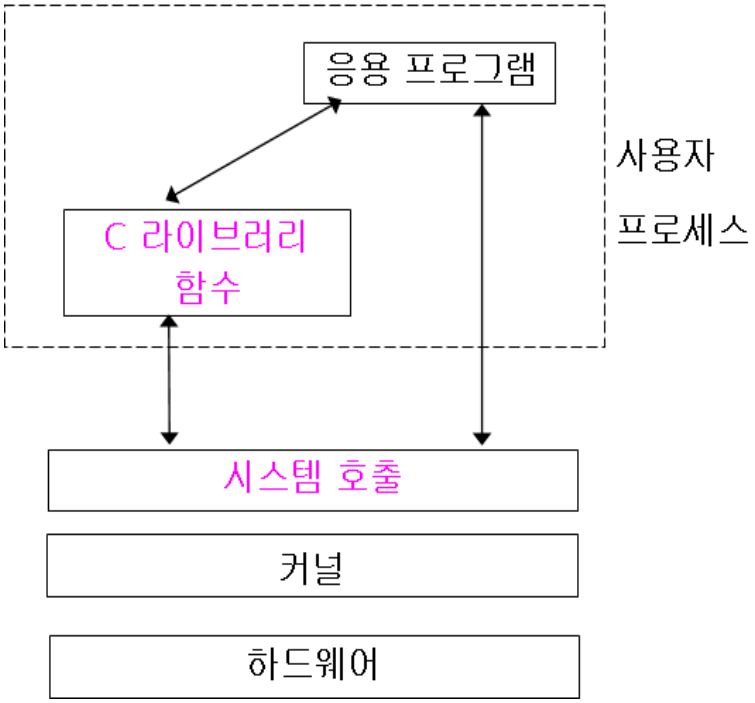
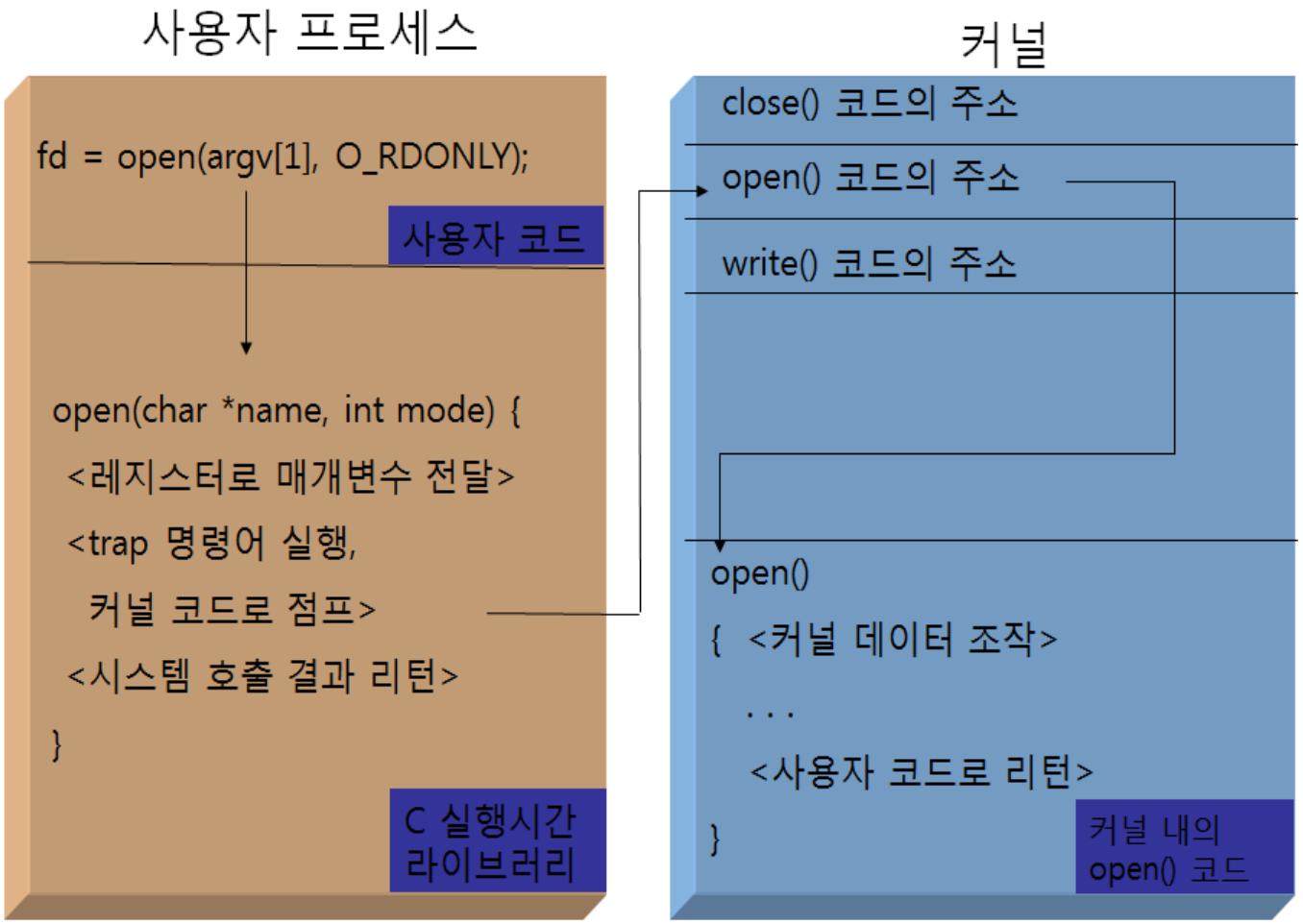
4.1 시스템 호출

시스템 호출(system call)

- 시스템 호출은 커널에 서비스 요청을 위한 프로그래밍 인터페이스
- 응용 프로그램은 시스템 호출을 통해서 커널에 서비스를 요청한다.



