Chapter 15

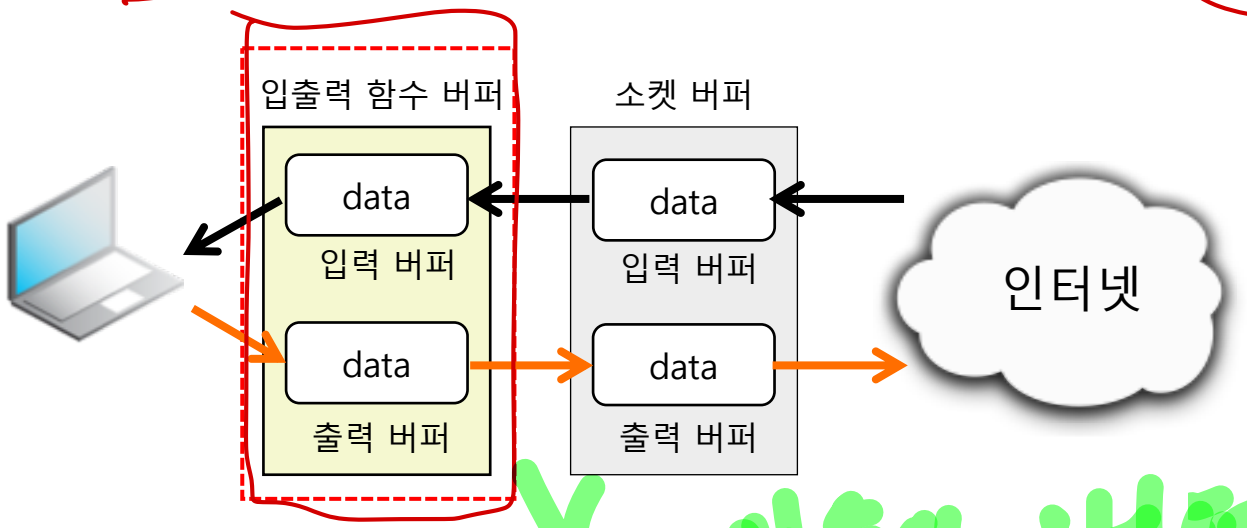
소켓과 표준 입출력

표준 입출력 함수의 장점

- 표준 입출력 함수 장점

  - 표준 입출력 함수는 이식성(Portability)이 좋음
  - 표준 입출력 함수는 버퍼링을 통한 성능 향상에 도움이 됨

  – ANSI C 기반의 표준 입출력 함수는 모든 컴파일러가 지원하기 때문에 이식성이 좋음



표준 입출력 함수를 이용해서 데이터를 전송할 경우 그림과 같이 소켓의 입출력 버퍼 이외의 입출력 함수 버퍼를 통해서 버퍼링이 됨

표준 입출력 함수와 시스템 함수 성능 비교

예제 syscpy.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/time.h>
#include <unistd.h>

#define BUF_SIZE 3

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

    fd1 = open("news.txt", O_RDONLY);
    fd2 = open("cpy.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);

    while((len=read(fd1, buf, sizeof(buf))) > 0)
        write(fd2, buf, len);

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

예제 stdcpy.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/time.h>
#include <unistd.h>

#define BUF_SIZE 3

int main(int argc, char *argv[])
{
    FILE *f1;
    FILE *f2;
    char buf[BUF_SIZE];

    f1 = fopen("news.txt", "r");
    f2 = fopen("cpy.txt", "w");

    while(fgets(buf, BUF_SIZE, f1) != NULL)
        fputs(buf, f2);

