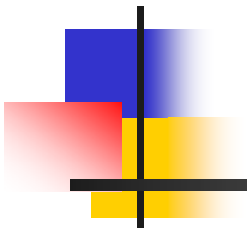
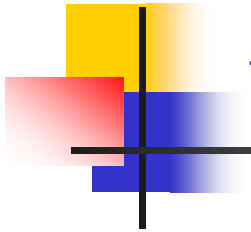
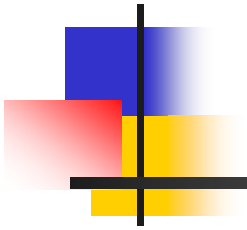


제2장. 파일 입출력 (File I/O)



목표

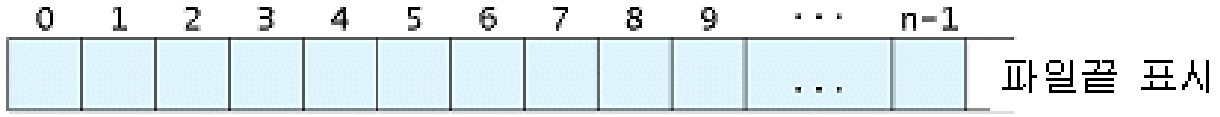
- 파일의 구조 및 특성을 이해한다.
- 파일을 열고 닫는다.
- 파일로부터 데이터를 읽고 쓴다.
- 파일 현재 위치 변경
- 기타 파일 제어

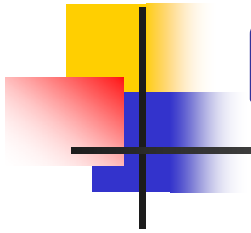


2.1 파일 구조

What is a file ?

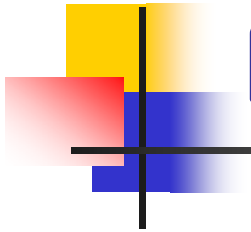
- a file is a contiguous sequence of bytes
- no format imposed by the operating system
- each byte is individually addressable in a disk file
- a file is also a uniform interface to external devices





File Descriptor

- **File descriptor**
  - open() returns a fd, an integer value
  - 열린 파일을 나타내는 번호
  - used in subsequent I/O operations on that file
  - close(fd) closes that file described by fd
  - all of a process's open files are automatically closed when it terminates

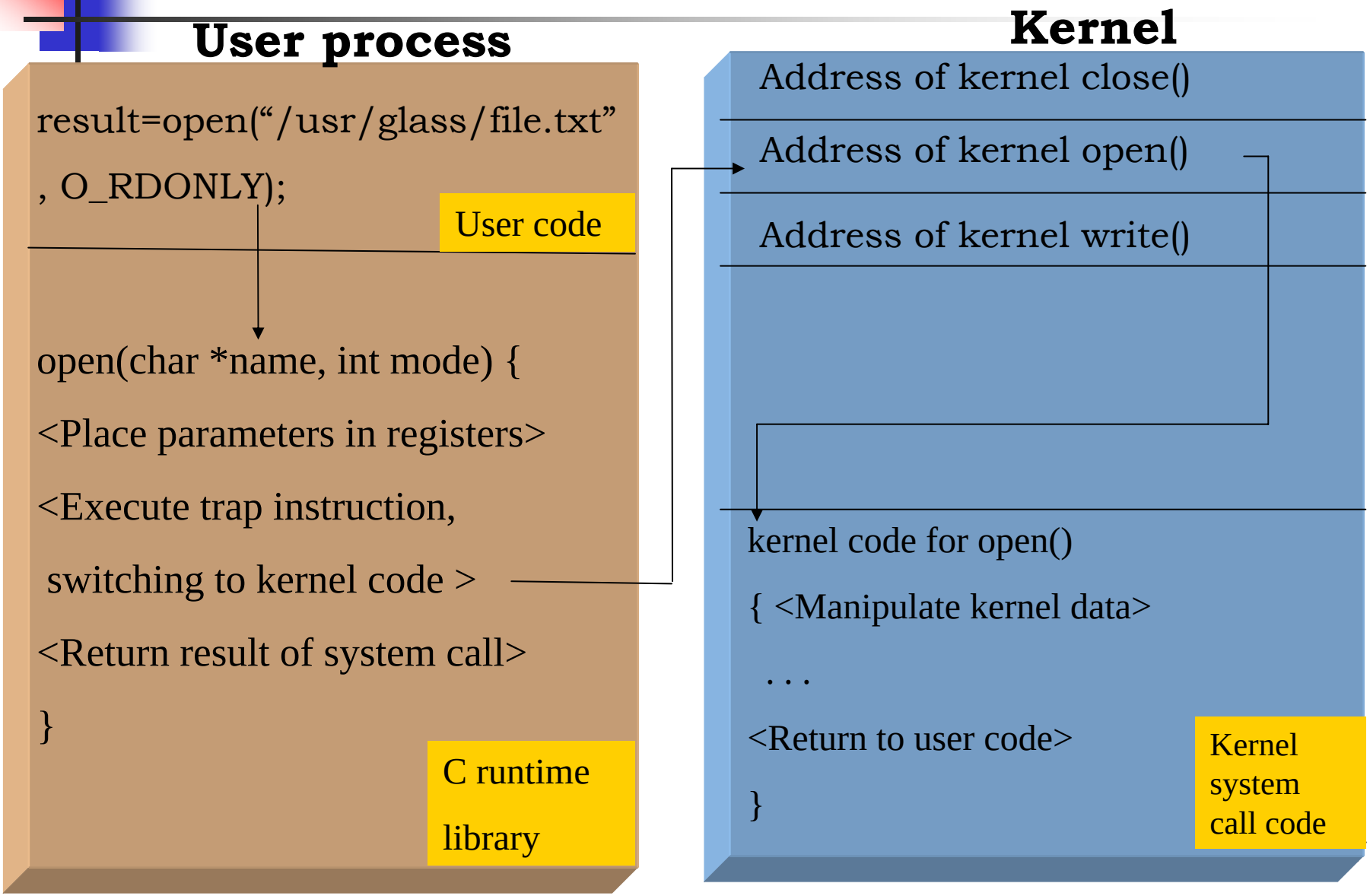


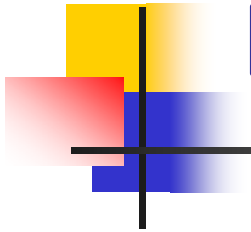
File Descriptor

- **file descriptor : 0 ~ 19**

Value	Meaning
0	standard input
1	standard output
2	standard error
3 .. 19	fds for users

User and Kernel Mode





Inode (Index node)

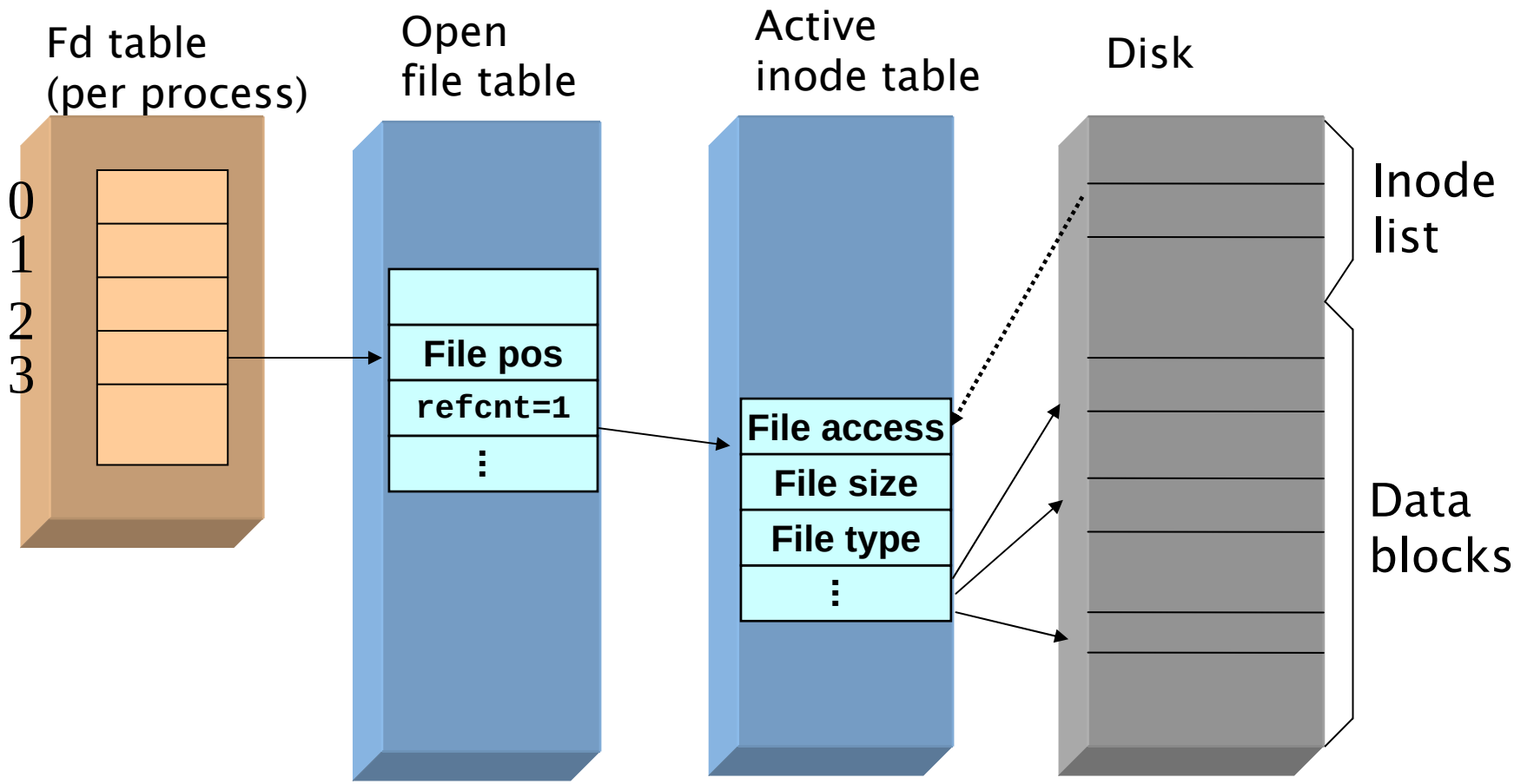
- 한 파일은 하나의 i-node를 갖는다.