시스템 호출 과정


사용자 프로세스
fd = open(argv[1], O_RDONLY);
사용자 코드
open(char *name, int mode) {
<레지스터로 매개변수 전달>
<trap 명령어 실행,
커널 코드로 점프>
<시스템 호출 결과 리턴>
}
C 실행시간
라이브러리
커널
close() 코드의 주소
open() 코드의 주소
write() 코드의 주소
open()
{ <커널 데이터 조작>
. . .
<사용자 코드로 리턴>
}
커널 내의
open() 코드

시스템 호출 요약

주요 자원	시스템 호출
파일	open(), close(), read(), write(), dup(), lseek() 등
프로세스	fork(), exec(), exit(), wait(), getpid(), getppid() 등
메모리*	malloc(), calloc(), free() 등
시그널	signal(), alarm(), kill(), sleep() 등
프로세스 간 통신	pipe(), socket() 등

4.2 파일

파일 열기: open()

- 파일을 사용하기 위해서는 먼저 open() 시스템 호출을 이용하여 파일을 열어야 한다.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int oflag, [ mode_t mode ]);
파일 열기에 성공하면 파일 디스크립터를, 실패하면 -1을 리턴
```

- 파일 디스크립터는 열린 파일을 나타내는 번호이다.

파일 열기: open()

- oflag
  - O_RDONLY
    읽기 모드, read() 호출은 사용 가능
  - O_WRONLY
    쓰기 모드, write() 호출은 사용 가능
  - O_RDWR
    읽기/쓰기 모드, read(), write() 호출 사용 가능
  - O_APPEND
    데이터를 쓰면 파일 끝에 첨부된다.
  - O_CREAT
    해당 파일이 없는 경우에 생성하며
    mode는 생성할 파일의 사용권한을 나타낸다.

파일 열기: open()

- oflag
  - O_TRUNC
    파일이 이미 있는 경우 내용을 지운다.
  - O_EXCL
    O_CREAT와 함께 사용되며 해당 파일이 이미 있으면 오류

파일 개방에 사용되는 플래그	
O_RDONLY	파일을 읽기 전용으로 개방한다. 읽기 이외의 다른 작업을 수행할 수 없다.
O_WRONLY	파일을 쓰기 전용으로 개방한다. 쓰기 이외의 다른 작업을 수행할 수 없다.
O_RDWR	파일을 읽기와 쓰기가 동시에 가능한 상태로 개방한다.
O_CREAT	지정한 경로의 파일이 존재하지 않으면 새롭게 생성한 후 개방한다. 지정한 경로의 파일이 존재하면 지정한 상태로 개방한다.
O_EXCL	지정한 경로의 파일이 존재하지 않으면 새롭게 생성하나, 지정한 경로의 파일이 존재하면 open 호출을 실패한다. (※O_CREAT 플래그와 함께 사용해야 한다.)
O_APPEND	파일을 개방한 직후에 읽기/쓰기 포인터의 위치를 파일 내용의 마지막 바로 뒤로 이동한다.
O_TRUNC	기존 내용을 삭제하고 open함

파일 열기: 예

- fd = open("account",O_RDONLY);
- fd = open(argv[1], O_RDWR);
- fd = open(argv[1], O_RDWR | O_CREAT, 0600);
- fd = open("tmpfile", O_WRONLY|O_CREAT|O_TRUNC, 0600);
- fd = open("/sys/log", O_WRONLY|O_APPEND|O_CREAT, 0600);
- if ((fd = open("tmpfile", O_WRONLY|O_CREAT|O_EXCL, 0666))==-1)

fopen.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd;
    if ((fd = open(argv[1], O_RDWR)) == -1)
        printf("파일 열기 오류₩n");
    else printf("파일 %s 열기 성공 : %d₩n", argv[1], fd);

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

파일 생성: creat()

- creat() 시스템 호출
  - path가 나타내는 파일을 생성하고 쓰기 전용으로 연다.
  - 생성된 파일의 사용권한은 mode로 정한다.
  - 기존 파일이 있는 경우에는 그 내용을 삭제하고 연다.
  - 다음 시스템 호출과 동일

    open(path, WRONLY | O_CREAT | O_TRUNC, mode);

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

파일 생성에 성공하면 파일 디스크립터를, 실패하면 -1을 리턴

- 해당 파일을 개방함과 동시에 파일이 가지고 있는 데이터를 모두 삭제한다

```
filedes = creat(pathname, 0644);
   ...
filedes = open(pathmame, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

예제 프로그램

```
01 #include <fcntl.h>
02 #include <stdio.h>
03 int main()
04 {
05    int filedes1, filedes2;
06
07    filedes1 = open("data1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
08    filedes2 = creat("data2.txt", 0644);
09
10    close(filedes1);
11    close(filedes2);
12 }
```

파일 닫기: close()

- close() 시스템 호출은 fd가 나타내는 파일을 닫는다.

```
#include <unistd.h>
int close( int fd );
```

fd가 나타내는 파일을 닫는다.

성공하면 0, 실패하면 -1을 리턴한다.

개방된 파일을 닫지 않고 프로그램이 종료한 경우

- **프로그램이 종료할 때 개방된 파일은 커널에 의해 자동으로 닫힌다.**
- **이런 사실을 알고 있더라도 사용된 파일은 마지막에 닫아주는 것이 좋다.**

- Standard library의 fclose()의 경우 비정상적인 종료에 대해 데이터 손실이 발생함

데이터 읽기: read()

- read() 시스템 호출
  - fd가 나타내는 파일에서
  - nbytes 만큼의 데이터를 읽고
  - 읽은 데이터는 buf에 저장한다.