    fclose(f1);
    fclose(f2);
    return 0;
}
```

- 300 Mbytes 이상의 파일을 대상으로 테스트 시 속도 차이가 매우 큼

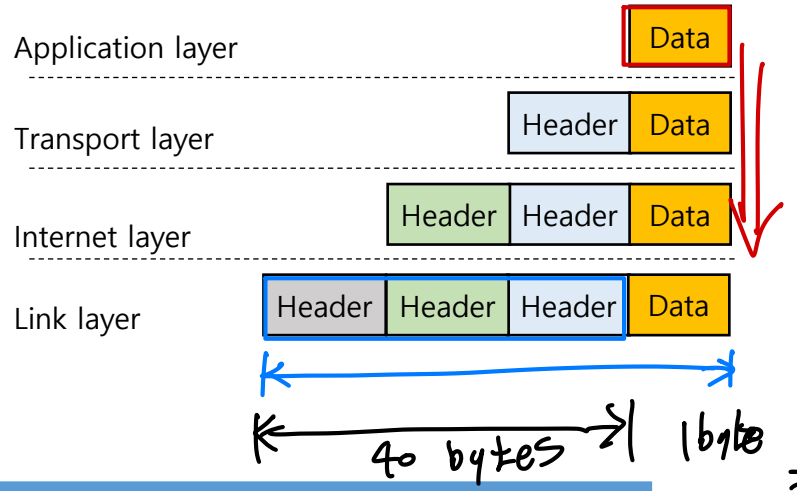
- 시스템 함수를 이용한 파일 복사
  - ✓ 버퍼링 없는 파일 복사
  - ✓ open(), read(), write() 함수

- 표준 입출력 함수를 이용한 파일 복사
  - ✓ 버퍼링 기반의 파일 복사
  - ✓ fopen(), fgets(), fputs() 함수

버퍼링 성능 비교

- 버퍼링
  - 전송해야 될 데이터 양이 많은 경우 성능 차이가 발생

- 고려 사항
  - 전송하는 데이터 양
  - 출력 버퍼로 데이터의 이동 횟수



⇒ 41 bytes

전송 데이터 크기	전송 횟수	전체 전송 데이터 크기 (전체 헤더 크기: 40 bytes + 데이터)
1 byte	10회	(40 + 1) bytes x 10회 = 410 bytes
10 bytes	1 회	(40 + 10) bytes x 1회 = 50 bytes

1 byte

10 byte

50 bytes x 1

시스템 함수 기반의 파일 복사: syscpy1.c

- 시스템 함수 기반의 파일 복사: read(), write()

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

#define BUF_SIZE 3

int main(int argc, char *argv[])
{
    int fd1, fd2;
    int len;
    char buf[BUF_SIZE];
    const unsigned long nano = 1000000000;
    unsigned long t1, t2;
    struct timespec start, end;

    if(argc != 3)
    {
        printf("Usage: %s <src_file> <dest_file>\n", argv[0]);
        return -1;
    }

    fd1 = open(argv[1], O_RDONLY);
    fd2 = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0644);

    clock_gettime(CLOCK_REALTIME, &start);
    t1 = start.tv_nsec + start.tv_sec * nano;

    while((len=read(fd1, buf, sizeof(buf))) > 0)
        write(fd2, buf, len);

    clock_gettime(CLOCK_REALTIME, &end);
    t2 = end.tv_nsec + end.tv_sec * nano;

    printf("syscpy elapsed time: %ld milliseconds\n",
                    (t2-t1)/1000000); // 1 msec = 10^6 nano sec

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

동작 지연을 위해 버퍼 크기를 작게 설정

수행 시간 측정

```
$ gcc syscpy1.c –o syscpy1
$ ./syscpy1 sample.flac sys.flac
syscpy elapsed time: 12794 milliseconds
```

- 1 sec = 1000 msec
- 1 msec = 1000 usec

- 1 usec = 1000 nano sec
- 1 msec = 10^6 nano sec

표준 함수 기반의 파일 복사: stdcpy1.c

- 표준함수 기반의 파일 복사: fgets(), fputs()

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

#define BUF_SIZE 3

int main(int argc, char *argv[])
{
    FILE *f1;
    FILE *f2;
    char buf[BUF_SIZE];
    unsigned long t1, t2;
    unsigned long nano = 1000000000;
    struct timespec start, end;

    if(argc != 3)
    {
        printf("Usage: %s <src_file> <dest_file>\n", argv[0]);
        return -1;
    }

    f1 = fopen(argv[1], "r");
    f2 = fopen(argv[2], "w");

    clock_gettime(CLOCK_REALTIME, &start);
    t1 = start.tv_nsec + start.tv_sec * nano;

    while(fgets(buf, BUF_SIZE, f1) != NULL)
        fputs(buf, f2);

    clock_gettime(CLOCK_REALTIME, &end);
    t2 = end.tv_nsec + end.tv_sec * nano;

    printf("Stdcpy elapsed time: %ld milliseconds\n",
                                (t2-t1)/1000000);

    fclose(f1);
    fclose(f2);
    return 0;
}
```

수행 시간 측정

nano sec → milli sec 변환

```
$ gcc stdcpy1.c –o stdcpy
$ ./stdcpy1 sample.flac std.flac

Stdcpy elapsed time: 577 milliseconds
```

수행 시간 비교

File size	syscpy1.c	stdcpy1.c	비고
25 Mbytes (sample.flac)	12794 msec	577 msec	• 표준 입출력 함수가 약 20배 빠름
25 Kbyte (txt) (data.txt)	11 msec	0 msec	

```c
#include <time.h>

int clock_gettime(clockid_t clk_id, struct timespec *res);
```