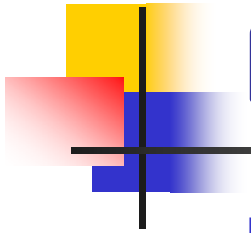
- 파일에 대한 모든 정보를 가지고 있음
  - file type : regular, directory, block special, character special, etc
  - file size
  - file permissions
  - the owner and group ids
  - the last modification and last access times

  - if it's a regular or directory, the location of data blocks
  - if it's a special file, device numbers
  - if it's a symbolic link, the value of the symbolic link

파일을 위한 커널 자료 구조

- fd= open("file", O_RDONLY);

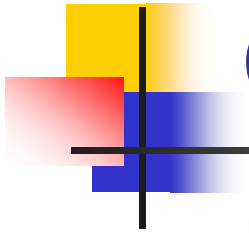




Process table entry

- 프로세스 테이블 (Process table)
  - 커널(kernel) 자료구조
  - 프로세스 목록
  - 프로세스 → 프로세스 테이블 항목

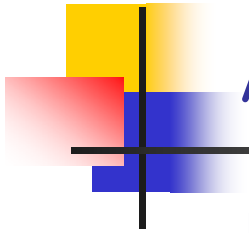
- 프로세스 테이블 항목 (Process table entry)
  - 파일 디스크립터 배열(file descriptor array) 포함
  - fd array



Open File Table

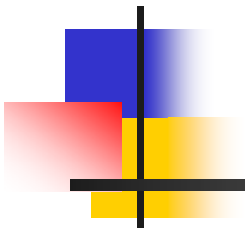
- 파일 테이블 (file table)
  - 커널 자료구조
  - 열려진 모든 파일 목록
  - 열려진 파일 → 파일 테이블의 항목

- 파일 테이블 항목 (file table entry)
  - 파일 상태 플래그
    (read, write, append, sync, nonblocking,…)
  - 파일의 현재 위치 (current file offset)
  - i-node에 대한 포인터



Active i-node table

- Active i-node table
  - 커널 내의 자료 구조
  - Open 된 파일들의 i-node를 저장하는 테이블
- i-node
  - 하드 디스크에 저장되어 있는 파일에 대한 자료구조
  - 한 파일에 하나의 i-node
  - 하나의 파일에 대한 정보 저장
    - 소유자, 크기
    - 파일이 위치한 장치
    - 파일 내용 디스크 블럭에 대한 포인터
- i-node table vs. i-node



2.2 파일 관련 시스템 호출

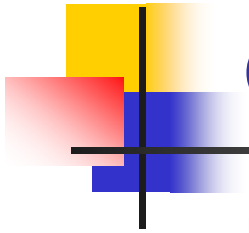
- open() – 열기
- creat() – 파일 생성
- close() – 닫기
- read() – 읽기
- write() – 쓰기
- lseek() – 이동

open() – 열기

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *pathname, int oflag, [ mode_t  mode ]);
```

- 파일을 연다
- 파일이 없으면 경우에 따라 새로 만들어 질 수도 있다

- 리턴 값 : 파일 디스크립터(file descriptor), 실패하면 -1
- *pathname* : 파일의 이름
- *mode* : 파일의 access permission 값. 생략가능.
  새로운 파일을 만드는 경우에만 사용됨
  creat() 함수 설명 참조

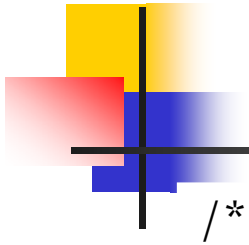


open() 의 파라미터

- 두 번째 파라미터 *oflag*는 다음 상수들의 OR 이다
  - 예
    ```
    int fd;
    fd = open("afile", O_RDWR | O_CREAT, 0600 );
    ```
- 반드시 하나 지정해주어야 할 값
  - O_RDONLY : 읽기 모드, `write` 함수를 사용 할 수 없음
  - O_WRONLY : 쓰기 모드, `read` 함수를 사용 할 수 없음
  - O_RDWR : 읽고 쓰기 모드, `read write` 사용 가능

open() 의 파라미터

- 선택적 지정 가능
  - O_APPEND: 모든 `write` 된 데이터는 파일의 끝에 추가
  - O_CREAT : 파일이 없는 경우 파일 생성 세 번째 인자 mode
  - O_EXCL : O_CREAT이고 그 파일이 이미 있으면 에러
  - O_TRUNC : 파일이 있는 경우 파일 크기를 0으로 만든다
  - O_NONBLOCK : nonblocking 모드로 입출력을 함
  - O_SYNC : 각 `write` 함수 호출은
    디스크에 물리적으로 쓰여진 후 리턴한다



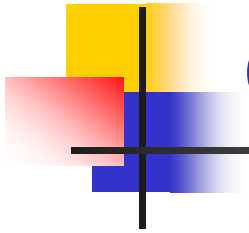
예제: open.c

```
/* open.c */
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
  int fd;
  char fname[] = "afile";

  if ((fd = open (fname, O_RDWR)) == -1)
    perror(fname);

  printf("%s is opened.\n", fname);
  close(fd);
  return 0;
}
```

creat () – 파일 생성

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat ( const char *pathname, mode_t mode );
```

- 새로운 파일을 생성한다
- 리턴 값 : 파일 디스크립터, 실패하면 -1
- pathname : 생성하고자 하는 파일의 이름
- mode : 파일의 access permission 값



creat () 의 파라미터

- 두 함수 호출은 동일
  - fd = creat (pathname, mode);
  - fd = open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
- 두 번째 인자 mode는 permission mode
  - 예
    - 0644
    - 0777
    - 0444

close () – 닫기

```
#include <unistd.h>

int close ( int fd );
```

- 작업이 끝난 후 파일을 닫는다.

- 리턴 값 : 성공하면 0, 실패하면 -1
- fd : 닫고자 하는 파일의 파일 디스크립터

- 프로세스가 종료되면 모든 열려진 화일들은 자동적으로 닫힌다

read () – 읽기

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

- *fd*가 나타내는 파일에서 데이터를 읽는다
- 리턴 값:
  - 성공하면 읽은 바이트 수
  - 파일의 끝을 만나면 0
  - 실패하면 -1
- *buf*
  - 읽은 데이터를 저장할 메모리의 시작 주소
- *nbytes*
  - 읽을 데이터의 바이트 수

read () – 읽기

- 읽을 데이터가 충분하면 한 번에 *nbytes* 만큼 읽는다.
- 읽을 데이터가 *nbytes* 보다 적으면 더 적게 읽는다.
  - 파일의 끝에서
  - 네트웍 입출력에서

- size_t : unsigned integer
- ssize_t : signed integer

예제 /* count.c: 파일의 문자 수를 센다*/

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

int main()
{
  char buffer[BUFSIZE];
  int fd;
  ssize_t nread;
  long total = 0;

  if ((fd = open("afile", O_RDONLY)) == -1)   perror("afile");

  /* 파일의 끝에 도달할 때까지 반복 */
  while( (nread = read(fd, buffer, BUFSIZE)) > 0)
    total += nread;

  close(fd);
  printf ("total chars in afile: %ld\n", total);
  return 0;
}
```

write() – 쓰기

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

- *fd*가 나타내는 파일에 데이터를 쓴다
- 리턴 값
  - 성공하면, 파일에 쓰여진 데이터의 바이트 수
  - 실패하면, -1
- *buf*
  - 쓸 데이터를 저장하고 있는 메모리의 시작주소
- *nbytes*
  - 쓸 데이터의 바이트의 수

예: Copy Files

```
#include <stdio.h>
#include <fcntl.h>
main(argc, argv)
int argc;     char *argv[ ];
{
     int fdin, fdout, n;
     char buf[BUFSIZE];
    if (argc != 3) {
          fprintf(stderr,"Usage: %s filein fileout\n",argv[0]);      exit(1); }
     if ((fdin = open(argv[1], O_RDONLY)) == -1) {
          perror(argv[1]);  exit(2); }
     if ((fdout=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC, 0644))==-1){
          perror(argv[2]);  exit(3);}

     while ((n = read(fdin, buf, BUFSIZE)) > 0)
          write(fdout, buf, n);
     exit(0);
}
```

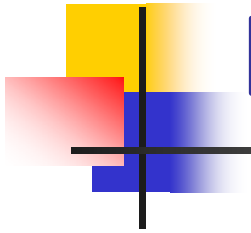
lseek() – 이동

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

- 파일의 현재 위치(current file offset)를 이동
  - 성공하면 현재 위치를 리턴, 실패하면 -1

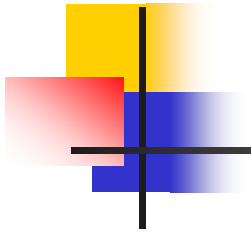
- *whence* : 위치 기준점
  - SEEK_SET : 파일의 시작점을 기준으로 이동
  - SEEK_CUR : 현재 위치를 기준으로 이동
  - SEEK_END : 파일의 끝을 기준으로 이동

- *offset* : 기준점에서의 상대적인 거리 (byte 단위)
  - SEEK_CUR, SEEK_END 와 같이 쓰일 때는 음수도 가능



lseek() – 이동

- 파일의 현재 위치(current file offset)
  - 파일을 처음 열면 현재 위치는 0 즉 파일의 시작
  - 파일에 대한 읽기/쓰기는 파일의 현재 위치에서 실행된다
  - 읽기/쓰기 후 파일의 현재 위치는 읽기/쓰기 한 byte 수만큼 자동적으로 뒤로 이동
- lseek()
  - 임의의 위치로 파일의 현재 위치를 이동할 수 있다



예 :lseek

- rewind

  lseek(fd, 0L, 0);

- append

  lseek(fd, 0L, 2);

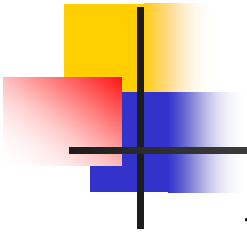
- record position

  loc = lseek(fd, 0L, 1);

- increase file

  lseek(fd, (long) MAX*sizeof(record), 2);

  write(fd, (char *) &reocrd, sizeof(record));

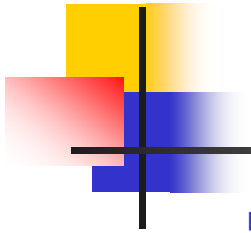


예제: /* lseek.c */

```
#include <unistd.h>

int main()
{
  if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
    printf("cannot seek\n");
  else
    printf("seek OK\n");
  return 0;
}
```

예제: /* lseek1.c */

```
#include <unistd.h>  /* lseek1.c */
#include <fcntl.h>

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

int main() {
  int fd;

  if ( (fd = creat("file.hole", 0644)) < 0)
    perror("file.hole");

  if (write(fd, buf1, 10) != 10) /* offset now = 10 */
    perror("buf1");

  if (lseek(fd, 40, SEEK_SET) == -1) /* offset now = 40 */
    perror("lseek");

