CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 11. 시스템 수준 입출력

교재 10장

Nov. 15, 2010

http://eslab.cnu.ac.kr

전달 사항

- 헤르미온느와 론 위즐리의 숙제
- 가을을 맞이해서 해야할 일

Introduction(1)

- Input/Output
  - **process of copying data between main memory and external devices such as disk, terminals, and networks**
- Input copies data from an I/O device to main memory
- Output copies data from memory to a device
- All language run-time systems provide higher level facilities for performing I/O
  - **ANSI C : standard I/O library, e.g.** `printf, fopen`
  - **C++ : <<("put to"), >> ("get from")**
- On Unix, these higher level I/O functions are implemented using system-level *Unix I/O functions* provided by kernel

Introduction (2)

- Most of the time, the higher level I/O functions works well and there is no need to use Unix I/O directly.
- Why bother learning about Unix I/O ?
  - **Understanding Unix I/O will help you understand other systems concepts**
    - e.g. I/O is integrated into the kernel
    - I/O plays key role in process creation and execution
  - **Sometimes you have no choice but to use Unix I/O**
    - e.g. no access to file metadata
- Ch.10 is about Unix I/O and standard I/O and you will learn how to use them reliably

Unix 에서의 파일이란

- A Unix *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- All I/O devices are represented as files:
  - **`/dev/sda2` (`/usr` disk partition)**
  - **`/dev/tty2` (terminal)**
  - **`/dev/cdrom`**
- Even the kernel is represented as a file:
  - **`/dev/kmem` (kernel memory image)**
  - **`/proc` (kernel data structures)**

Unix 파일의 종류

- Regular file
  - **Binary or text file.**
  - **Unix does not know the difference!**
- Directory file
  - **A file that contains the names and locations of other files**
- Character special and block special files
  - **Terminals (character special) and disks (block special)**
- FIFO (named pipe)
  - **A file type used for interprocess communication(?)**
- Socket
  - **A file type used for network communication between processes**

Unix I/O

- The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.
- Key Unix idea: All input and output is handled in a consistent and uniform way.
- Basic Unix I/O operations (system calls):
  - **Opening and closing files**
    - `open()`and `close()`
  - **Changing the *current file position* (seek)**
    - `lseek` (not discussed)
  - **Reading and writing a file**
    - `read()` and `write()`

파일 열기

- Opening a file informs the kernel that you are getting ready to access that file.

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - **fd == -1 indicates that an error occurred**
- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - **0: STDIN_FILENO, standard input**
  - **1: STDOUT_FILENO, standard output**
  - **2: STDERR_FILENO, standard error**
  - **check <unistd.h> file**

파일 닫기

- Closing a file informs the kernel that you are finished accessing that file.

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- Closing an already closed file
  - **is an error**
  - **is a recipe for disaster in threaded programs (more on this later)**
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

파일 읽기

- Reading a file copies bytes from the current file position to memory, and then updates file position.

```
char buf[512];
int fd;        /* file descriptor */
int nbytes;    /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- Returns number of bytes read from file `fd` into buf
  - **`nbytes < 0` indicates that an error occurred.**
  - ***short counts*** **(`nbytes < sizeof(buf)`) are possible and are not errors!**

파일 쓰기

- Writing a file copies bytes from memory to the current file position, and then updates current file position.

```
char buf[512];
int fd;         /* file descriptor */
int nbytes;     /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`.
  - **`nbytes < 0` indicates that an error occurred.**
  - **As with reads, short counts are possible and are not errors!**
- Transfers up to 512 bytes from address `buf` to file `fd`

Unix I / O 예제

- Copying standard input to standard output one byte at a time.

```
int main(void)
{
    char c;
    int len;

    while ((len = read(0 /*stdin*/, &c, 1)) == 1) {
       if (write(1 /*stdout*/, &c, 1) != 1) {
          exit(20);
       }
    }
    if (len < 0) {
       printf ("read from stdin failed");
       exit (10);
    }
    exit(0);
```

- Note the use of error handling wrappers for read and write (Appendix B).

Short Counts 처리하기

- Short counts can occur in these situations:
  - **Encountering (end-of-file) EOF on reads.**
  - **Reading text lines from a terminal.**
  - **Reading and writing network sockets or Unix pipes.**
- Short counts never occur in these situations:
  - **Reading from disk files (except for EOF)**
  - **Writing to disk files.**
- How should you deal with short counts in your code?
  - **Use the RIO (Robust I/O, REB 11.4) package from your textbook's `csapp.c` file (Appendix B).**

파일의 메타데이타

- *Metadata* is data about data, in this case file data.
- Maintained by kernel, accessed by users with the `stat` and `fstat` functions.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

파일 메타데이타 접근하기

```
/* statcheck.c - Querying and manipulating a file's meta data */
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode)) /* file type*/
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* OK to read?*/
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
bass> ./statcheck statcheck.c
type: regular, read: yes
bass> chmod 000 statcheck.c
bass> ./statcheck statcheck.c
type: regular, read: no
```