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

파일 읽기에 성공하면 읽은 바이트 수, 파일 끝을 만나면 0,

실패하면 -1을 리턴

fsize.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define BUFSIZE 512

/* 파일 크기를 계산 한다 */
int main(int argc, char *argv[])
{
    char buffer[BUFSIZE];
    int fd;
    ssize_t nread;
    long total = 0;
    if ((fd = open(argv[1], O_RDONLY)) == -1)
        perror(argv[1]);
```

fsize.c

```
    /* 파일의 끝에 도달할 때까지 반복해서 읽으면서 파일 크기 계산 */
    while( (nread = read(fd, buffer,     BUFSIZE)) > 0)
        total += nread;
    close(fd);
    printf ("%s 파일 크기 : %ld 바이트 ₩n", argv[1], total);
    exit(0);
}
```

데이터 쓰기: write()

- write() 시스템 호출
  - buf에 있는 nbytes 만큼의 데이터를 fd가 나타내는 파일에 쓴다

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

파일에 쓰기를 성공하면 실제 쓰여진 바이트 수를 리턴하고,
실패하면 -1을 리턴

copy.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
/* 파일 복사 프로그램 */
main(int argc, char *argv[])
{
   int fd1, fd2, n;
   char buf[BUFSIZ];
   if (argc != 3) {
      fprintf(stderr,"사용법: %s file1 file2\n",
               argv[0]);
      exit(1);
   }
   if ((fd1 = open(argv[1], O_RDONLY)) ==
    -1) {
      perror(argv[1]);
      exit(2);
   }
   if ((fd2 =open(argv[2], O_WRONLY |
      O_CREAT|O_TRUNC 0644)) == -1) {
      perror(argv[2]);
      exit(3);
   }

   while ((n = read(fd1, buf, BUFSIZ)) > 0)
      write(fd2, buf, n);   // 읽은 내용을 쓴다.
   exit(0);
}
```

- **read/write를 사용할 수 있는 파일의 개방 상태**

함수	파일 개방 상태	
read	O_RDONLY	O_RDWR
write	O_WRONLY	

- **함수 호출의 성공 여부 판단**
  - read 함수
    - 대부분의 경우 세 번째 인수 count로 지정한 값이 반환됨
    - 파일의 마지막 부분을 읽을 경우 count보다 작은 값이 반환됨
    - 반환값이 0일 경우 읽기/쓰기 포인터가 EOF(end-of-file)에 있음
  - write 함수
    - 모든 경우에서 반환값은 세 번째 인수 count로 지정한 값이 반환됨
    - 반환값이 count로 지정한 값이 아닌 경우 쓰기 작업이 실패함

```
01 #include <unistd.h>
02 #include <fcntl.h>
03
04 int main()
05 {
06     int fdin, fdout;
07     ssize_t nread;
08     char buffer[1024];
09
10     fdin = open("temp1.txt", O_RDONLY);
11     fdout = open("temp2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
12     /* 정상적으로 읽어 들인 내용이 1바이트 이상인 동안 반복문 수행 */
13     while((nread = read(fdin, buffer, 1024)) > 0)
14    {
15        /* write가 비정상적으로 수행되었다. (실패) */
16        if(write(fdout, buffer, nread) < nread)
17        {
18            close(fdin);
19            close(fdout);
20        }
21    }
22
23    /* 프로그램이 정상적으로 수행되었다. */
24    close(fdin);
25    close(fdout);
26 }
```

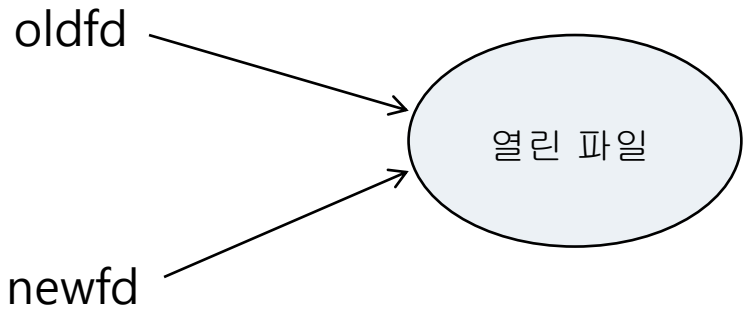
파일 디스크립터 복제

- dup()/dup2() 호출은 기존의 파일 디스크립터를 복제한다.

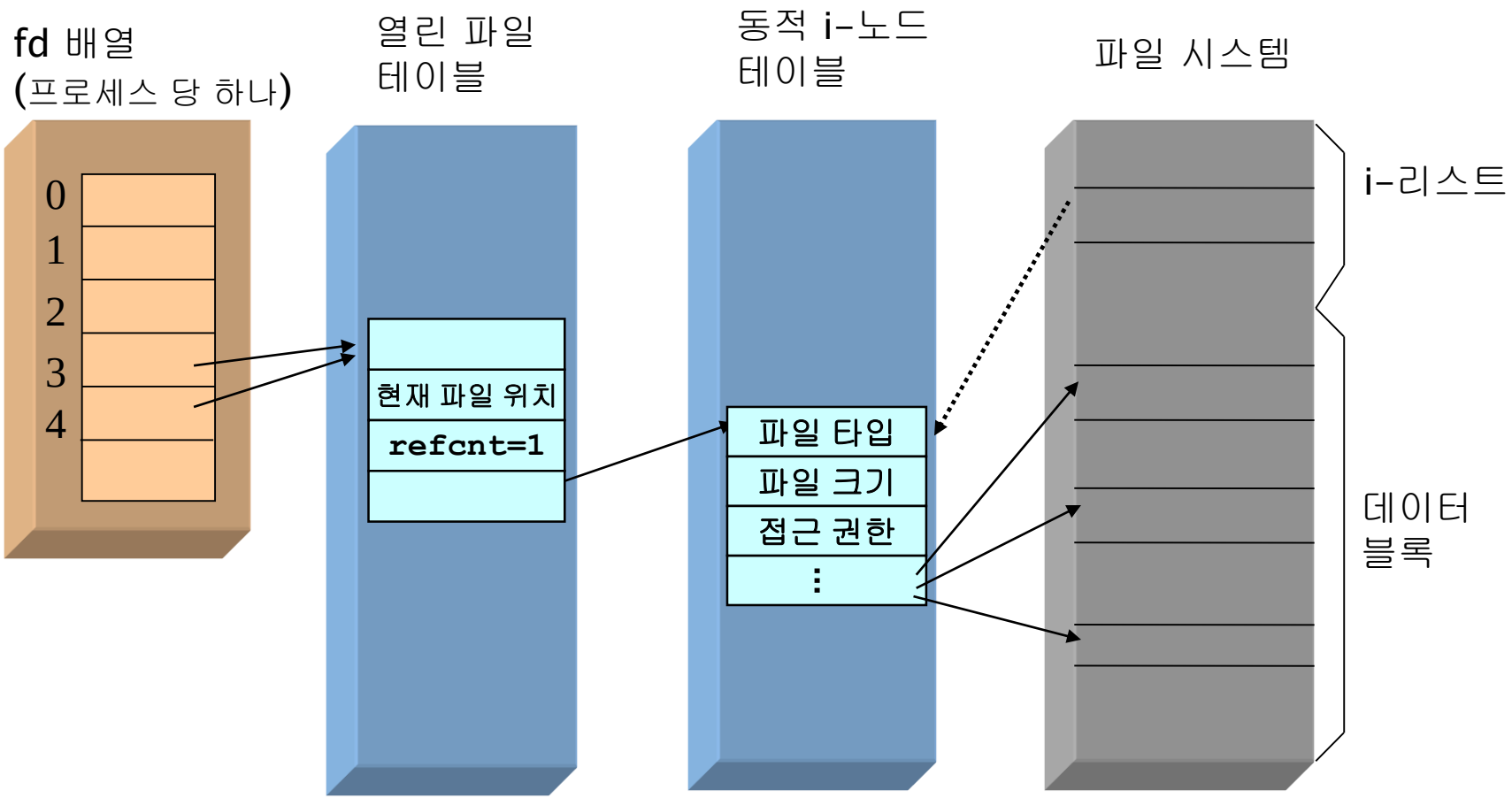
```
#include <unistd.h>
int dup(int oldfd);
oldfd에 대한 복제본인 새로운 파일 디스크립터를 생성하여 반환한다.
실패하면 –1을 반환한다.
int dup2(int oldfd, int newfd);
oldfd을 newfd에 복제하고 복제된 새로운 파일 디스크립터를 반환한다.
실패하면 –1을 반환한다.
```

