indicates if the file is read or write or non-blocking, etc.
The offset refers to where in the file will the next byte be read/ wrote. E.g., after reading 10 bytes, the offset is 10.
The v-node pointer references information about the type of file, e.g., is it a terminal device, like stdin, or a file that exists on disc.

Sharing


File descriptors
0
1
2
3
4
...
File table
read
...
write
read-write
...
Inode table
/home/joe/wikidb
...
/etc/passwd
...

Vnode vs Inode



All file entries refer to a **v-node** which is a virtual file system entry

The **info** in the v-node contains information about the file and information about how to access and modify this file type.

On many Linux systems only i-nodes are used, with a generic i-node flavor, which is the same as v-node.

The **i-node** is a file system specific structure containing information about the file within that file system, such as it's size. It also refers back to the v-node.

read()예제 1 (p.78)

<ssu_employee.h>

```
#define NAME_SIZE 64

struct ssu_employee {
char name[NAME_SIZE];
int pid;
int salary;
};
```

<ssu_read.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "ssu_employee.h"

int main(int argc, char *argv[])
{
    struct ssu_employee record;
    int fd;
    int record_num;

    if (argc < 2) {
        fprintf(stderr, "Usage : %s file₩n", argv[0]);
        exit(1);
    }

    if ((fd = open(argv[1], O_RDONLY)) < 0) {
        fprintf(stderr, "open error for %s₩n", argv[1]);
        exit(1);
    }

    while (1) {
        printf("Enter record number : ");
        scanf("%d", &record_num);

        if (record_num < 0)
            break;

        if (lseek(fd, (long)record_num * sizeof(record), 0) < 0) {
            fprintf(stderr, "lseek error₩n");
            exit(1);
        }

        if (read(fd, (char *)&record, sizeof(record)) > 0)
            printf("Employee : %s Salary : %d₩n", record.name, record.salary);
        else
            printf("Record %d not found₩n", record_num);
    }

    close(fd);
    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_read ssu_employeefile
Enter record number : 1
Employee : LeeSoonShin Salary : 1500000
Enter record number : 2
Employee : JangyeongSil Salary : 2000000
Enter record number : 56
Record 56 not found
Enter record number : -1
```

read()예제 2 (p.80)

<ssu_read_2.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 1024

int main(void)
{
    char buf[BUFFER_SIZE];
    char *fname = "ssu_test.txt";
    int count;
    int fd1, fd2;

    fd1 = open(fname, O_RDONLY, 0644);
    fd2 = open(fname, O_RDONLY, 0644);

    if (fd1 < 0 || fd2 < 0) {
        fprintf(stderr, "open error for %s₩n", fname);
        exit(1);
    }

    count = read(fd1, buf, 25);
    buf[count] = 0;
    printf("fd1's first printf : %s₩n", buf);
    lseek(fd1, 1, SEEK_CUR);
    count = read(fd1, buf, 24);
    buf[count] = 0;
    printf("fd1's second printf : %s₩n", buf);
    count = read(fd2, buf, 25);
    buf[count] = 0;
    printf("fd2's first printf : %s₩n", buf);
    lseek(fd2, 1, SEEK_CUR);
    count = read(fd2, buf, 24);
    buf[count] = 0;
    printf("fd2's second printf : %s₩n", buf);

    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_read_2
fd1's first printf :Linux System Programming!
fd1's second printf :Unix System Programming!
fd2's first printf :Linux System Programming!
fd2's second printf :Unix System Programming!
```

read()예제 3 (p.82)

<ssu_read_3A.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    char c;
    int fd;

    if ((fd = open("ssu_test.txt", O_RDONLY)) < 0) {
        fprintf(stderr, "open error for %s₩n", "ssu_test.txt");
        exit(1);
    }

    while (1) {
        if (read(fd, &c, 1) > 0)
            putchar(c);
        else
            break;
    }

    exit(0);
}
```

<ssu_read_3B.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    char character;
    int fd;
    int line_count = 0;

    if ((fd = open("ssu_test.txt", O_RDONLY)) < 0) {
        fprintf(stderr, "open error for %s₩n", "ssu_test.txt");
        exit(1);
    }

    while (1) {
        if (read(fd, &character, 1) > 0) {
            if (character == '₩n')
                line_count++;
        }
        else
            break;
    }

    printf("Total line : %d₩n", line_count);
    exit(0);
}
```