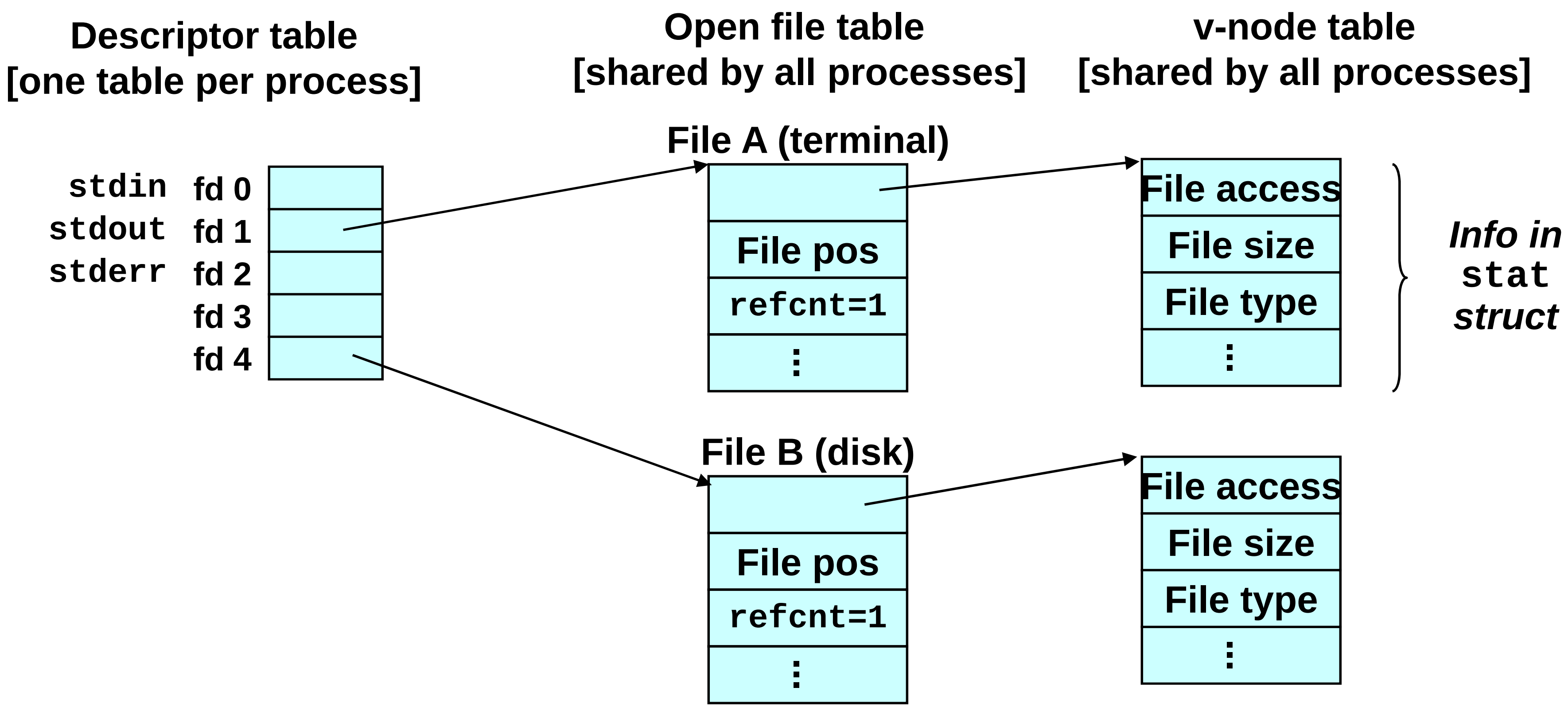
디렉토리 접근하기

- The only recommended operation on directories is to read its entries.

```
#include <sys/types.h>
#include <dirent.h>

{
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(dir_name)))
      error("Failed to open directory");
  ...
  while (0 != (de = readdir(directory))) {
      printf("Found file: %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```

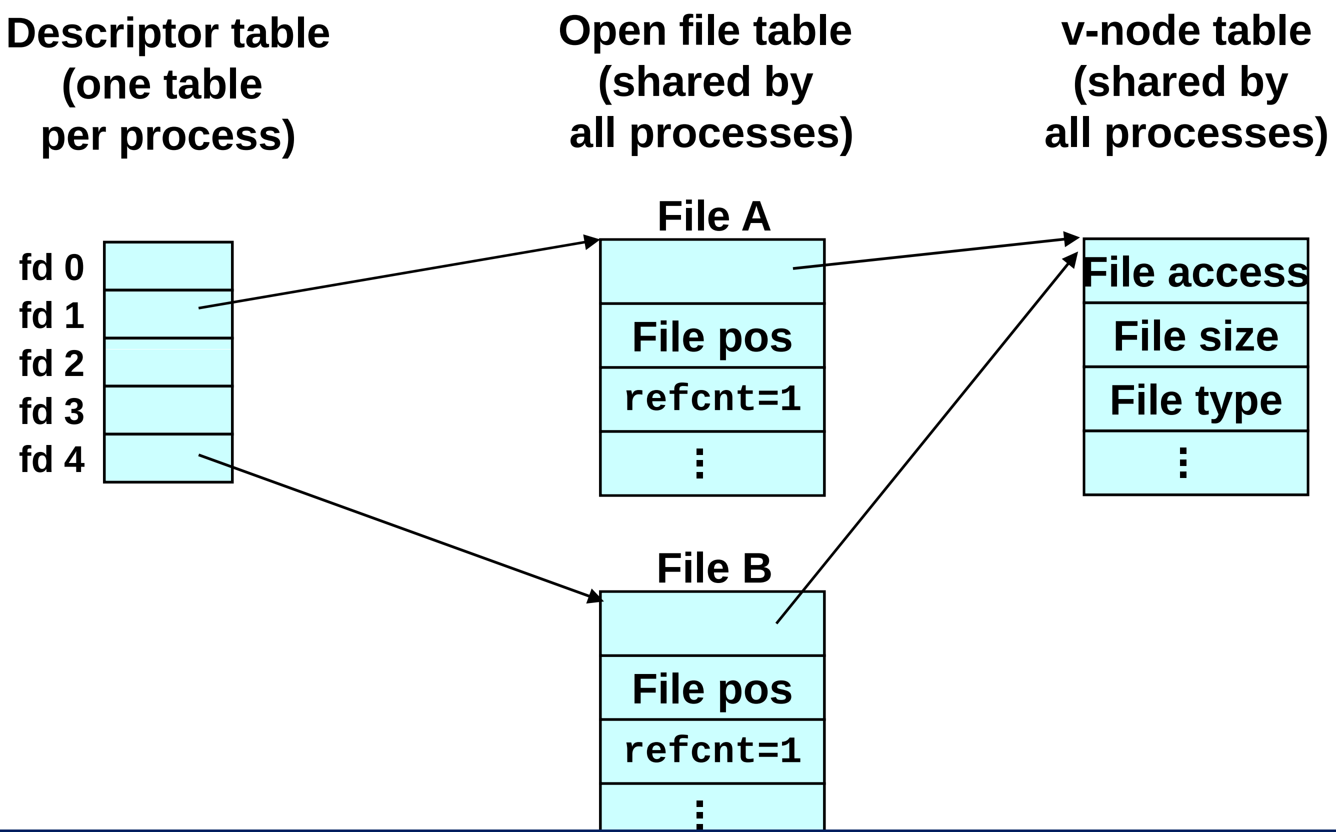
유닉스에서 파일 열기

- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file.