- 현재의 시간을 가져옴: nanosecond 단위의 시간까지 리턴

```c
struct timespec
{
    time_t tv_sec;   // seconds
    long tv_nsec;    // nanoseconds
}
```

```
clockid_t
- CLOCK_REALTIME: 시스템 전역에서 사용되는 실제 시간
- CLOCK_MONOTONIC: 특정 시점 이후로 흐른 시간
```

표준 입출력 함수 사용의 불편 사항

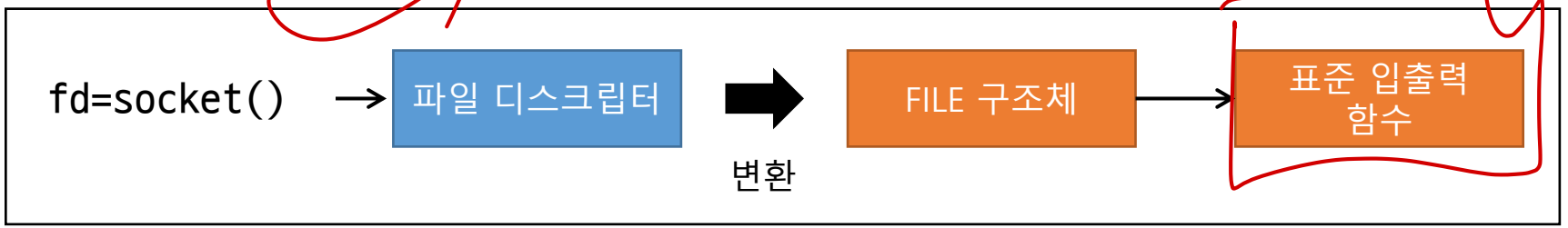
- 표준 입출력 함수 사용

  - 양방향 통신이 쉽지 않다.
  - 상황에 따라서 빈번한 fflush 함수의 호출이 필요함
  - 파일 디스크립터를 FILE 구조체 포인터로 변환해야 됨

  – fopen() 함수 호출시 반환되는 FILE 구조체 포인터를 대상으로 입출력을 진행할 경우
    - ➢ 입력과 출력이 동시에 진행되도록 하는 것은 간단하지 않음(r+, w+, a+ 옵션 사용)
    - ➢ 데이터가 버퍼링 됨: fflush() 함수 사용

  – 소켓 생성시 반환되는 것은 파일 디스크립터

  – 표준 C 함수에서 요구하는 것은 FILE 구조체 포인터임
    - ➢ 파일 디스크립터를 FILE 구조체 포인터로 변환하는 과정이 필요함



FILE 구조체 (Linux 버전에 따라 다름)

```
typedef struct _IO_FILE __FILE;
```

```
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

   ... // 생략

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

fdopen(int fd) 및 fileno(fp) 호출시 사용됨

fdopen 함수를 이용한 FILE 구조체 포인터 변호나

- fdopen() 함수

```
#include <stdio.h>

FILE* fdopen(int filedes, const char * mode);
-> 성공 시 변환된 FILE 구조체 포인트, 실패 시 NULL 반환
```

"w", "r"

fdopen(fd)
- 파일 디스크립터 → FILE 포인터
  (fd) (fp)
  소켓 → FILE*

- filedes: 변환할 파일 디스크립터를 인자로 전달
- mode: 생성할 FILE 구조체 포인터의 모드 정보 전달

<예제 desto.c>

```
#include <stdio.h>
#include <fcntl.h>
int main(void)
{
    FILE *fp;
    int fd = open("data.dat", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    if(fd==-1)
    {
        fputs("file open error", stdout);
        return -1;
    }

    fp = fdopen(fd, "w");
    fputs("Network C programming \n", fp);
    fclose(fp);
    return 0;
}
```

fd ⇒ FILE* fp 로 변환 ⇓ fputs의 호출

FILE* fp

실행 결과

```
$ gcc desto.c -o desto
$ ./desto
$ cat data.dat
Network C programming
```

왼쪽 예제에서 fdopen() 함수 호출을 통해서 쓰기모드의 FILE 구조체 포인터가 반환됨

fileno 함수를 이용한 파일 디스크립터로의 변환

- fileno() 함수