  if (write(fd, buf2, 10) != 10) /* offset now = 50 */
    perror("buf2");
  return 0;
}
```



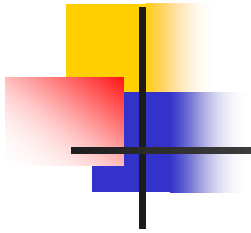
예제: /* lseek1.c */

- lseek1.c 의 출력 파일 file.hole의 내용

	0	1	2	3	4	5	6	7	8	9
0	a	b	c	d	e	f	g	h	i	j
10	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
20	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
30	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
40	A	B	C	D	E	F	G	H	I	J

예제: /* lseek2.c */

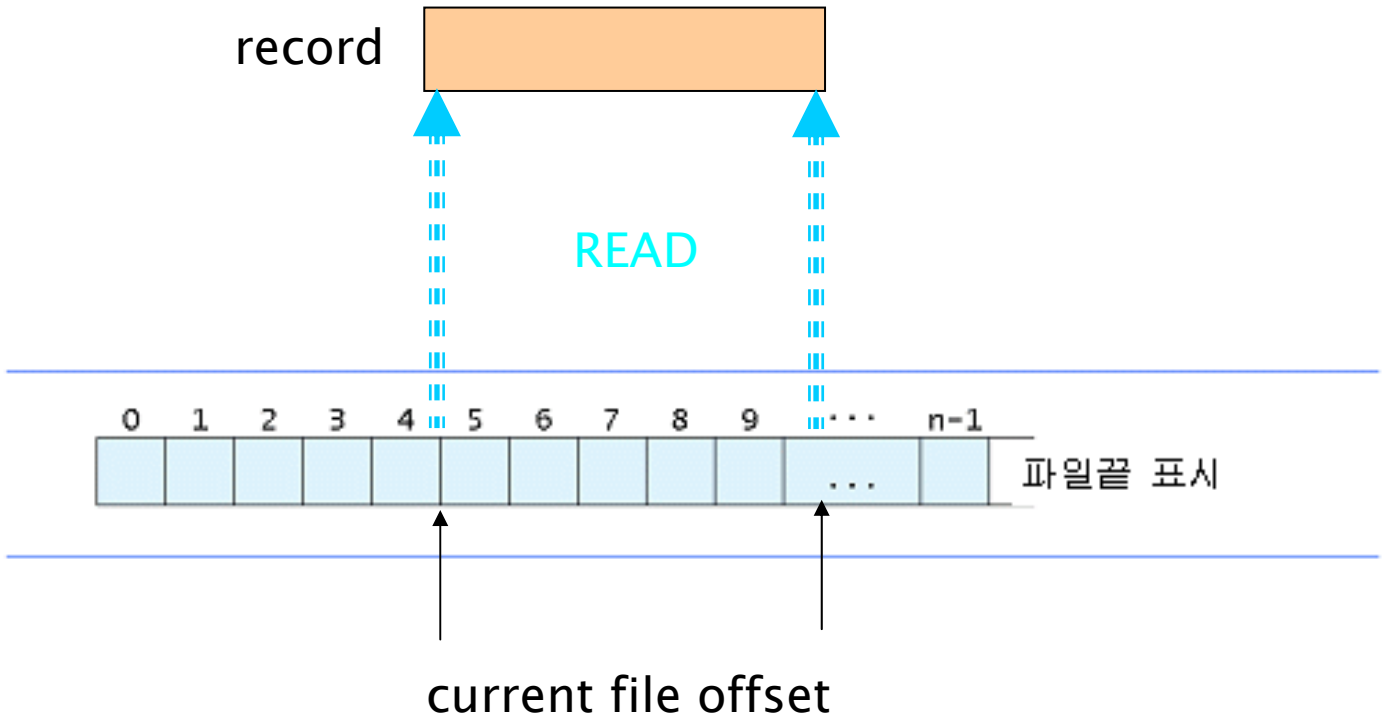
```
#include <sys/types.h>  /* lseek2.c */
#include <unistd.h>
#include <fcntl.h>

int main()
{
  int fd;
  off_t fsize;
  char buf[11];

  if((fd=open("file.hole", O_RDONLY)) < 0)
    perror("file.hole");

  fsize = lseek(fd, 0, SEEK_END);
  printf("size: %lu\n", fsize);

  lseek(fd, 40, SEEK_SET);
  read(fd, buf, 10);
  buf[10] = '\0';
  puts(buf);
  return 0;
}
```



레코드 수정