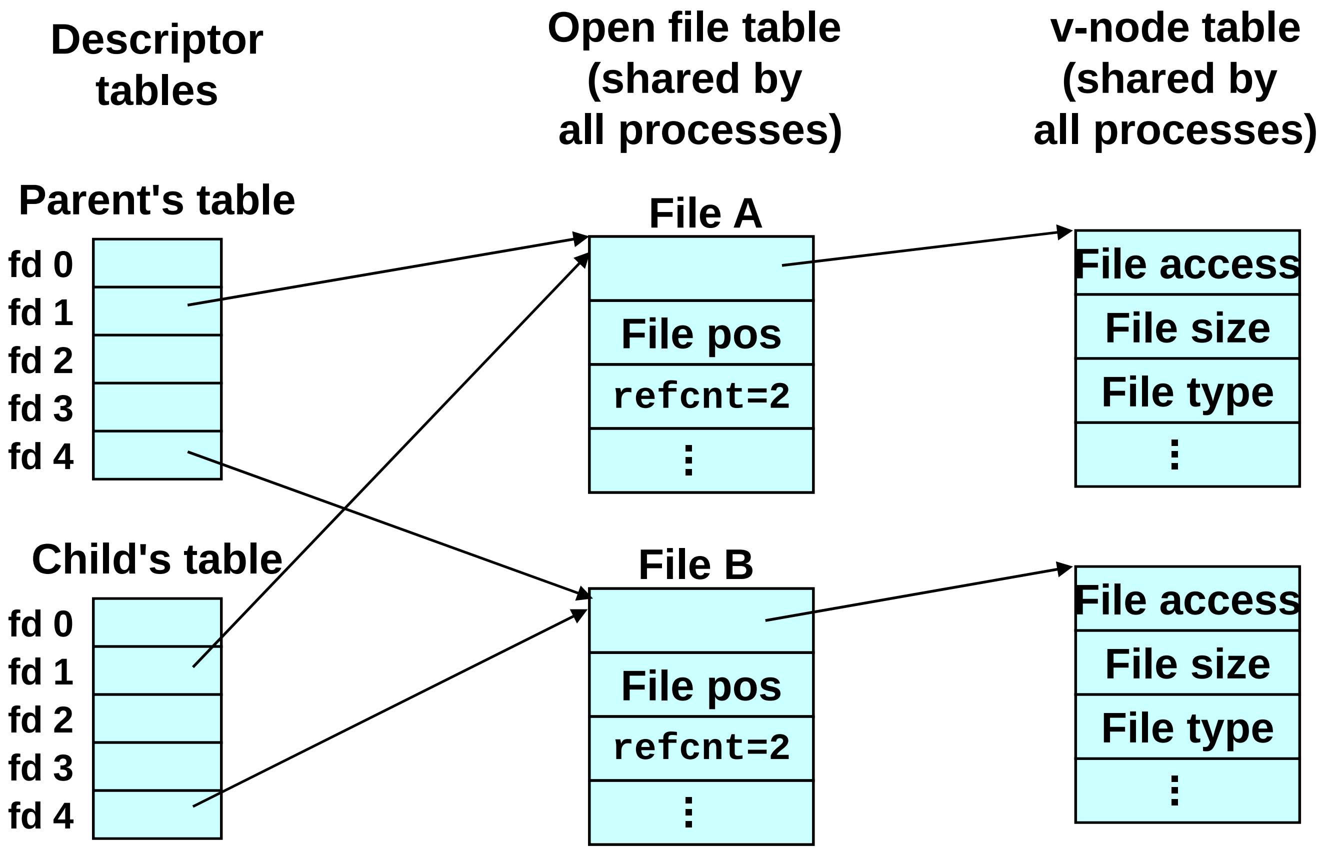
파일의 공유

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling `open` twice with the same `filename` argument



프로세스들 간의 파일의 공유

- A child process inherits its parent's open files. Here is the situation immediately after a `fork`



Problem 11.2

```
int main()
{
    int fd1, fd2;
    char c;

    fd1 = open("foobar.txt", O_RDONLY, 0);
    fd2 = open("foobar.txt", O_RDONLY, 0);
    read(fd1, &c, 1);
    read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

foobar.txt contains “foobar”, 6 characters.

화면에 무엇이 출력되는가?