- oldfd와 복제된 새로운 디스크립터는 하나의 파일을 공유한다.

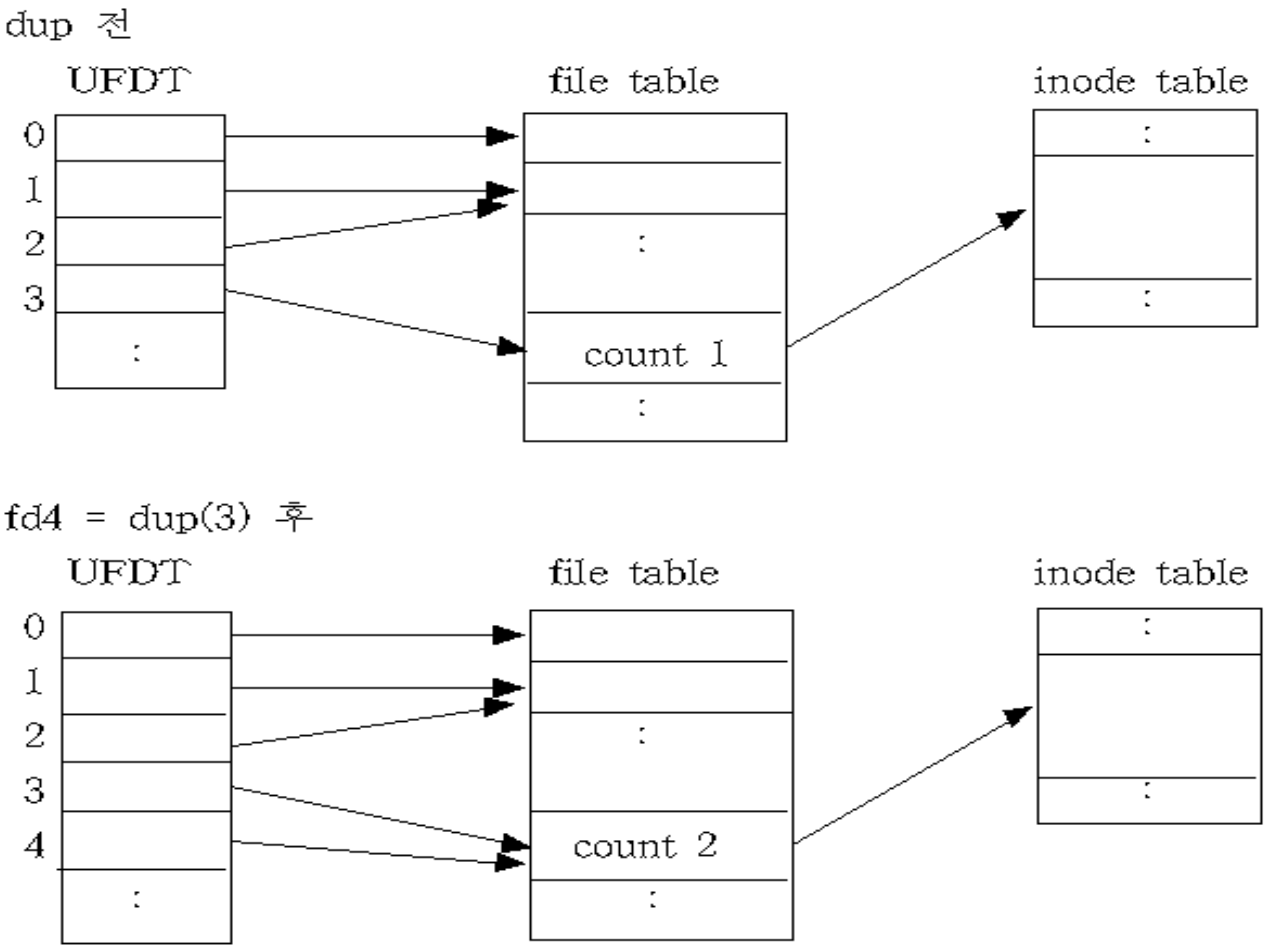
파일 디스크립터 복제



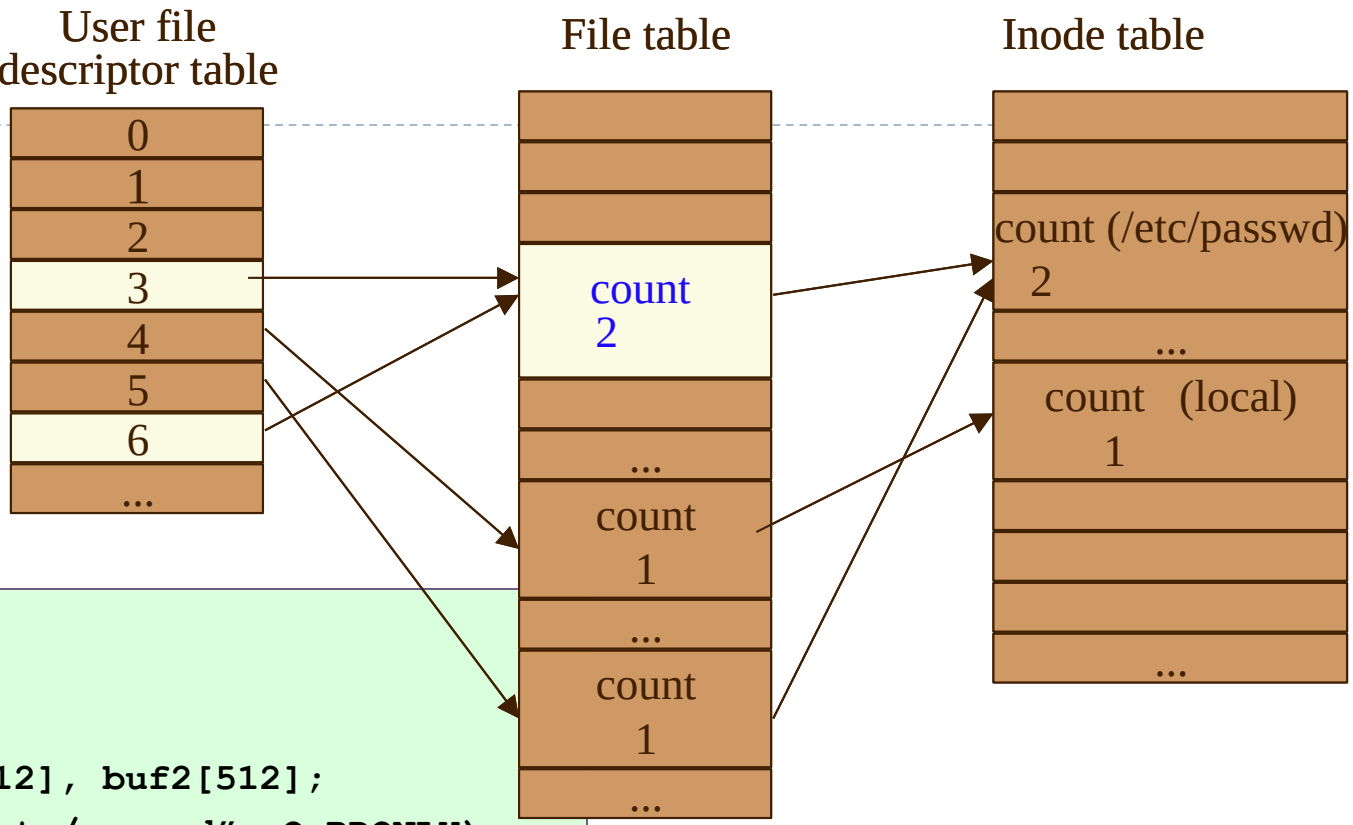
- fd = dup(3); 혹은 fd = dup2(3,4);



dup, dup2


dup 전
UFDT
file table
inode table
0
1
2
3
:
count 1
:
fd4 = dup(3) 후
UFDT
file table
inode table
0
1
2
3
4
:
count 2
:

예)




```
main()
    {
        int i, j;
        char buf1[512], buf2[512];
        i = open("/etc/passwd", O_RDONLY);
        j = dup(i);
        read(i, buf1, sizeof(buf1));
        read(j, buf2, sizeof(buf2));
        close(i);
        read(j, buf2, sizeof(buf2));
    }
```

※ buf1의 데이터와 buf2의 데이터는 같을까?

dup.c

```
1  #include <unistd.h>
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <stdio.h>
6  int main()
7  {
8      int fd1, fd2;
9
10   if((fd1 = creat("myfile", 0600)) == -1)
11         perror("myfile");
12
13    write(fd1, "Hello! Linux", 12);
14    fd2 = dup(fd1);
15    write(fd2, "Bye! Linux", 10);
16    exit(0);
17 }
```

```
$ dup
$ cat myfile
Hello! LinuxBye! Linux
```

예)

- 표준 입출력의 redirection