```
struct XXX record;
        …
read(fd, (char *) &record, sizeof(record));
… update record …
lseek(fd, (long) -sizeof(record), 1);
 write(fd, (char *) &record, sizeof(record));
```

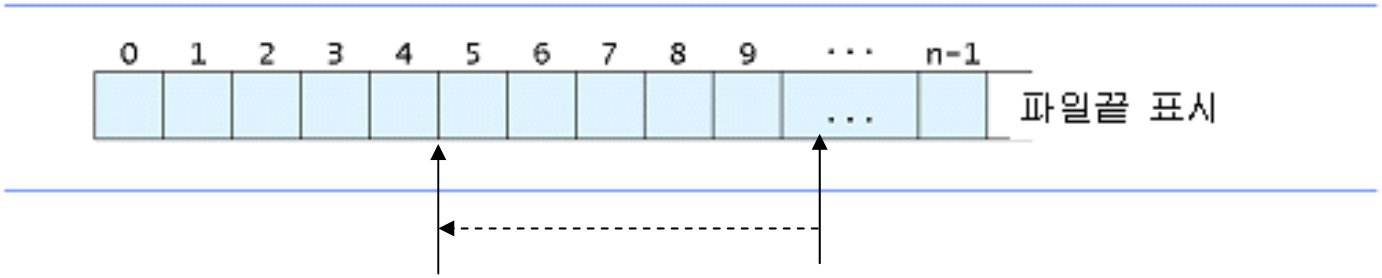
레코드 수정



레코드 수정

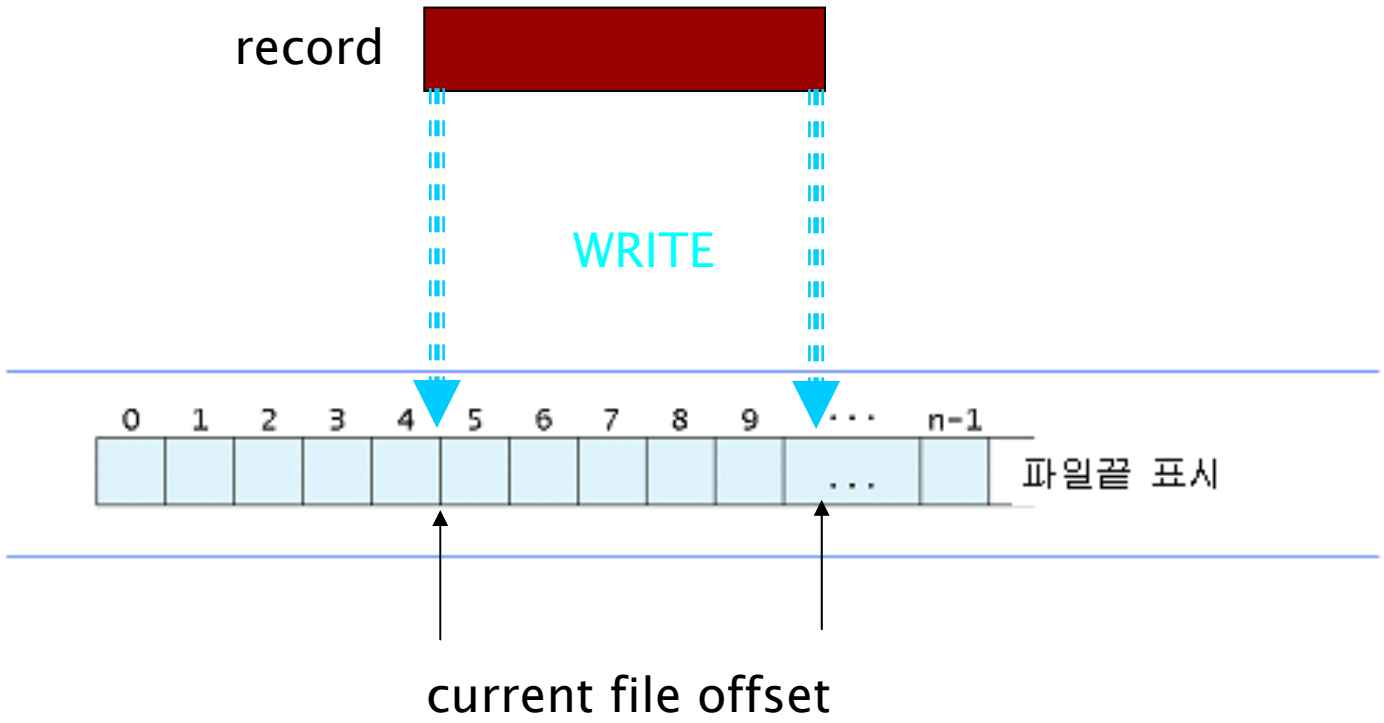
record 

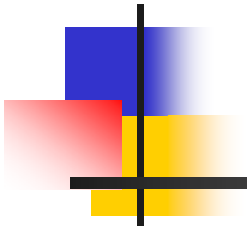
LSEEK



current file offset

레코드 수정

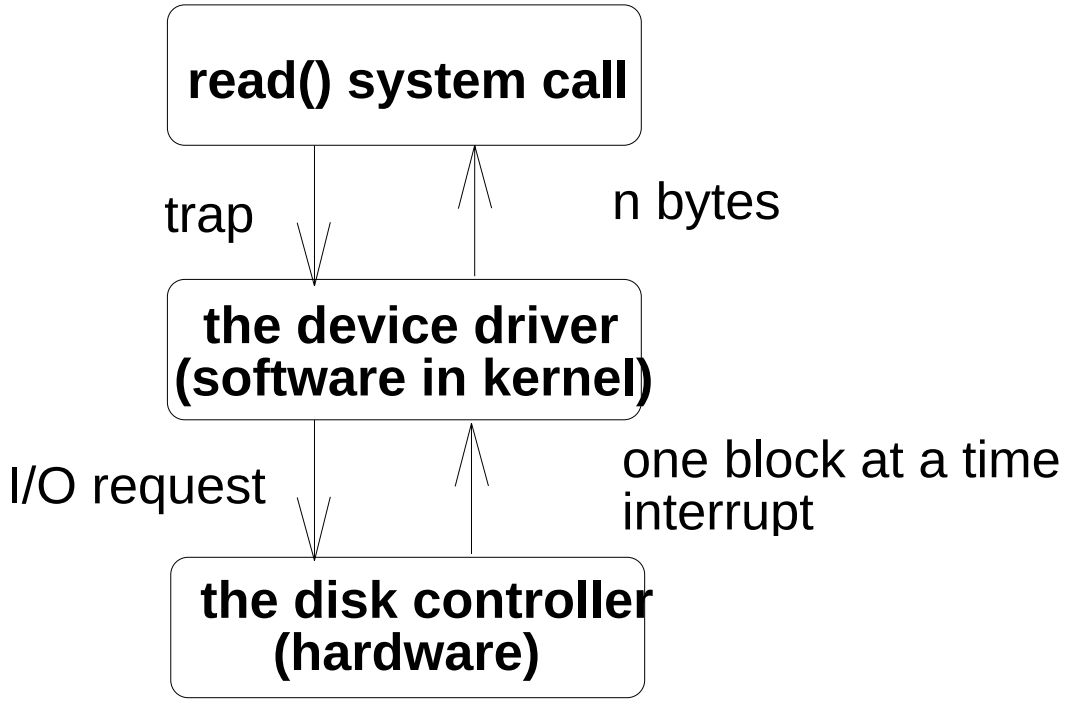




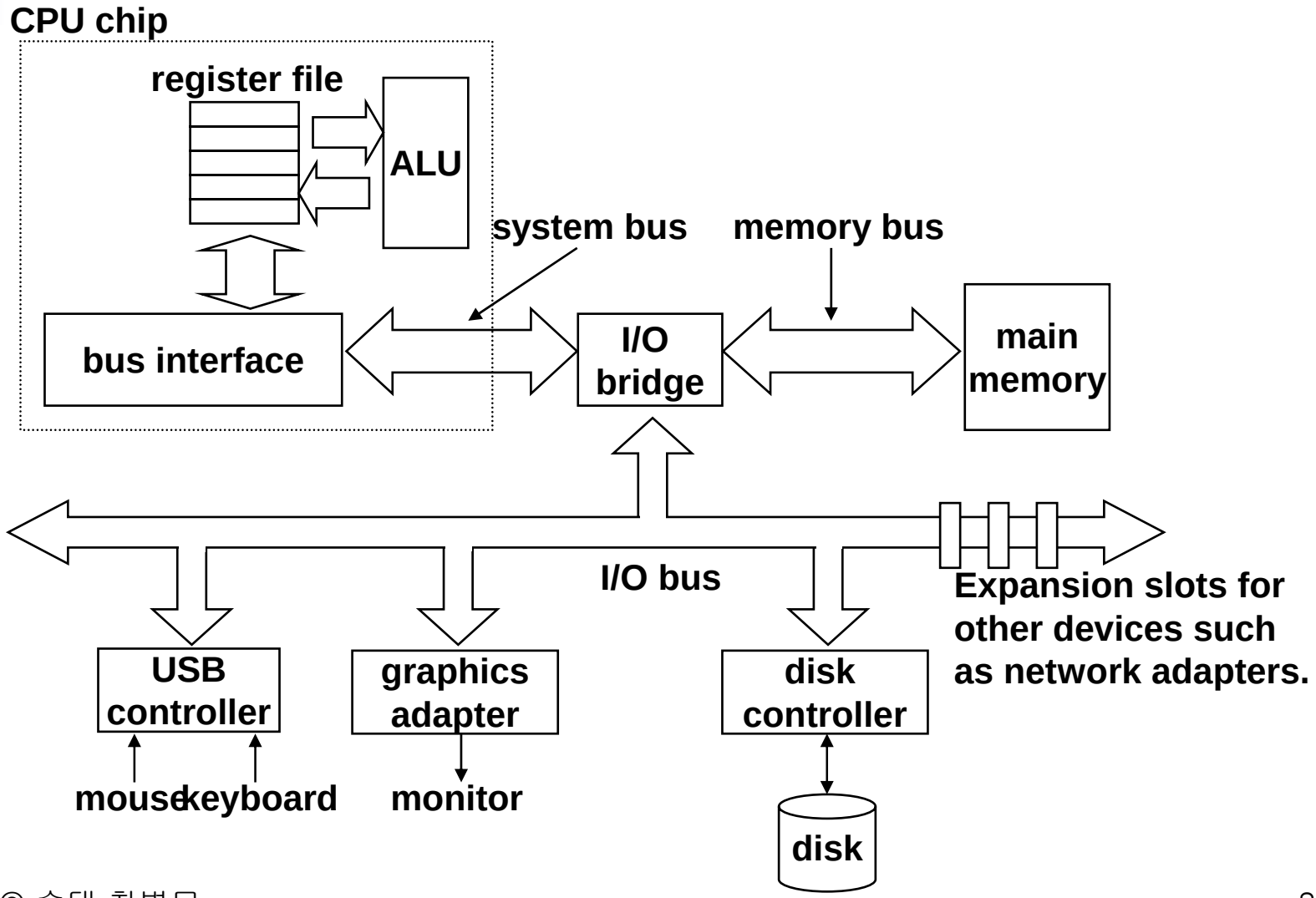
2.3 시스템 호출 구현

Block I/O

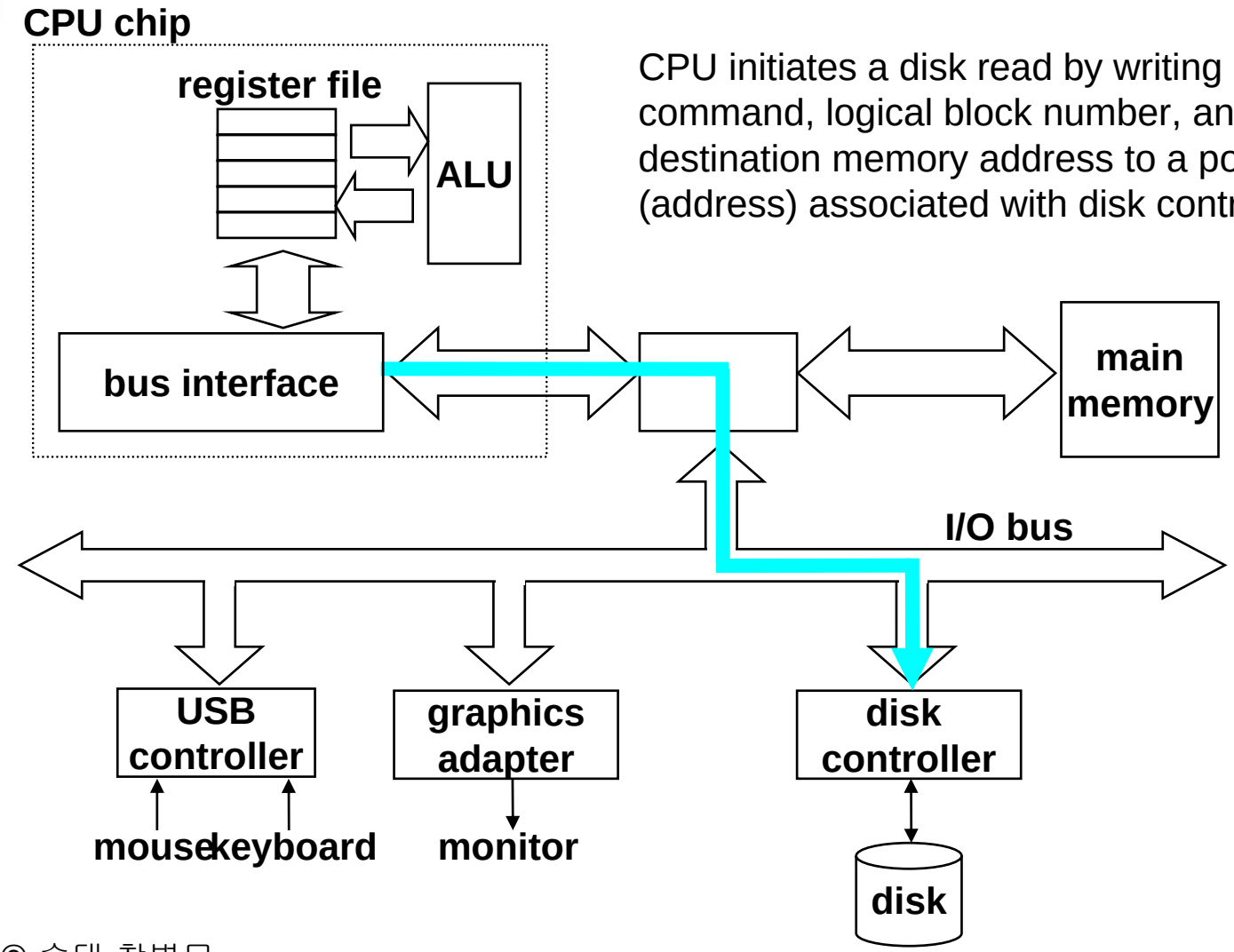
- **I/O is always done in terms of blocks**
- **Sequence of a read() system call**



A Typical Hardware System

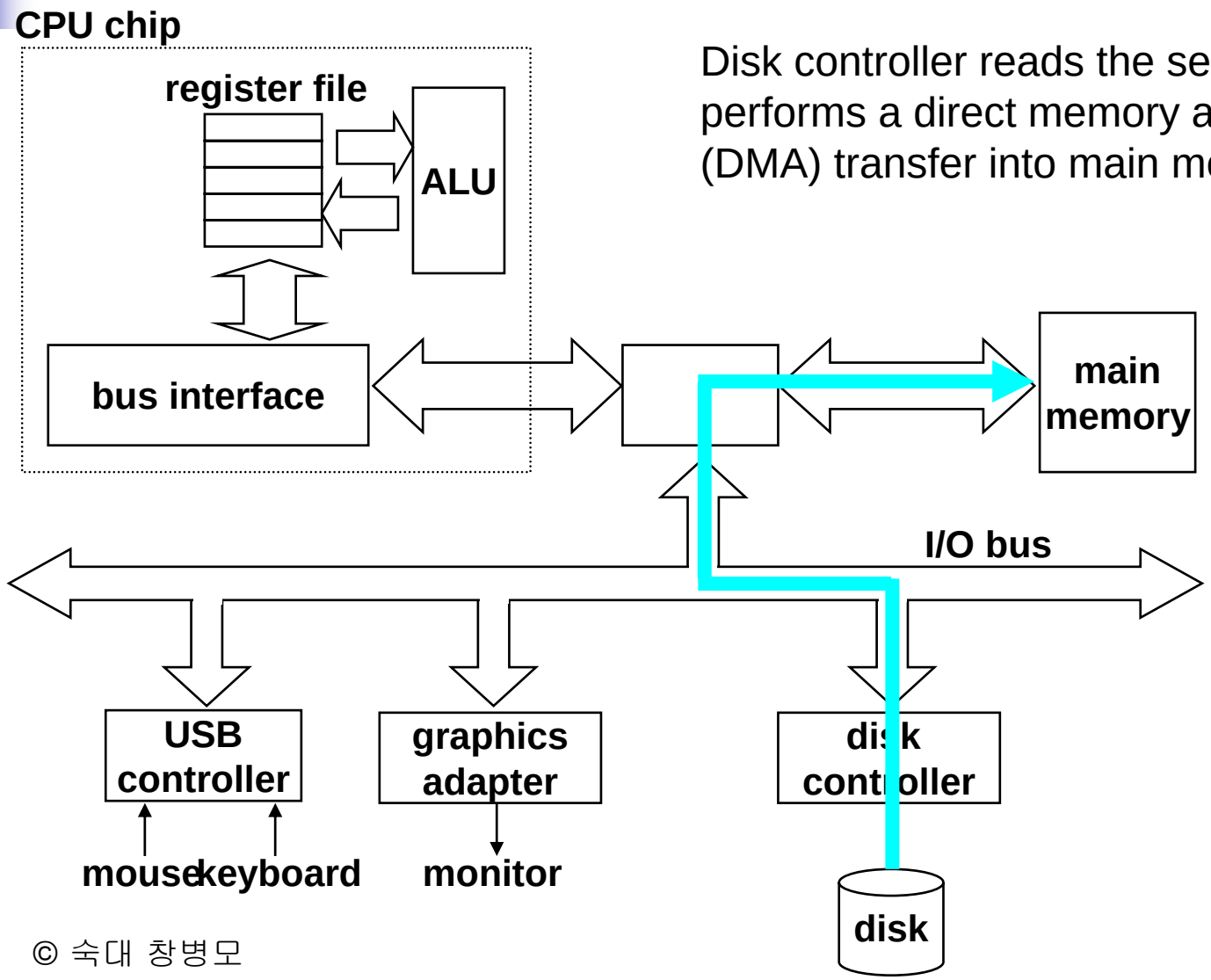


Reading a Disk Sector: Step 1



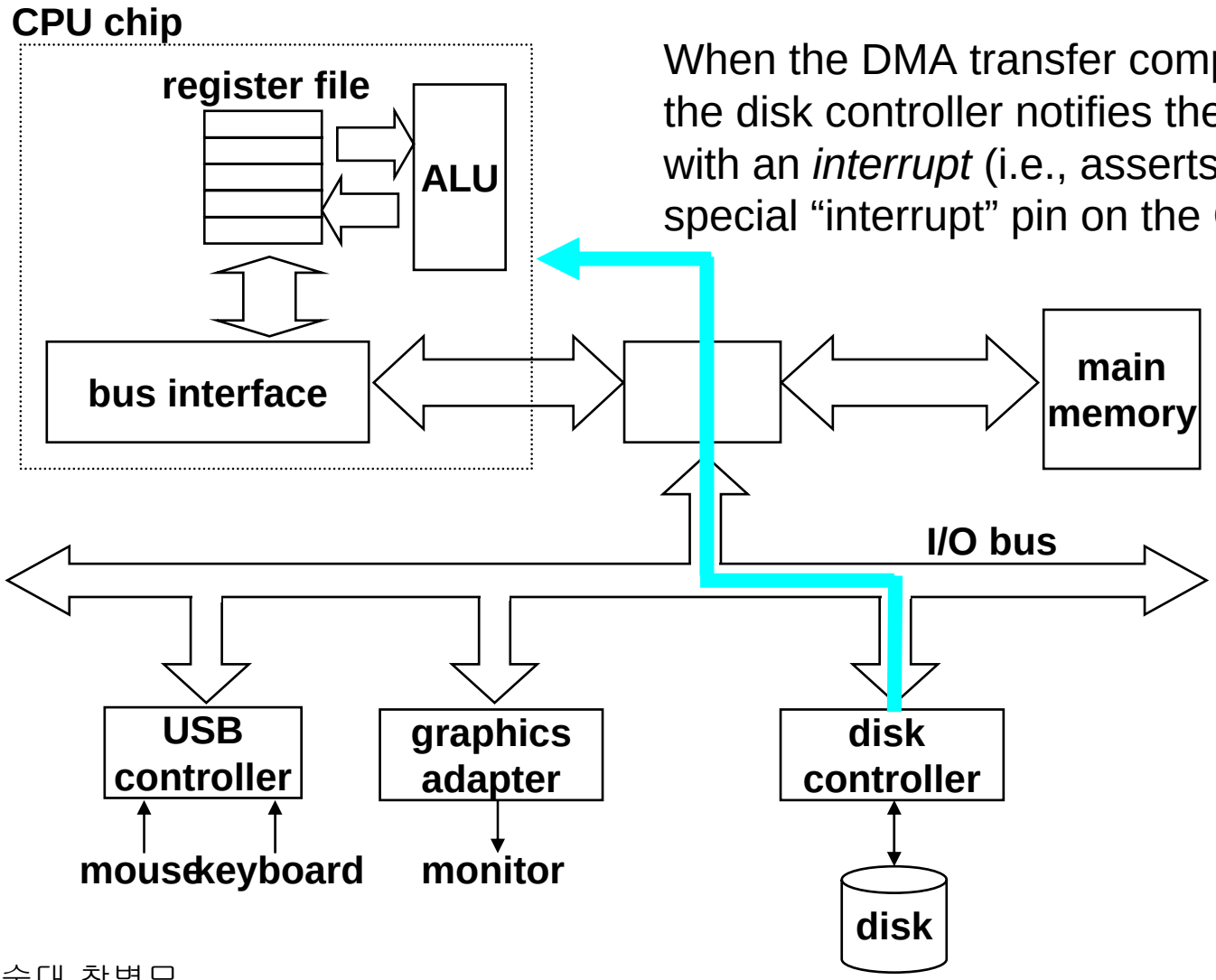
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Reading a Disk Sector: Step 2

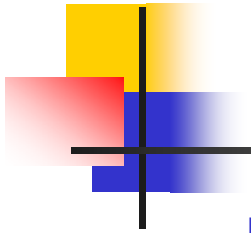


Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector: Step 3

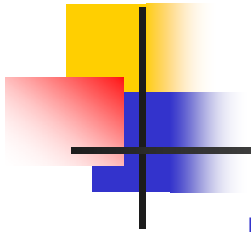


When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special “interrupt” pin on the CPU)



I/O 효율

- 디스크는 메모리에 비해서 매우 느린 장치
- 성능을 위해서 디스크 I/O의 횟수 최소화가 바람직
  - 적은 양을 여러 번 I/O 하는 것 보다
  - 많은 양을 I/O 하여 횟수를 줄이는 것이 좋다
- write1.c 에서 BUFFSIZE 값을 변경하며 수행 시간을 측정
  - 8192 까지는 BUFFSIZE 가 클수록 성능이 향상
  - 8192 보다 큰 경우는 성능에 변화가 없다



I/O 효율

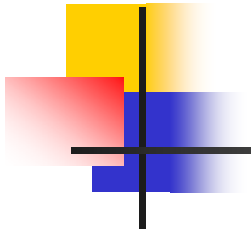
- BSD fast file system 의 I/O 단위는 8192 바이트
- BUFSIZE가 8192 보다 큰 경우
  - 내부 I/O 의 단위는 8192 이므로 성능의 향상이 없다
- BUFSIZE가 8192의 배수가 아닌 경우는
  - 오히려 8192 인 경우보다 I/O 횟수가 많아져
  - 성능이 하락할 수 있다

예제: /* write1.c */