Prob. 11.3

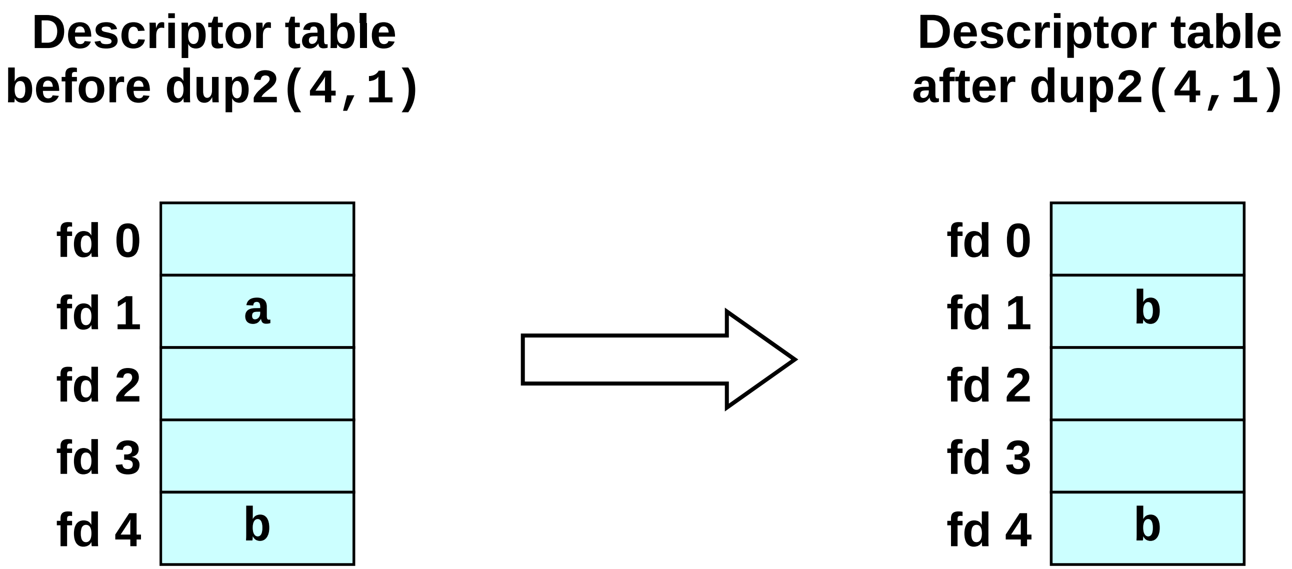
```
int main()
{
   int fd;
   char c;

   fd = open("foobar.txt", O_RDONLY, 0);
   if (fork() == 0) {
      read(fd, &c, 1);
      exit(0);
   }
   wait(NULL);
   read(fd, &c, 1);
   printf("c = %c\n", c);
   exit(0);
}
```

입출력에서의 경로변경(Redirection)

- Question: How does a shell implement I/O redirection?

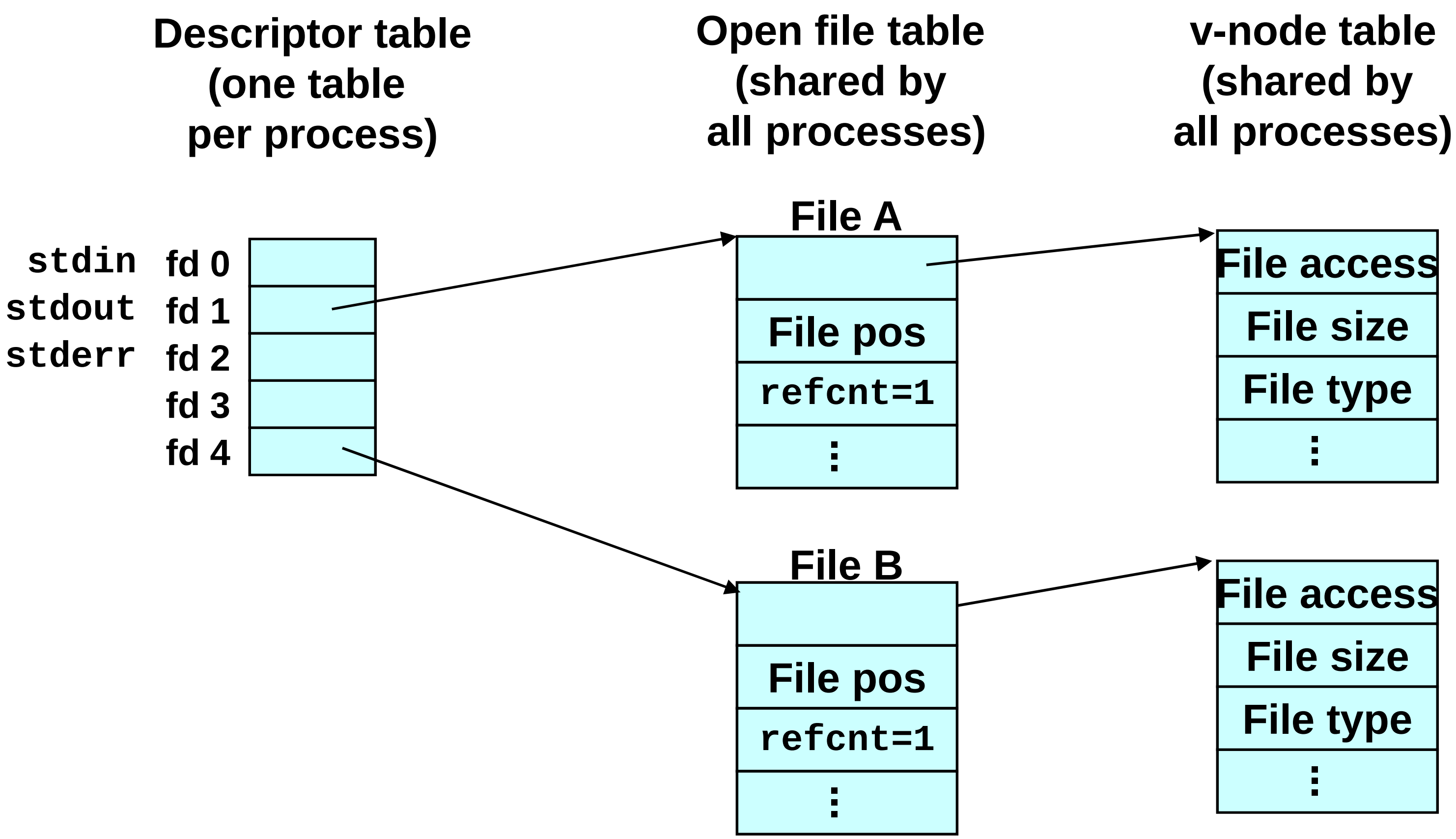
  `unix> ls > foo.txt   ; redirecting STDOUT to foo.txt`