  - 표준 입출력 대상을 파일로바꿈
  - 표준 입출력의 file descriptor

  - #include <unistd.h>
  - #define STDIN_FILENO 0 /* 표준입력*/
  - #define STDOUT_FILENO 1 /* 표준출력*/
  - #define STDERR_FILENO 2 /* 표준에러*/

예)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
 int main()
{
        int fd;
        if((fd= creat("afile", 0600)) == -1)
                perror("afile");
        printf("This is displayed on the screen.\n");
        dup2(fd, STDOUT_FILENO);
        printf("This is written into the redirected file.\n");
        return 0;
}
```

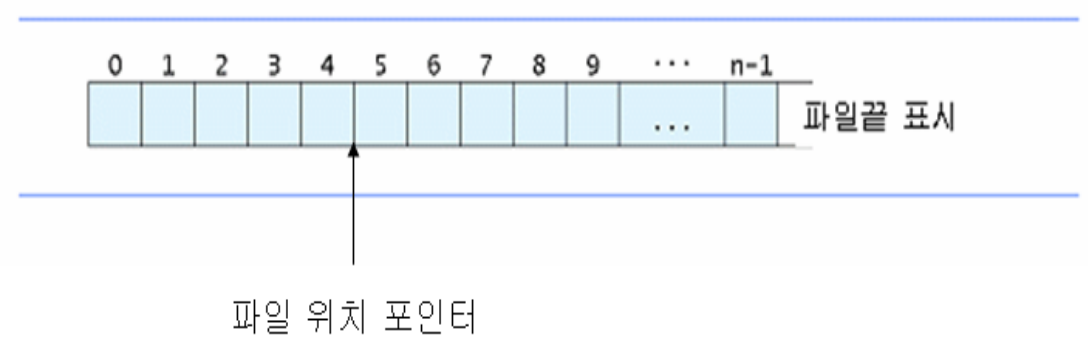
예)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
int main()
{
        int fd;
        if((fd= creat("afile", 0600)) == -1)
                perror("afile");
        printf("This is displayed on the screen.\n");
        close(STDOUT_FILENO);
        dup(fd);
        printf("This is written into the redirected file.\n");
        return 0;
}
```

4.3 임의 접근 파일

파일 위치 포인터(file position pointer)

- 파일 위치 포인터는 파일 내에 읽거나 쓸 위치인 현재 파일 위치(current file position)를 가리킨다.