```
#include <sys/types.h> /* write1.c */
#include <sys/stat.h>
#include <fcntl.h>

#define BUFFSIZE 512
#define FILESIZE (100 * 1024 * 1024)
#define COUNT    FILESIZE / BUFFSIZE

int main() {
  int i, fd;
  char buf[BUFFSIZE];

  memset(buf, '.', BUFFSIZE);
  if ((fd = creat("file.write", 0600)) < 0)
    perror("file.write");

  for (i=0; i < COUNT; ++i)
    write(fd, buf, BUFFSIZE);

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

예제: /* write1.c */

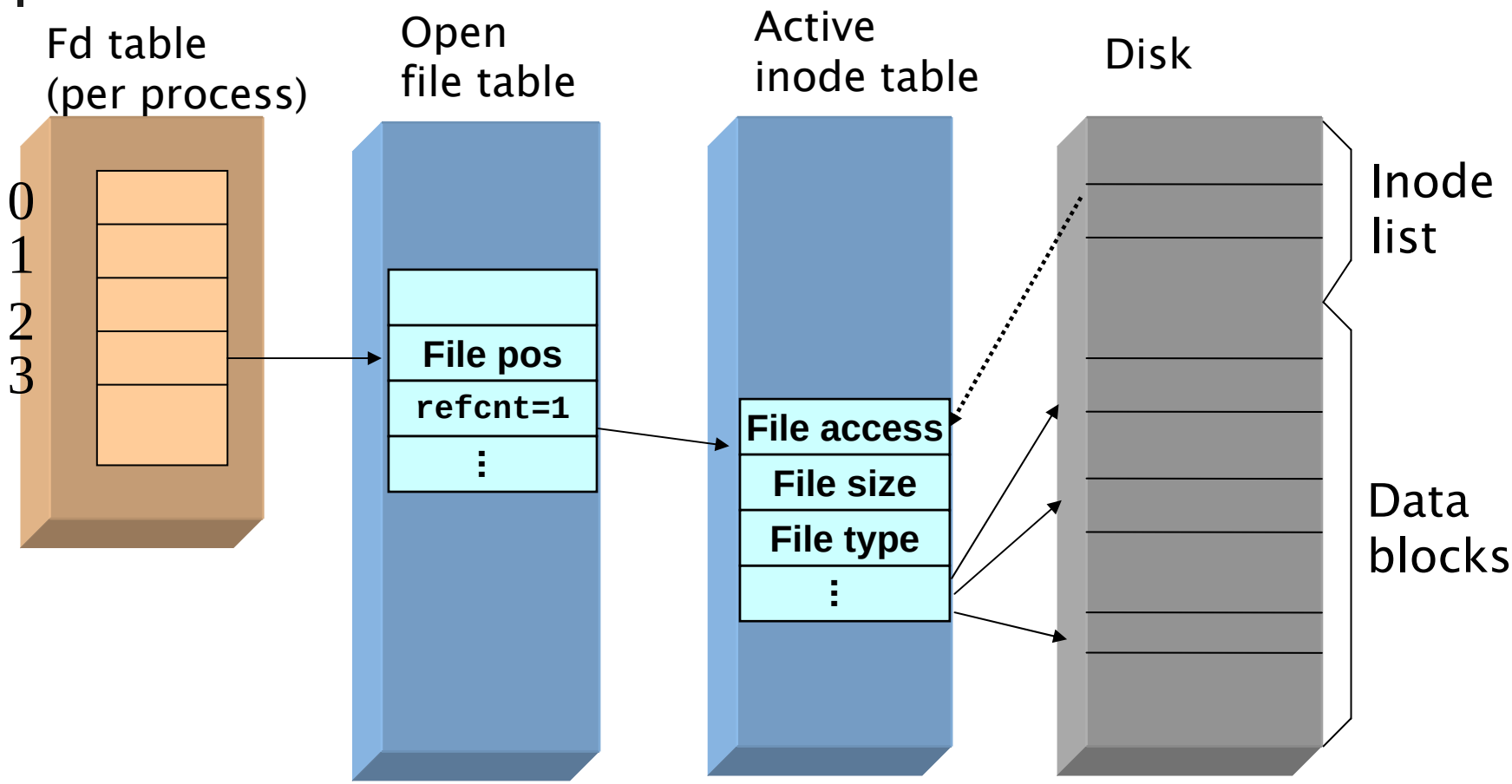
BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	140.9	1,378.1	1,527.8	104,857,600
2	73.3	681.1	764.5	52,428,800
4	31.6	343.9	379.6	26,214,400
8	16.6	172.3	191.5	13,107,200
16	8.5	86.7	98.1	6,553,600
32	3.9	45.1	51.6	3,276,800
64	2.0	23.5	28.1	1,638,400
128	1.1	12.8	16.3	819,200
256	0.5	7.4	10.4	409,600
512	0.3	4.9	7.9	204,800
1,024	0.1	2.9	6.0	102,400
2,048	0.0	2.0	4.7	51,200
4,096	0.0	1.6	4.2	25,600
8,192	0.0	1.1	3.6	12,800
16,384	0.0	1.1	3.6	6,400
32,768	0.0	1.1	3.6	3,200
65,536	0.0	1.1	3.6	1,600
131,072	0.0	1.1	3.6	800

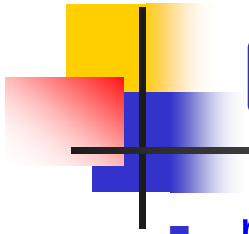


read/write 구현

- read/write 시스템 호출
  - file table entry의 file position 값이 읽은/쓰여진 바이트 수 만큼 증가
- 파일 크기 증가
  - Write 후 file position 이 i-node의 file size보다 크면 파일 크기 증가
- 파일을 O_APPEND로 열면
  - 먼저 file position 을 i-node 의 file size 값으로 설정한 후 write 수행
  - 따라서 모든 write는 파일의 끝에 덧붙여진다.