```
#include <stdio.h>

int fileno(FILE* stream);
-> 성공 시 변환된 파일 디스크럽터, 실패 시 -1 반환
```

fileno(fp)
- FILE 포인터 → 파일 디스트립터
  (fp) (fd)

<예제 todes.c>

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

int main()
{
    FILE *fp;
    int fd = open("data.dat", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    if(fd == -1)
    {
        fputs("file open error", stdout);
        return -1;
    }
    printf("First file descriptor: %d\n", fd);
    fp = fdopen(fd, "w");
    fputs("TCP/IP SOCKET PROGRAMMING\n", fp);

    printf("Second file descriptor: %d\n", fileno(fp));
    fclose(fp);
    return 0;
}
```

int fd
FILE *
int fd

실행 결과

```
$ gcc todes.c -o todes
$ ./todes
First file descriptor: 3
Second file descriptor: 3
```

fileno() 함수호출을 통해서 FILE 구조체 포인터를 파일 디스크립터로 변환

소켓 기반에서의 표준 C 입출력 함수의 호출 예

- 표준 C 입출력 함수의 호출 모델: echo_stdclnt.c 파일 일부

```
FILE * readfp;
FILE * writefp;

sock = socket(PF_INET, SOCK_STREAM, 0);
. . .
readfp = fdopen(sock, "r");
writefp = fdopen(sock, "w");

while(1)
{
    fputs("Input message(Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
        break;

    fputs(message, writefp);
    fflush(writefp);
    fgets(message, BUF_SIZE, readfp);
    printf("Message from server: %s", message);
}
```

- 입력용, 출력용 FILE 구조체 포인터를 각각 생성해야 됨
- 표준 C 입출력 함수를 사용할 경우 소켓의 버퍼 이외에 버퍼링이 되기 때문에. 필요하다면, fflush 함수를 직접 호출해야 됨

- 일반적인 순서
  1. 파일 디스크립터를 FILE 구조체 포인터로 변환
  2. 표준 입출력 함수 호출
  3. 함수 호출 후 fflush 함수 호출을 통해 버퍼를 비움

서버: echo_stdserv.c #1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 1024
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    char message[BUF_SIZE];
    int str_len, i;
    struct sockaddr_in serv_adr;
    struct sockaddr_in clnt_adr;
    socklen_t clnt_adr_sz;

    FILE * readfp;
    FILE * writefp;

    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
```

입출력용 파일 구조체 포인터 선언

```c
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    if(serv_sock==-1)
        error_handling("socket() error");

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));

    if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");

    if(listen(serv_sock, 5)==-1)
       error_handling("listen() error");

    clnt_adr_sz=sizeof(clnt_adr);
```

서버: echo_stdserv.c #2

```
    for(i=0; i<5; i++)
    {
        clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
        if(clnt_sock==-1)
            error_handling("accept() error");
        else
            printf("Connected client %d \n", i+1);
        readfp = fdopen(clnt_sock, "r");
        writefp = fdopen(clnt_sock, "w");
        while(!feof(readfp))
        {
            fgets(message, BUF_SIZE, readfp);
            fputs(message, writefp);
            fflush(writefp);
        }
        fclose(readfp);
        fclose(writefp);
    }
    close(serv_sock);
    return 0;
}
void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

입출력용 파일 포인터 생성
(소켓 디스크립터 -> 파일포인터)

```
while((str_len=read(clnt_sock, message, BUF_SIZE))!=0)
    write(clnt_sock, message, str_len);

close(clnt_sock);
```

클라이언트: echo_stdclnt.c #1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 1024
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int sock;
    char message[BUF_SIZE];
    int str_len;
    struct sockaddr_in serv_adr;

    FILE * readfp;
    FILE * writefp;

    if(argc!=3) {
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }
```

입출력용 파일 구조체 포인터 선언

```c
    sock=socket(PF_INET, SOCK_STREAM, 0);
    if(sock==-1)
        error_handling("socket() error");

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
    serv_adr.sin_port=htons(atoi(argv[2]));

    if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
        error_handling("connect() error!");
    else
        puts("Connected.....");
```

클라이언트: echo_stdclnt.c #2

```
    readfp = fdopen(sock, "r");
    writefp = fdopen(sock, "w");

    while(1)
    {
        fputs("Input message(Q to quit): ", stdout);
        fgets(message, BUF_SIZE, stdin);
        if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
            break;

        fputs(message, writefp);
        fflush(writefp);
        fgets(message, BUF_SIZE, readfp);
        printf("Message from server: %s", message);
    }
    fclose(writefp);
    fclose(readfp);
    return 0;
}

void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

입출력용 파일 포인터 생성
(소켓 디스크립터 -> 파일포인터)

```
write(sock, message, strlen(message));
str_len=read(sock, message, BUF_SIZE-1);
message[str_len]=0;
printf("Message from server: %s", message);
```

Questions?