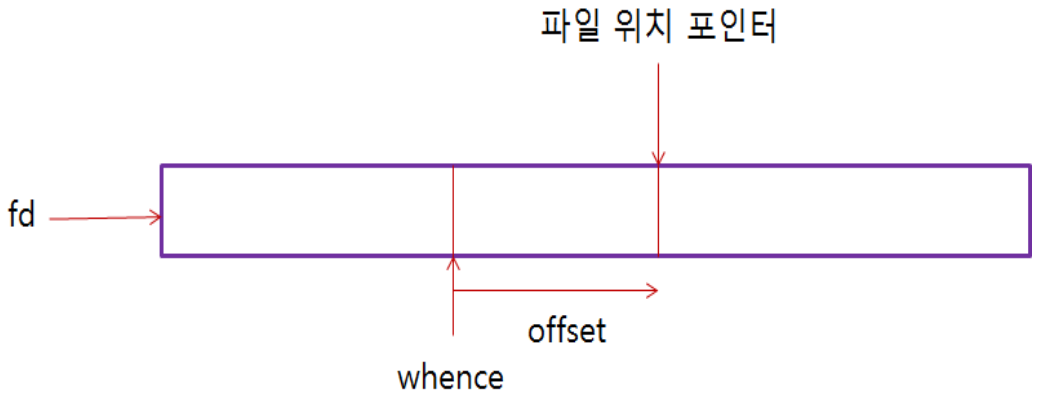


파일 위치 포인터 이동: lseek()

- lseek() 시스템 호출
  - 임의의 위치로 파일 위치 포인터를 이동시킬 수 있다.

```
#include <unistd.h>
off_t lseek (int fd, off_t offset, int whence );
```

이동에 성공하면 현재 위치를 리턴하고 실패하면 -1을 리턴한다.



파일 위치 포인터이동: 예

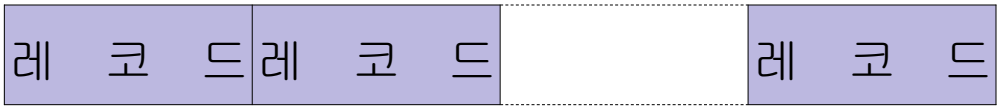
- 파일 위치 이동
  - lseek(fd, 0L, SEEK_SET); 파일 시작으로 이동(rewind)
  - lseek(fd, 100L, SEEK_SET); 파일 시작에서 100바이트 위치로
  - lseek(fd, 0L, SEEK_END); 파일 끝으로 이동(append)

- 레코드 단위로 이동
  - lseek(fd, n * sizeof(record), SEEK_SET); n+1번째 레코드 시작위치로
  - lseek(fd, sizeof(record), SEEK_CUR); 다음 레코드 시작위치로
  - lseek(fd, -sizeof(record), SEEK_CUR); 전 레코드 시작위치로 .

- 파일끝 이후로 이동
  - lseek(fd, sizeof(record), SEEK_END); 파일끝에서 한 레코드 다음 위치로

레코드 저장 예

```
write(fd, &record1, sizeof(record));
write(fd, &record2, sizeof(record));
lseek(fd, sizeof(record), SEEK_END);
write(fd, &record3, sizeof(record));
```



unlink, remove

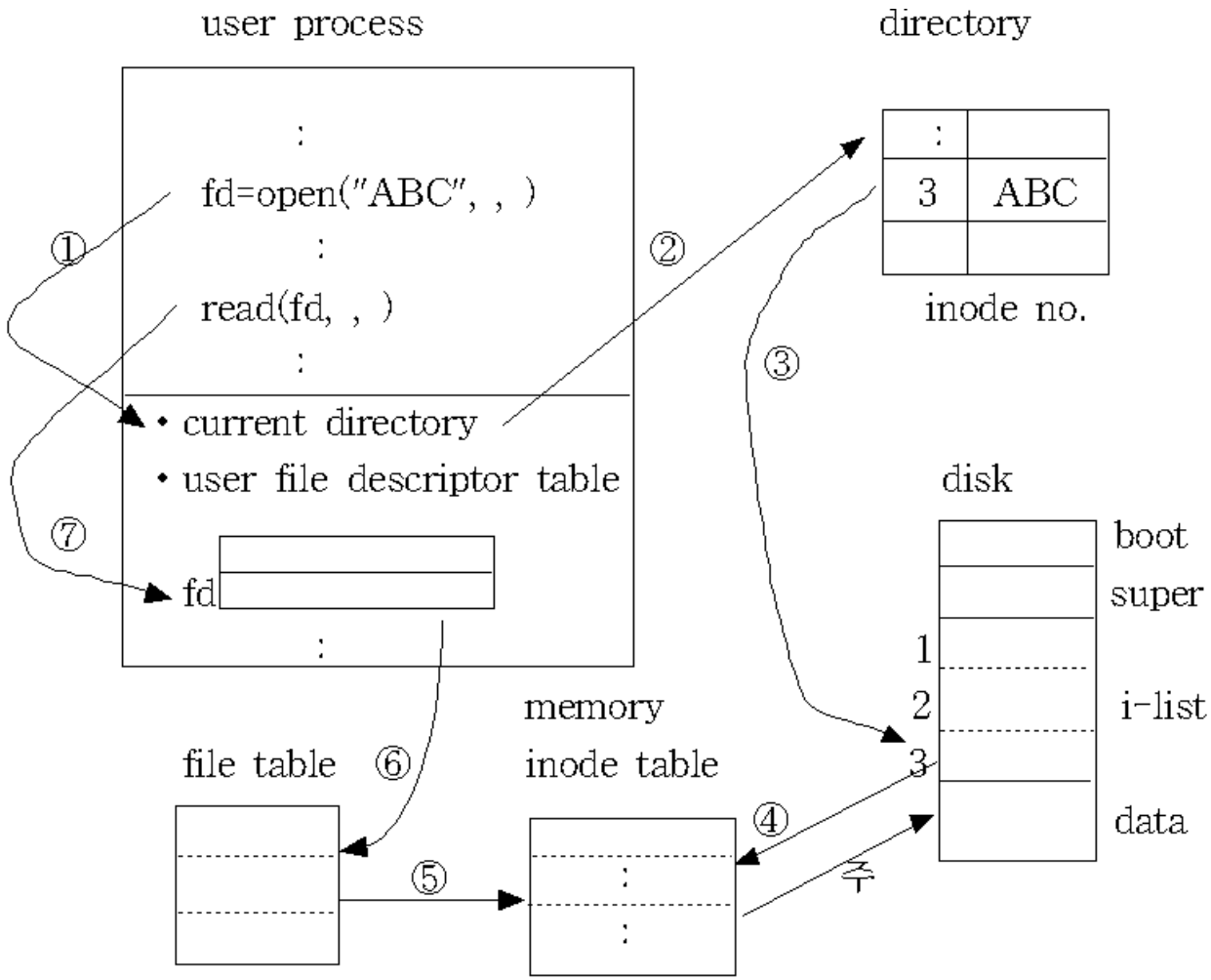
- **경로명으로 지정한 파일을 삭제한다**

```
#include <unistd.h>
int unlink(const char *pathname);
...
#include <stdio.h>
int remove(const char *pathname);
```