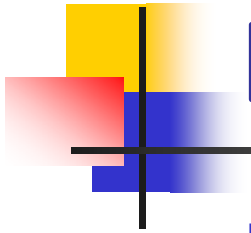
read/write 시스템 호출 구현





Example

- read(fd, buf1, 100);
  - copy the first block from disk into an buffer
  - copy the first 100 bytes from the buffer into buf1
  - offset <- 100
- read(fd, buf2, 200);
  - copy the next 200 bytes from the buffer into buf2
- read(fd, buf3, 5000);
  - copy the remaning (3796bytes) from the buffer to buf3
  - copy the second block from disk into an buffer
  - copy the remainig(1204 bytes) from the buffer into buf3
- write(fd, buf4, 100);
- write(fd, buf5, 4000);

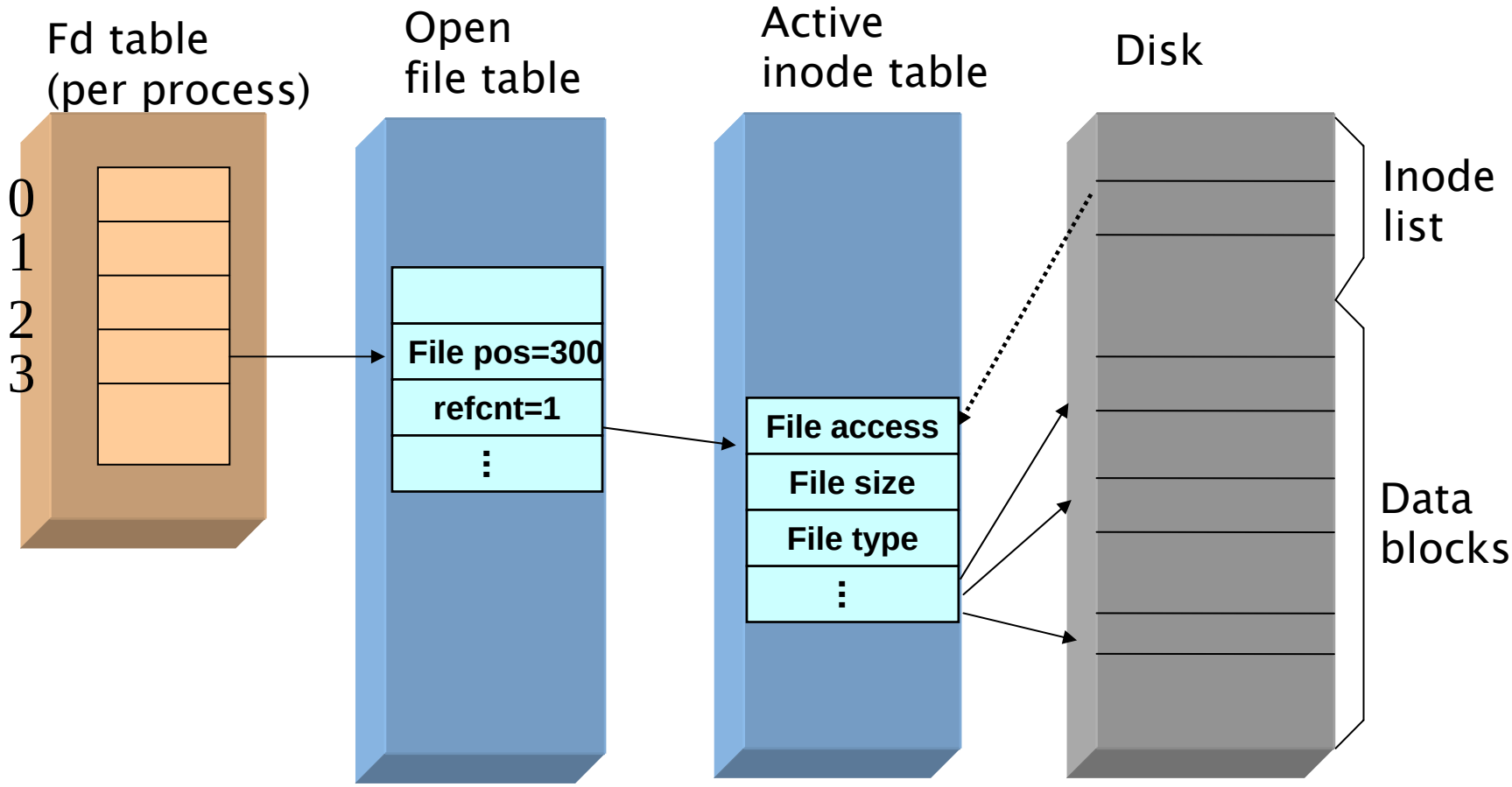


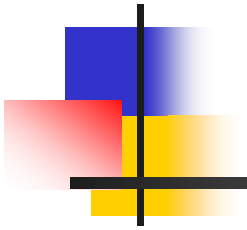
lseek 구현

- lseek 시스템 호출
  - file table entry 의 current file offset 값을 변경
  - disk I/O 는 없음
- SEEK_END를 사용하여 파일의 끝으로 이동하면
  - file table entry 의 current file offset 값은 i-node 의 파일 크기 값으로 설정됨

lseek 구현

- lseek(fd, 300, L_SET)





2.4 파일 공유

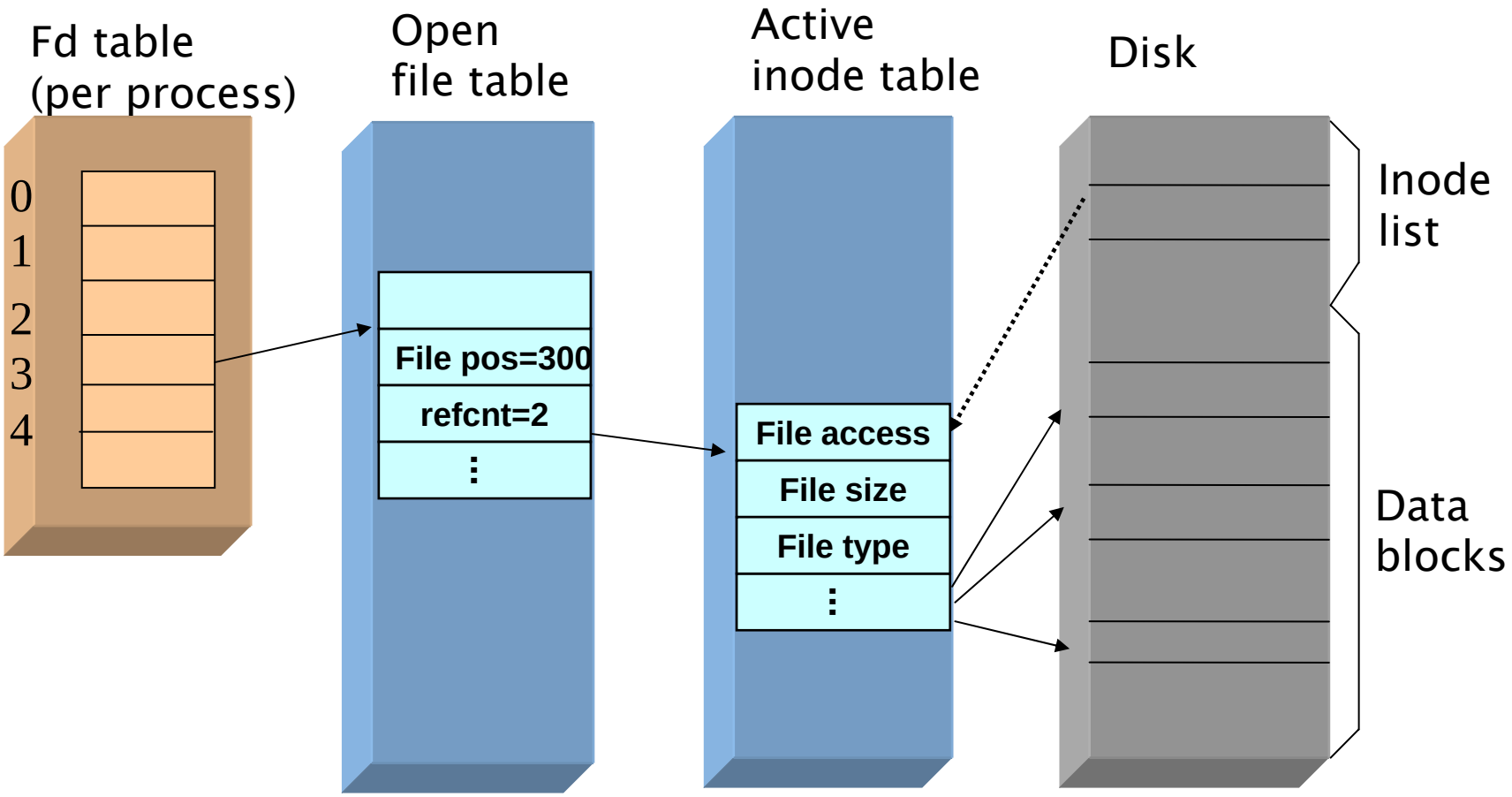
dup(), dup2()

```
#include <unistd.h>
int dup (int fd);
int dup2 (int fd, int fd2);
```

- 사용 중인 파일 디스크립터의 복사본을 만듦
  - dup()는 새 파일 디스크립터 번호가 할당됨
  - dup2()는 *fd2* 를 사용
- 리턴 값
  - 성공하면 복사된 새 파일 디스크립터, 실패하면 -1
  - dup() 함수는 할당 가능한 가장 작은 번호를 리턴한다.
  - dup2() 함수는 *fd2* 를 리턴한다.

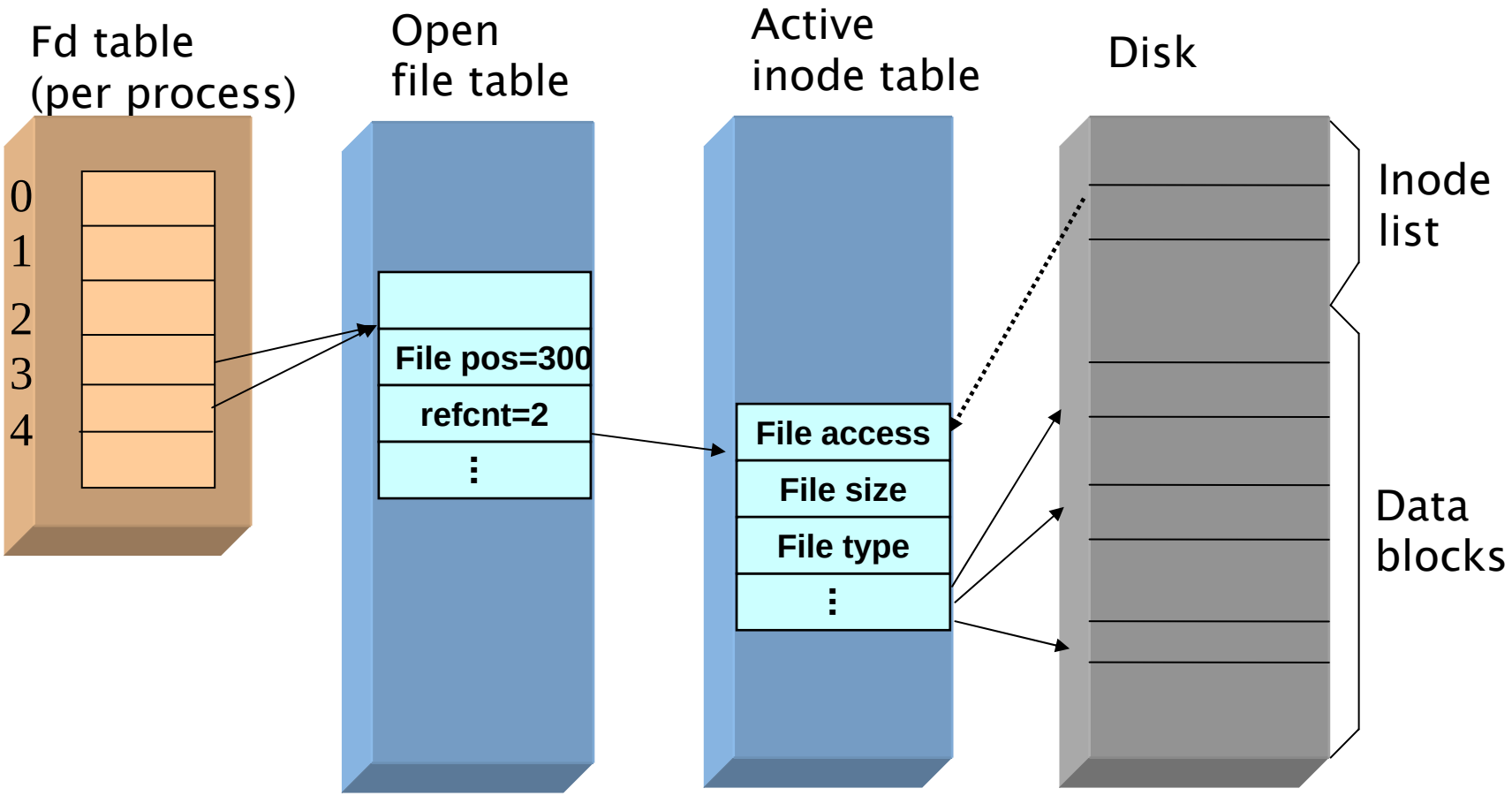
dup 구현

- fd2 = dup(3);