실행 결과 [Ubuntu]

```
root@localhost:/home/oslab# ./ssu_read_3A & ./ssu_read_3B
[1] 12787
Total line : 4
root@localhost:/home/oslab# Linux System Programming!
Unix System Programming!
Linux Mania
Unix Mania
^C
[1]+ Done ./ssu_read_3A
```

실행 결과 [Fedora, macOS]

```
[root@localhost 3]# ./ssu_read_3A & ./ssu_read_3B
[1] 28131
Linux System Programming!
Unix System Programming!
Linux Mania
Unix Mania
Total line : 4
[1]+ Done ./ssu_read_3A
```

read()예제 4 (p.84)

<ssu_read_4.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define TABLE_SIZE 128
#define BUFFER_SIZE 1024

int main(int argc, char *argv[])
{
    static struct {
        long offset;
        int length;
    } table [TABLE_SIZE];
    char buf[BUFFER_SIZE];
    long offset;
    int entry;
    int i;
    int length;
    int fd;

    if (argc < 2) {
        fprintf(stderr, "usage: %s <file>₩n", argv[0]);
        exit(1);
    }

    if ((fd = open(argv[1], O_RDONLY)) < 0) {
        fprintf(stderr, "open error for %s₩n", argv[1]);
        exit(1);
    }

    entry = 0;
    offset = 0;
    while ((length = read(fd, buf, BUFFER_SIZE)) > 0) {
        for (i = 0 ; i < length ; i++) {
        table[entry].length++;
        offset++;

        if (buf[i] == '₩n')
            table[++entry].offset = offset;
    }
}

#ifdef DEBUG
    for (i = 0 ; i < entry ; i++)
        printf("%d : %ld, %d₩n", i + 1, table[i].offset, table[i].length);
#endif

    while (1) {
        printf("Enter line number : ");
        scanf("%d", &length);

        if (--length < 0)
            break;

        lseek(fd, table[length].offset, 0);

        if (read(fd, buf, table[length].length) <= 0)
        continue;

        buf[table[length].length] = '₩0';
        printf("%s", buf);
    }

    close(fd);
    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_read_4 /etc/group
Enter line number : 1
root:x:0:
Enter line number : 2
daemon:x:1:
Enter line number : 3
bin:x:2:
Enter line number : 4
sys:x:3:
Enter line number : 1
root:x:0:
Enter line number : -1
```

write()

```
#include <unistd.h>
ssize_t write(int filedes, const void *buf, size_t nbytes);

리턴 값: 성공 시 기록된 바이트 수, 에러 시 –1을 리턴하고 errno가
설정됨
```

□ 오픈한 파일에 데이터를 쓸 때 사용하는 시스템호출

- 지정된 메모리로부터 쓰기 전용 또는 읽기·쓰기 혼용 접근 권한으로 오픈된 파일로 정해진 바이트만큼의 데이터를 씀
- write()의 호출이 완료되면 파일의 현재 오프셋 위치는 이전 오프셋 위치에 실제로 쓴 바이트 수만큼이 더해진 값으로 갱신 => 리턴 값도 파일에 실제로 쓴 데이터의 바이트 수

write()예제 1 (p.87)

<ssu_write_1.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(void)
{
    char buf[BUFFER_SIZE];
    int length;

    length = read(0, buf, BUFFER_SIZE);
    write(1, buf, length);
    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_write_1
hello os
hello os
```

write()예제 2 (p.88)