- Answer: By calling the `dup2(oldfd, newfd)` function
  - **Copies (per-process) descriptor table entry `oldfd` to entry `newfd`**



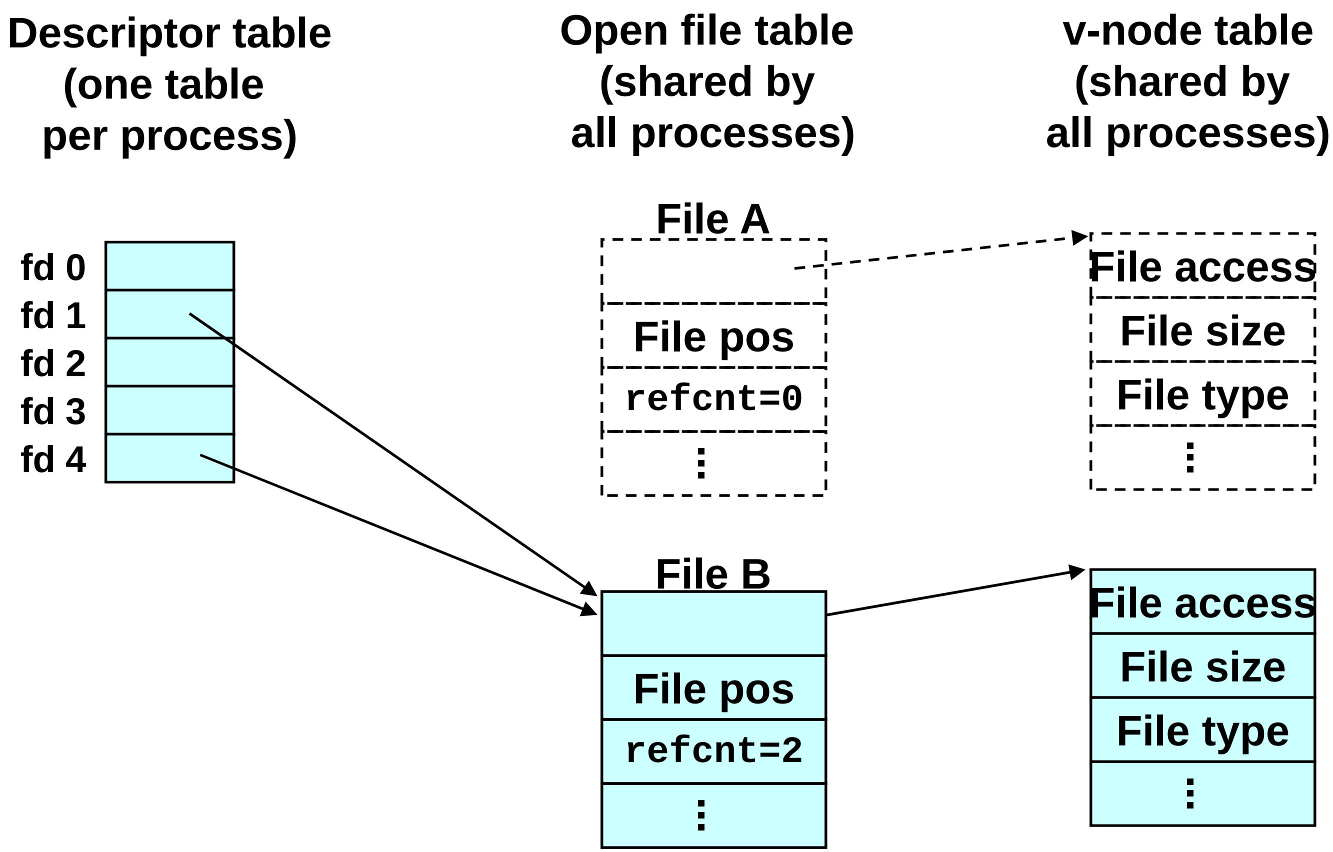
I/O 경로변경 Example

- Before calling dup2(4,1), stdout (descriptor 1) points to a terminal and descriptor 4 points to an open disk file.



I/O 경로변경 Example (cont)

- After calling `dup2(4,1)`, stdout is now redirected to the disk file pointed at by descriptor 4.



표준 입출력 함수

- The C standard library (`libc.a`) contains a collection of higher-level standard I/O functions
  - **Documented in Appendix B of K&R.**
- Examples of standard I/O functions:
  - **Opening and closing files (`fopen` and `fclose`)**
  - **Reading and writing bytes (`fread` and `fwrite`)**
  - **Reading and writing text lines (`fgets` and `fputs`)**
  - **Formatted reading and writing (`fscanf` and `fprintf`)**