pathname	삭제할 파일의 경로 이름이다.
반환값	작업이 성공할 경우 0이 반환되며, 실패할 경우 -1이 반환된다.

- **pathname으로 지정한 파일을 삭제한다.**
- **비어 있는 디렉터리는 remove만 삭제할 수 있다. (unlink는 불가능)**
  - **↔ 비어 있지 않은 디렉터리는 둘 모두 삭제할 수 없다.**

File open 절차


user process
:
fd=open("ABC", ,)
:
read(fd, ,)
:
• current directory
• user file descriptor table
fd
:
①
②
③
④
⑤
⑥
⑦
directory
:
3
ABC
inode no.
disk
boot
super
1
2
3
i-list
data
memory
inode table
file table
:
:
주

File open 절차

①사용자 프로세스에서 open("ABC", ,) 시스템 호출

② current directory file에서 "ABC"란 파일의 inode no.가 3이라는 것을 알아냄

③ inode 3에 해당하는 블록이 in-core inode에 없으면 디스크로 가서 i-list의 세 번째 항목을 찾음

④ 디스크의 inode 내용을 in-core inode로 복사

⑤ inode에 대한 file table entry를 세팅

⑥ user file descriptor table에 해당 파일 테이블 엔트리에 대한 user file descriptor table entry 세팅

user file descriptor를 리턴함으로서 open 시스템 호출 완료

⑦ read 시스템 호출은 이미 확립된 UFDT→file table→inode table→디스크 순으로

Close 절차

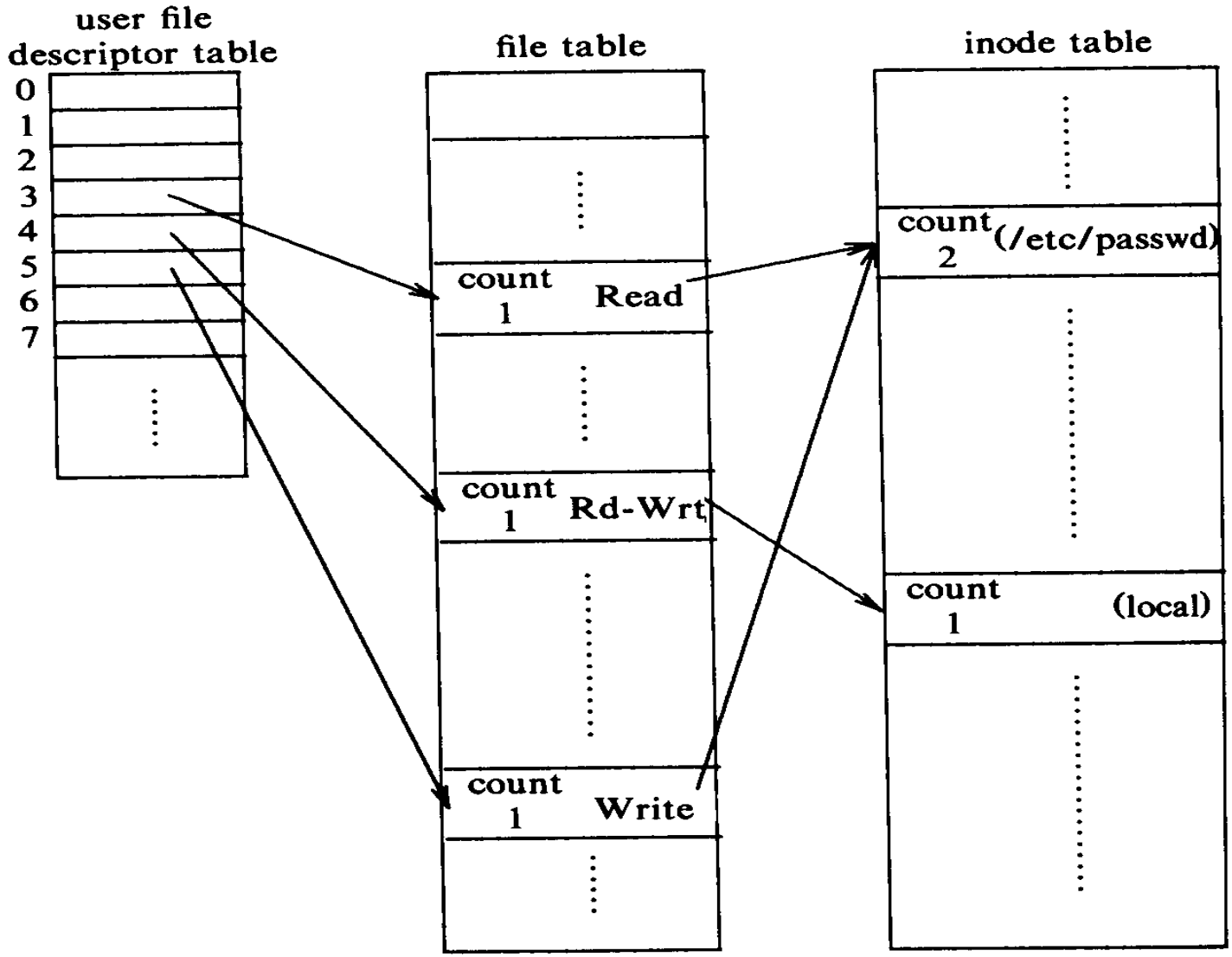
- File descriptor, 해당 file table 항 및 inode table 항을 차례로 반납
  - 만일 file table 항의 reference count가 1보다 크면 단지 count만 감소시키고 close를 끝낸다.
  - 만일 count가 1이면 entry를 비우고 반납함
  - 만일 다른 process가 아직 그 inode를 참조하고 있으면 inode의 reference count만 감소
  - inode reference count가 0인 경우에만 inode를 반납
  - 시스템 호출이 완료되면 user file descriptor table의 항은 empty가 된다.