<ssu_write_2.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define S_MODE 0644
#define BUFFER_SIZE 1024

int main(int argc, char *argv[])
{
    char buf[BUFFER_SIZE];
    int fd1, fd2;
    int length;

    if (argc != 3) {
        fprintf(stderr, "Usage : %s filein fileout₩n", argv[0]);
        exit(1);
    }

    if ((fd1 = open(argv[1], O_RDONLY)) < 0) {
        fprintf(stderr, "open error for %s₩n", argv[1]);
        exit(1);
    }

    if ((fd2 = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_MODE)) < 0) {
        fprintf(stderr, "open error for %s₩n", argv[2]);
        exit(1);
    }

    while ((length = read(fd1, buf, BUFFER_SIZE)) > 0)
        write(fd2, buf, length);

    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_write_2 /etc/passwd password
root@localhost:/home/oslab# ls -l /etc/passwd password
-rw-r--r-- 1 root root 2240 Jan 2 23:55 /etc/passwd
-rw-r--r-- 1 root root 2240 Jan 3 05:27 password
```

write()예제 3 (p.89)

<ssu_employee.h>

```
#define NAME_SIZE 64

struct ssu_employee {
    char name [NAME_SIZE];
    int pid;
    int salary;
};
```

<ssu_write_3.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "ssu_employee.h"

int main(int argc, char *argv[])
{
    struct ssu_employee record;
    int fd;

    if (argc < 2) {
        fprintf(stderr, "usage : %s file₩n", argv[0]);
        exit(1);
    }

    if ((fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0640)) < 0) {
        fprintf(stderr, "open error for %s₩n", argv[1]);
        exit(1);
    }

    while (1) {
        printf("Enter employee name <SPACE> salary: ");
        scanf("%s", record.name);

        if (record.name[0] == '.')
            break;

        scanf("%d", &record.salary);
        record.pid = getpid();
        write(fd, (char *)&record, sizeof(record));
    }

    close(fd);
    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_write_3 ssu_employeefile
Enter employee name <SPACE> salary: HongGilDong 1000000
Enter employee name <SPACE> salary: LeeSoonShin 1500000
Enter employee name <SPACE> salary: JangyeongSil 2000000
Enter employee name <SPACE> salary: AhnChangHo 1200000
Enter employee name <SPACE> salary: KimGoo 1800000
Enter employee name <SPACE> salary: .
root@localhost:/home/oslab# ls –l ssu_employeefile
-rw-r----- 1 root root 360 Jan 3 05:37 ssu_employeefile
```

pread() / pwrite()

```
#include <unistd.h>
ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);
리턴 값: 성공 시 읽은 바이트 수를 리턴, 파일의 끝에 도달하면 0을 리턴
요청한 바이트 수보다 적은 값이 리턴되더라도 에러 아님
에러 발생 시 –1을 리턴하며 상응하는 errno가 설정

ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
리턴 값: 성공 시 기록된 바이트 수를 리턴
쓰여야 할 바이트 수보다 적은 값이 리턴되더라도 에러 아님
에러 발생 시 –1을 리턴하며 상응하는 errno가 설정
```

- pread()/ pwrite()는 오픈된 파일에 파일 탐색과 입출력 연산을 원자적(atomic)으로 수행할 때 사용하는 시스템호출
  - pread() : 오픈된 파일(filedes)의 offset 위치에 있는 데이터를 지정된 크기(nbytes)만큼 지정된 메모리(buf)로 데이타를 읽어옴
  - pwrite() : 메모리(buf)에 있는 데이터를 정해진 바이트(nbytes) 만큼 오픈된 파일의 offset부터 씀

dup() / dup2()

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);