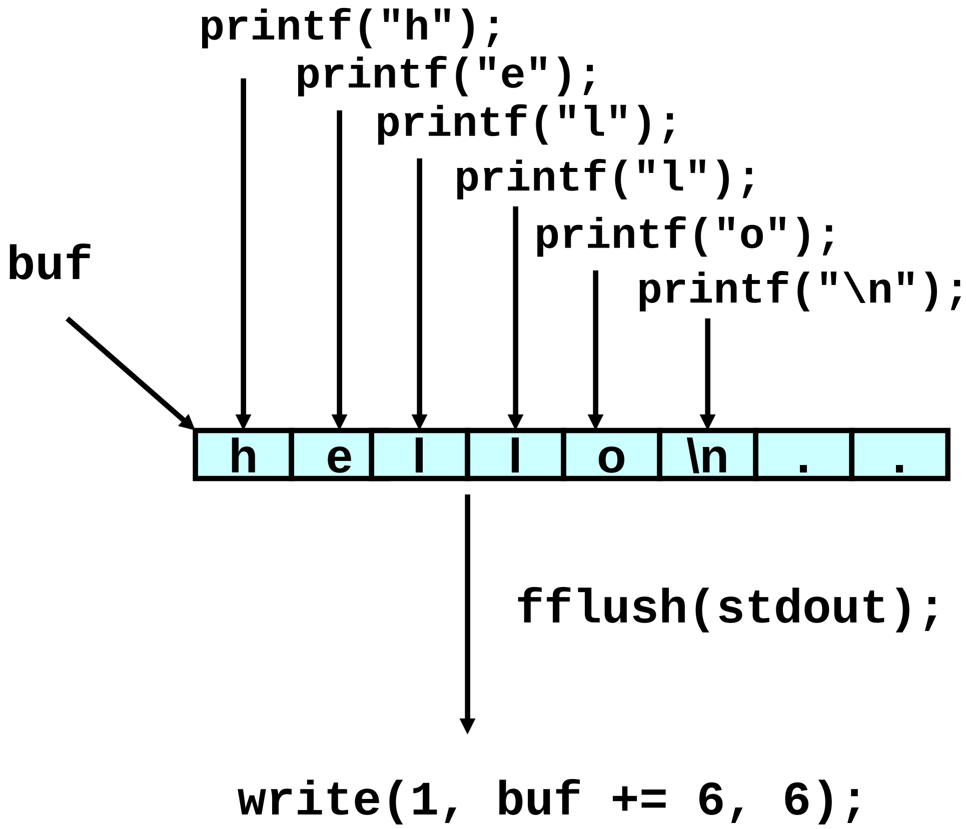
표준 입출력 스트림

- Standard I/O models open files as *streams*
  - **Abstraction for a file descriptor and a buffer in memory.**
- C programs begin life with three open streams (defined in `stdio.h`)
  - **`stdin` (standard input)**
  - **`stdout` (standard output)**
  - **`stderr` (standard error)**

```
#include <stdio.h>
extern FILE *stdin;  /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

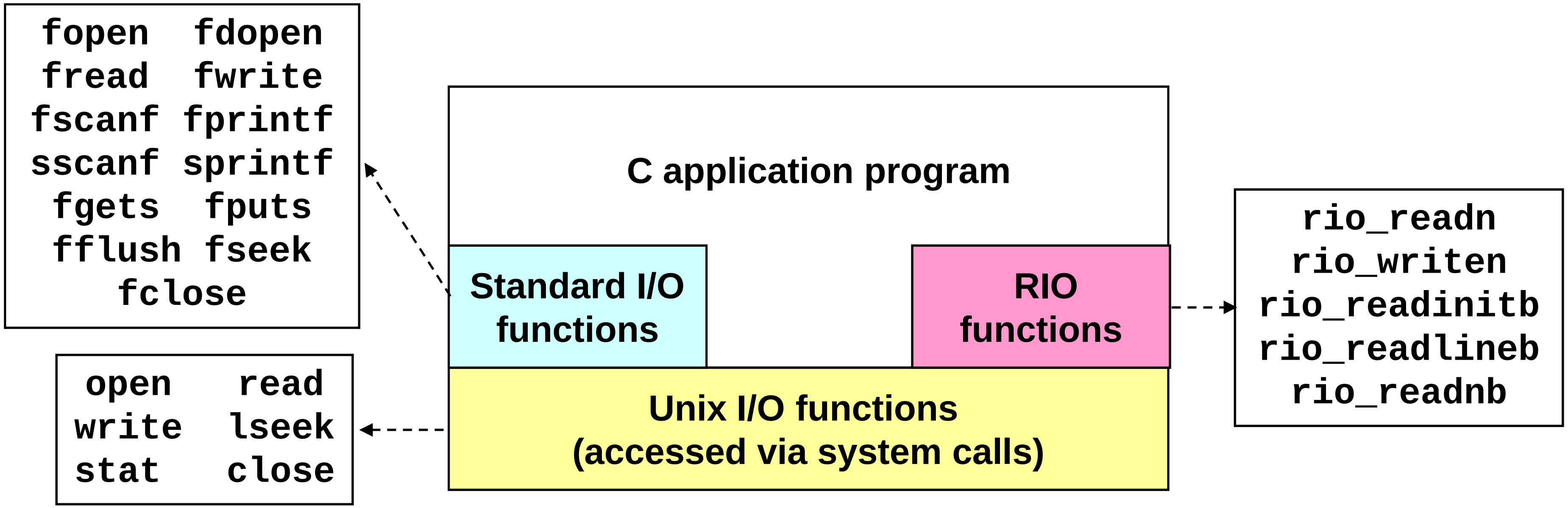
표준 입출력에서의 버퍼링

- Standard I/O functions use buffered I/O



Unix I/O vs. Standard I/O vs. RIO

- Standard I/O and RIO are implemented using low-level Unix I/O.



- Which ones should you use in your programs?

Unix I/O의 장단점

- Pros
  - **Unix I/O is the most general and lowest overhead form of I/O.**
    - All other I/O packages are implemented using Unix I/O functions.
  - **Unix I/O provides functions for accessing file metadata.**
- Cons
  - **Dealing with short counts is tricky and error prone.**
  - **Efficient reading of text lines requires some form of buffering, also tricky and error prone.**
  - **Both of these issues are addressed by the standard I/O and RIO packages.**

Standard I/O의 장단점

- Pros:
  - **Buffering increases efficiency by decreasing the number of `read` and `write` system calls.**
  - **Short counts are handled automatically.**
- Cons:
  - **Provides no function for accessing file metadata**