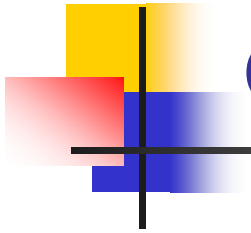
dup 구현

- fd2 = dup(3);



예제 /* dup.c */

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

int main()
{
  int fd1, fd2, n;
  char buf[11];

  if ((fd1 = open("afile", O_RDONLY)) < 0)
    perror("afile");
  if ((fd2 = dup(fd1)) < 0)
    error("dup");

  n = read(fd1, buf, 10);
  buf[n] = '\0';
  puts(buf);

  n = read(fd2, buf, 10);
  buf[n] = '\0';
  puts(buf);

  close(fd1);
  close(fd2);
  return 0;
}
```

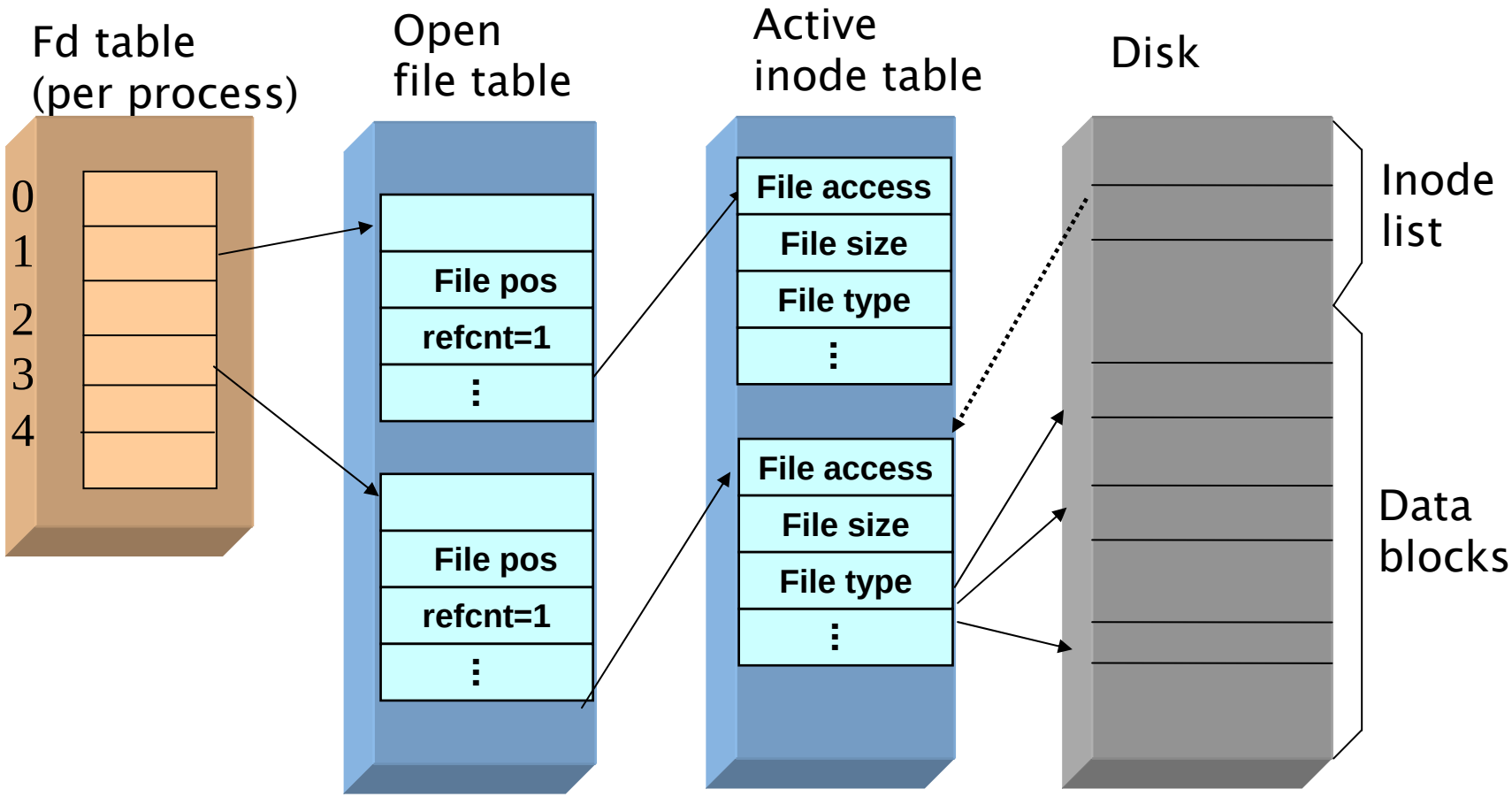


dup2()

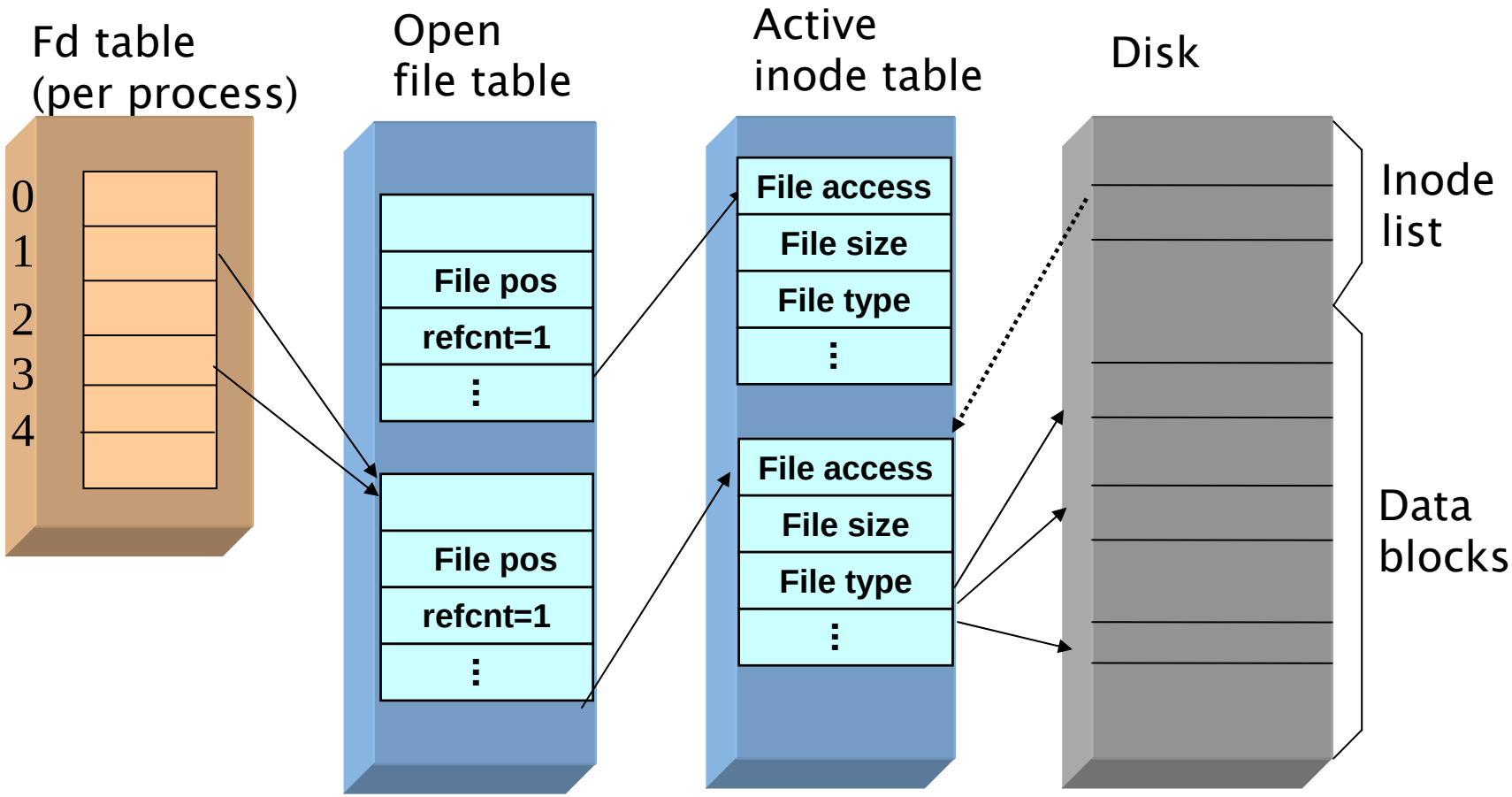
- newfd = dup2(3,1); 호출 후
  - 리턴 값은 1

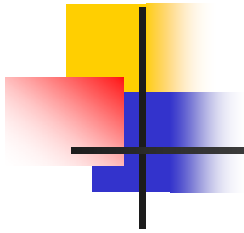
- file table의 file position과 fd status flags는 공유됨
- file descriptors의 fd flags는 공유 안 됨

dup2(3,1) 구현


Fd table
(per process)
Open
file table
Active
inode table
Disk
0
1
2
3
4
File pos
refcnt=1
File pos
refcnt=1
File access
File size
File type
File access
File size
File type
Inode
list
Data
blocks

dup2(3,1) 구현


Fd table
(per process)
Open
file table
Active
inode table
Disk
0
1
2
3
4
File pos
refcnt=1
File pos
refcnt=1
File access
File size
File type
File access
File size
File type
Inode
list
Data
blocks



dup(), dup2() 의 용도

- 표준 입출력의 redirection
  - 표준 입출력 대상을 파일로 바꿈
- 표준 입출력의 파일 디스크립터
  - #include <unistd.h>
  - #define STDIN_FILENO 0   /* 표준 입력 */
  - #define STDOUT_FILENO 1   /* 표준 출력 */
  - #define STDERR_FILENO 2   /* 표준 에러 */

예제: /* dup1.c */

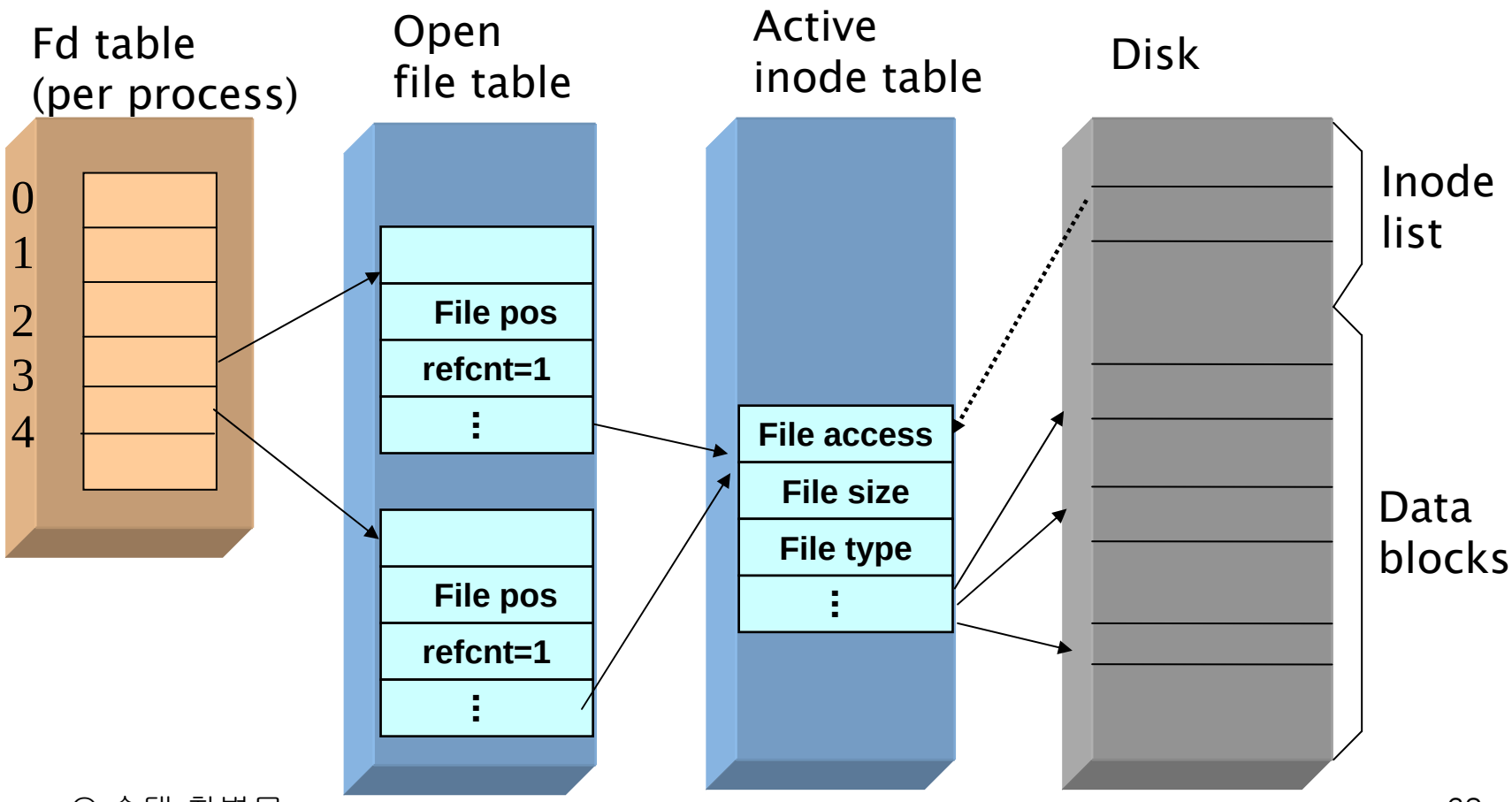
```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