- Process가 exit할 때 kernel은 그 process의 user file descriptor table을 보고 아직 empty가 아닌 항을 전부 close

파일 관련 테이블간의 관계(예)

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_WRONLY);
fd3 = open("/etc/passwd", O_RDWR);
```

- /etc/passwd에 대한 read는 fd1을 통하여 하며, write는 fd3을 통하여 한다.
- 같은 이름의 파일이라도 다른 fd와 다른 file table entry가 할당되며, inode는 같음.

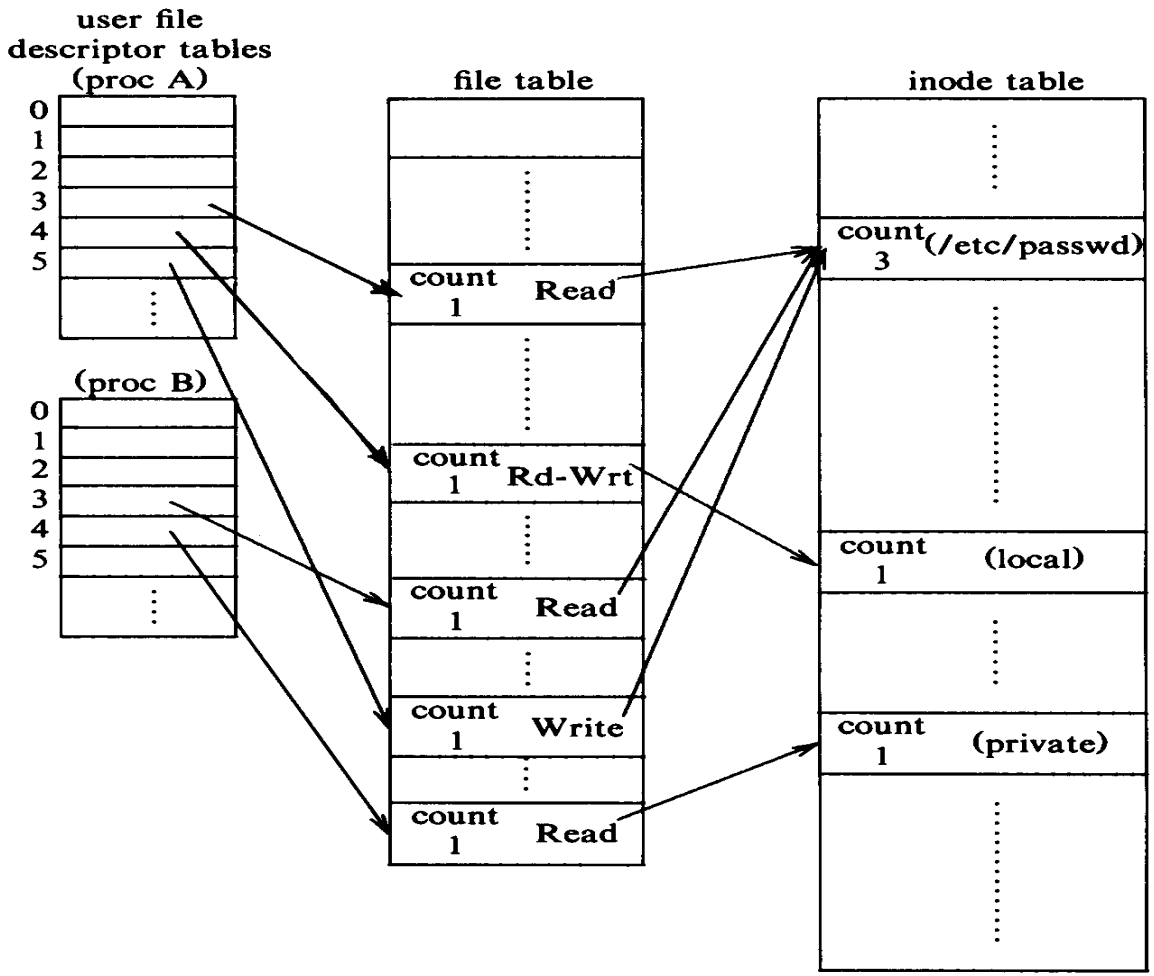

user file
descriptor table
0
1
2
3
4
5
6
7
file table
count
1
Read
count
1
Rd-Wrt
count
1
Write
inode table
count
2
(/etc/passwd)
count
1
(local)

Process 1

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_WRONLY);
fd3 = open("/etc/passwd", O_RDWR);
```

Process 2

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```


user file
descriptor tables
(proc A)
0
1
2
3
4
5
(proc B)
0
1
2
3
4
5
file table
count
1
Read
count
1
Rd-Wrt
count
1
Read
count
1
Write
count
1
Read
inode table
count
3
(/etc/passwd)
count
1
(local)
count
1
(private)