  - **Standard I/O is not appropriate for input and output on network sockets**
  - **There are poorly documented restrictions on streams that interact badly with restrictions on sockets**

어느 입출력 함수를 사용할 것인가

- General rule: Use the highest-level I/O functions you can.
  - **Many C programmers are able to do all of their work using the standard I/O functions.**
- When to use standard I/O?
  - **When working with disk or terminal files.**
- When to use raw Unix I/O
  - **When you need to fetch file metadata.**
  - **In rare cases when you need absolute highest performance.**
- When to use RIO?
  - **When you are reading and writing network sockets or pipes.**
  - **Never use standard I/O or raw Unix I/O on sockets or pipes.**

프로세스간의 통신
Inter-process Communication

Unix Pipes

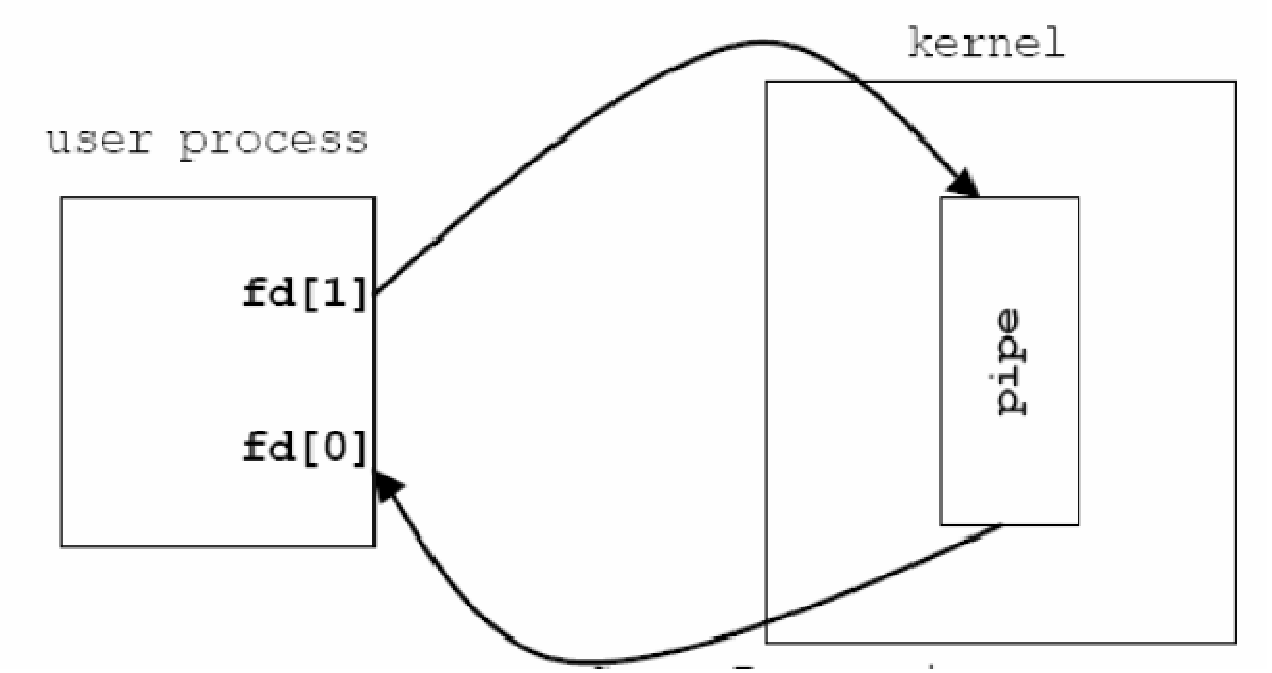
* 참고자료 : 관심있는 사람들만 !!!

파이프(Pipes)

- The oldest form of Unix IPC and provided by all Unix systems
- Two limitations
  - **Half-duplex : data flows only in one direction**
  - **Can be used only between processes that have a common ancestor**
    - Usu used between parent and child processes

파이프 생성하기(1)

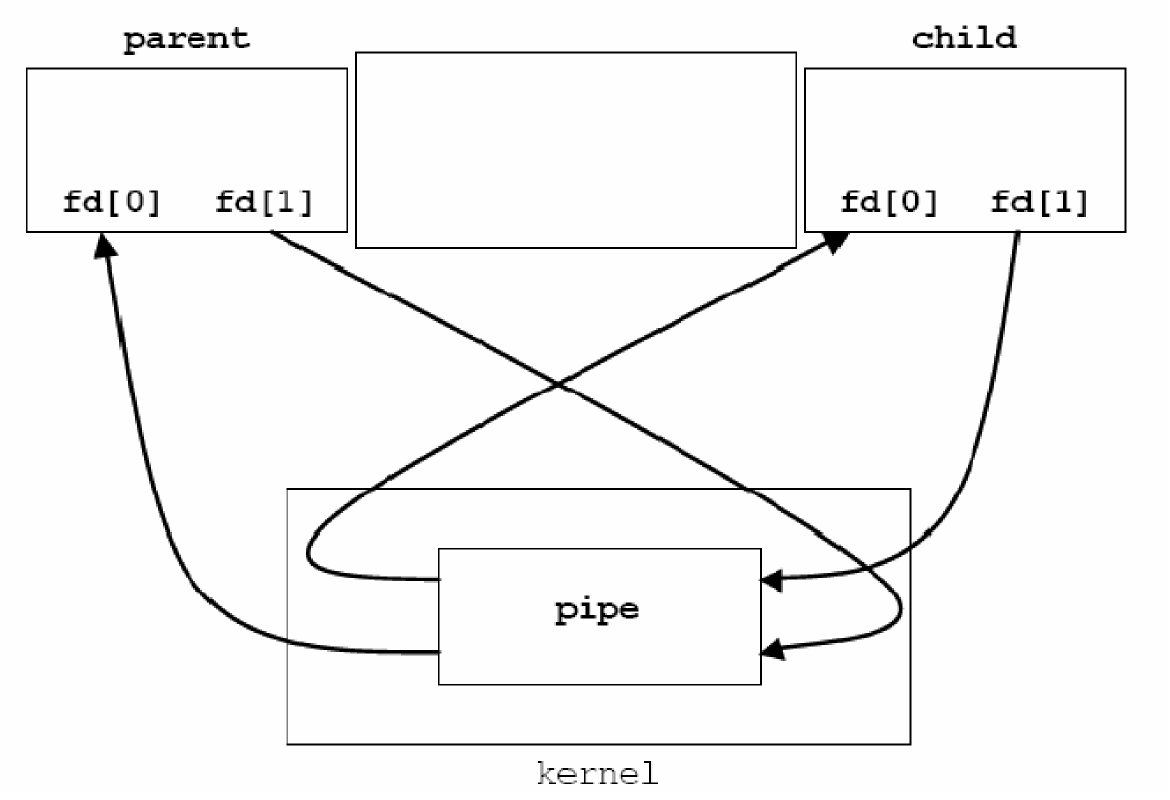
- `int pipe (int fd[2])`
  - Two file descriptors are returned through the **fd** argument
    - fd[0]: **open for reading**
    - fd[1]: **open for writing**
  - The output of **fd[1]** is the input for **fd[0].**



파이프 생성하기(2)