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;
}
```

예제: /* dup2.c */

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.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;
}
```

open() 두 번

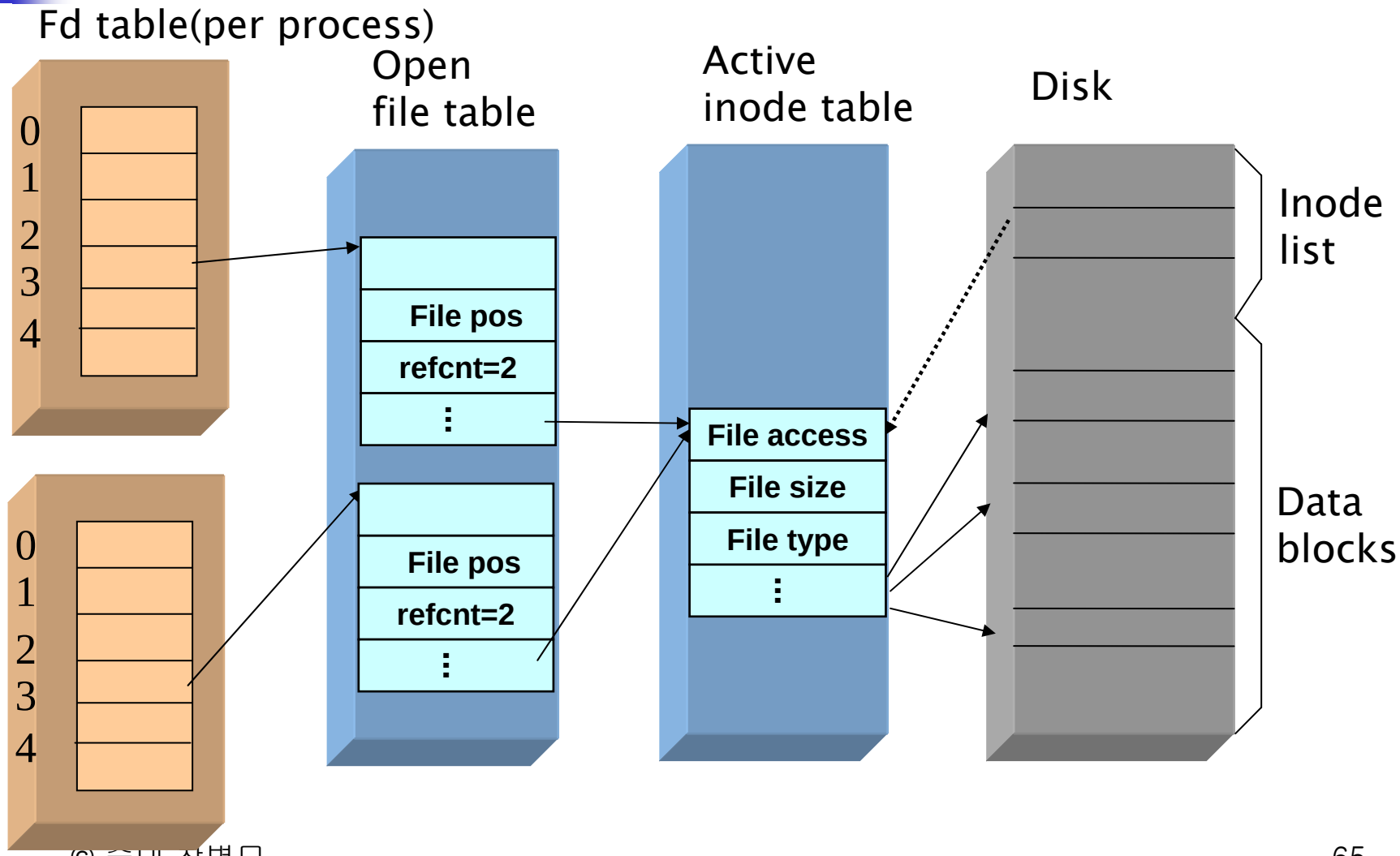
- 같은 파일 두 번 open("file", O_RDONLY);
- file position은 공유되지 않는다



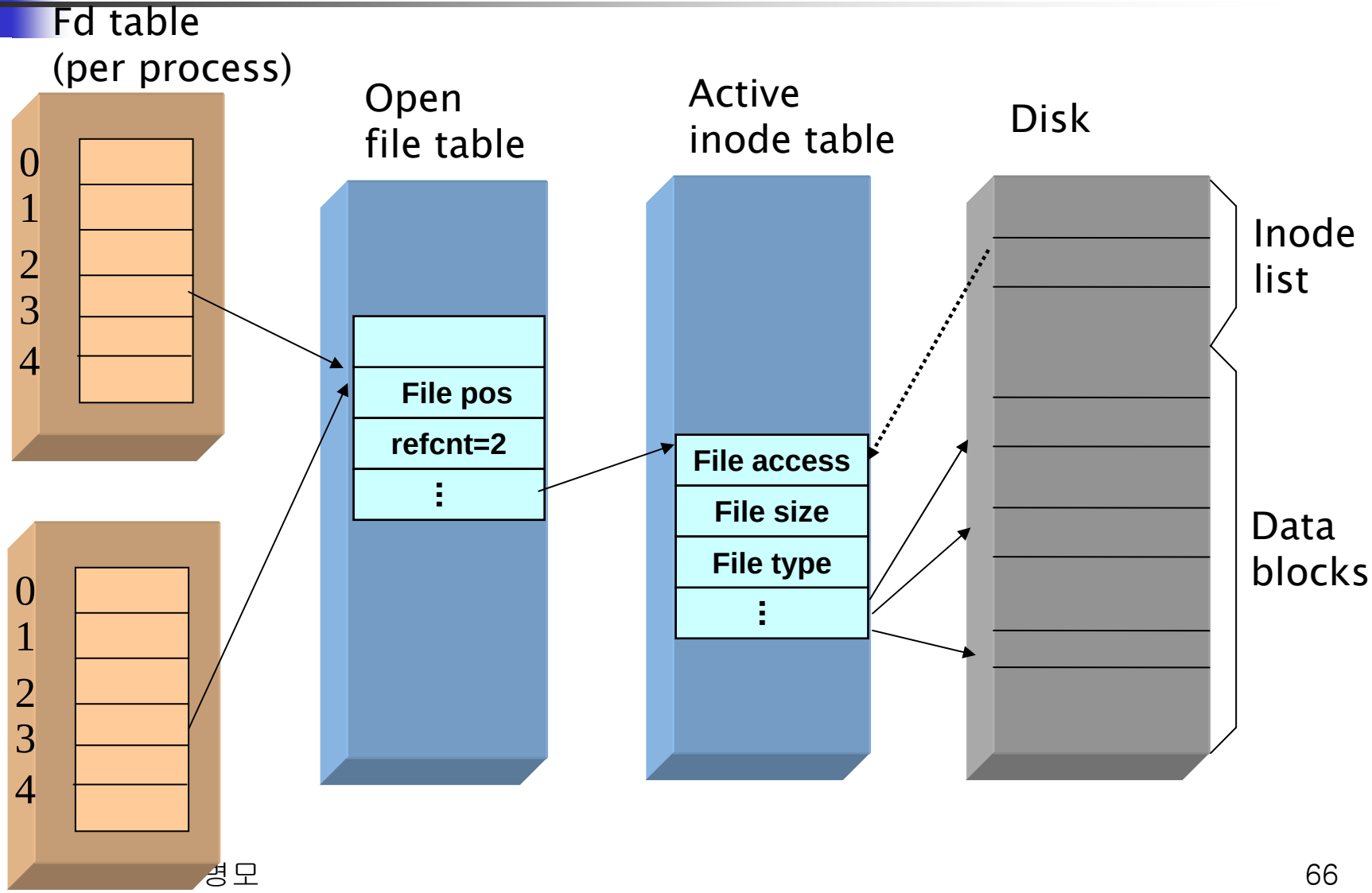
예제 /* open2.c */

```
#include <fcntl.h>          /* open2.c */
#include <unistd.h>

int main()
{
  int fd1, fd2, n;
  char buf[11];

  fd1 = open("afile", O_RDONLY);
  fd2 = open("afile", O_RDONLY);
  if (fd1 < 0 || fd2 < 0)
    perror("afile");

  n = read(fd1, buf, 10);
  buf[n] = '\0';
  puts(buf);

  n = read(fd2, buf, 10);
  buf[n] = '\0';
  puts(buf);

  close(fd1);       close(fd2);         return 0;
}
```

두 프로세스에서 같은 파일 open()



fork()


Fd table
(per process)
0
1
2
3
4
Open
file table
File pos
refcnt=2
Active
inode table
File access
File size
File type
Disk
Inode
list
Data
blocks
0
1
2
3
4