리턴 값: 성공 시 새로운 파일 디스크립터를 리턴. dup2()는 filedes2 리턴
에러 발생 시 –1을 리턴하며 상응하는 errno가 설정됨
```

- dup()/ dup2() : 기존 파일디스크립터를 복사를 위한 시스템호출
- 호출이 성공되면 파일디스크립터 테이블 상에서 현재 사용되지 않는 가장 적은 값의 새로운 파일 디스크립터를 리턴 => 리턴 값이 파일디스크립터 테이블 상의 가장 작은 미사용 항목 보장 => FD 지정을 원하면 dup2()사용
  - dup2() = close(filedes2); fcntl(filedes, F_DUPFN, filedes2);

dup() / dup2()

- 주의할 점
  - dup()에 의해 복사된 파일디스크립터와 원본 파일디스크립터는 파일의 플래그가 다를 수 있음 => dup() 호출 시 파일 플래그는 복사되지 않고 초기화
  - filedes 인자는 반드시 현재 사용 중인 파일디스크립터가 지정되어야 하며, 그렇지 않은 경우 에러
  - dup2()의 두번째 인자인 filedes2가 이미 다른 프로세스에 의해 사용 중
    - close(filedes2); 후 새로 오픈하여 사용
  - dup2()의 첫번째 인자 filedes1과 두번째 인자인 filedes2 동일
    - 아무런 복사 없이 filedes 리턴

dup() 예제 1 (p.93)

<ssu_dup.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 1024

int main(void)
{
    char buf[BUFFER_SIZE];
    char *fname = "ssu_test.txt";
    int count;
    int fd1, fd2;

    if ((fd1 = open(fname, O_RDONLY, 0644)) < 0) {
        fprintf(stderr, "open error for %s₩n", fname);
        exit(1);
    }

    fd2 = dup(fd1);
    count = read(fd1, buf, 12);
    buf[count] = 0;
    printf("fd1's printf : %s₩n", buf);
    lseek(fd1, 1, SEEK_CUR);
    count = read(fd2, buf, 12);
    buf[count] = 0;
    printf("fd2's printf : %s₩n", buf);
    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_dup
fd1's printf : Linux System
fd2's printf : Programming!
```

dup2 ()예제 1 (p.94)

<ssu_dup2_1.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    char *fname = "ssu_test.txt";
    int fd;

    if ((fd = creat(fname, 0666)) < 0) {
        printf("creat error for %s₩n", fname);
        exit(1);
    }

    printf("First printf is on the screen.₩n");
    dup2(fd, 1);
    printf("Second printf is in this file.₩n");
    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_dup2_1
First printf is on the screen.
root@localhost:/home/oslab# cat ssu_test.txt
Second printf is in this file.
```

dup2 ()예제 2 (p.95)

<ssu_dup2_2.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 1024

int main(void)
{
    char buf[BUFFER_SIZE];
    char *fname = "ssu_test.txt";
    int fd;
    int length;

    if ((fd = open(fname, O_RDONLY, 0644)) < 0) {
        fprintf(stderr, "open error for %s₩n", fname);
        exit(1);
    }

    if (dup2(1, 4) != 4) {
        fprintf(stderr, "dup2 call failed₩n");
        exit(1);
    }

    while (1) {
        length = read(fd, buf, BUFFER_SIZE);

        if (length <= 0)
            break;

        write(4, buf, length);
    }

    exit(0);
}
```

실행 결과

```
root@localhost:/home/oslab# ./ssu_dup2_2
Linux System Programming!
Unix System Programming!
Linux Mania
Unix Mania
```

sync() / fsync() / fdatasync()

```
#include <unistd.h>
int fsync(int filedes);
int fdatasync(int filedes);

리턴 값: 성공 시 0을 리턴, 에러 발생 시 –1을 리턴하고 상응하는 errno가 설정됨
void sync(void);
```

- 디스크 상의 파일 시스템과 버퍼 캐쉬의 내용 간에 일관성을 보장하기 위해서 사용하는 시스템호출 => 동기화
- sync()와 fsync() : 버퍼 캐쉬의 내용을 디스크에 반영
  - sync() : 변경된 모든 내용을 디스크에 반영. I/O 완료 전 바로 리턴
  - fsync() : 인자로 지정한 파일의 변경된 내용만 디스크에 반영. I/O완료 후 리턴
- fdatasync(): fsync()와 유사. 파일의 일부 데이타만 동기화. 메타데이타(inode 속성 값)는 디스크에 반영하지 않음. 시간적으로 유리