```
parent => child:
parent closes fd[0];
child closes fd[1];
```

```
parent <= child:
parent closes fd[1];
child closes fd[0];
```



파이프 읽기 / 쓰기

- When one end of a pipe is closed,
  - reading from a pipe returns an end of file.
  - writing to a pipe causes **SIGPIPE** is generated and the write returns an error (**EPIPE**).
  - A write of **PIPE_BUF** (kernel's pipe buffer size) bytes or less will not be interleaved with the writes from other processes.
  - **fstat** function returns a file type of FIFO for the pipe file descriptors (can be tested by **S_ISFIFO** macro)

- You should close unused file descriptors!

파이프 사용하기

```
#include <unistd.h>

#define MAXLINE   80

int main (void)
{
    int n, fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0) exit (1);
    if ((pid = fork()) < 0) exit (2);
    else if (pid > 0) {                  /* parent */
        close (fd[0]);
        write (fd[1], "hello world\n", 12);
    }
    else {                               /* child */
        close (fd[1]);
        n = read(fd[0], line, MAXLINE);
        write (1, line, n);
    }
}
```

FIFOs

- int mkfifo (const char *path, mode_t mode)
  - Named pipes
  - Unrelated processes can exchange data, whereas pipes can be used only between related processes.
  - FIFO is a type of file: FIFO type (**S_ISFIFO** macro)
  - Once a FIFO created, the normal file I/O functions all work with FIFO.

- /usr/bin/mkfifo program can be used to make FIFOs on the command line.

FIFOs 사용하기

Opening a FIFO

- An open for read(write)-only blocks until some other process opens the FIFO for writing(reading).

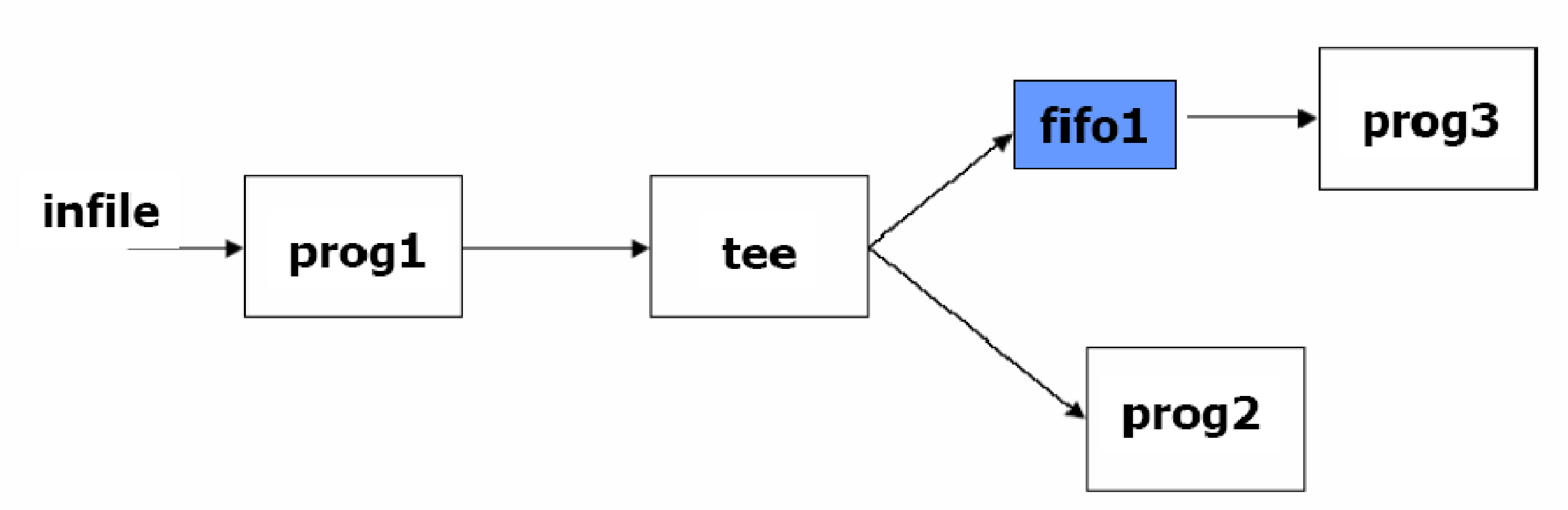
Reading/Writing a FIFO

- Writing to a FIFO that no process has open for reading causes **SIGPIPE** to generate.
- When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.
- **PIPE_BUF**: the maximum amount of data that can be written atomically to a FIFO (without being interleaved among multiple writers).

FIFO의 용도(1)

- Duplicating a Stream
  - Shell commands to pass data from one shell pipeline to another without creating intermediate temporary files

```
$ mkfifo fifo1
$ prog3 < fifo1 &
$ prog1 < infile | tee fifo1 | prog2
```



FIFO의 용도(2)

- Client-server Communication
  - A client-server application to pass data between the clients and server on the same machine.
    - Clients write to a “well-known” FIFO to send a request to the server.