Inter-Process Communication (IPC)

Instructor: Limei Peng
Dept. of CSE, KNU

Outlines

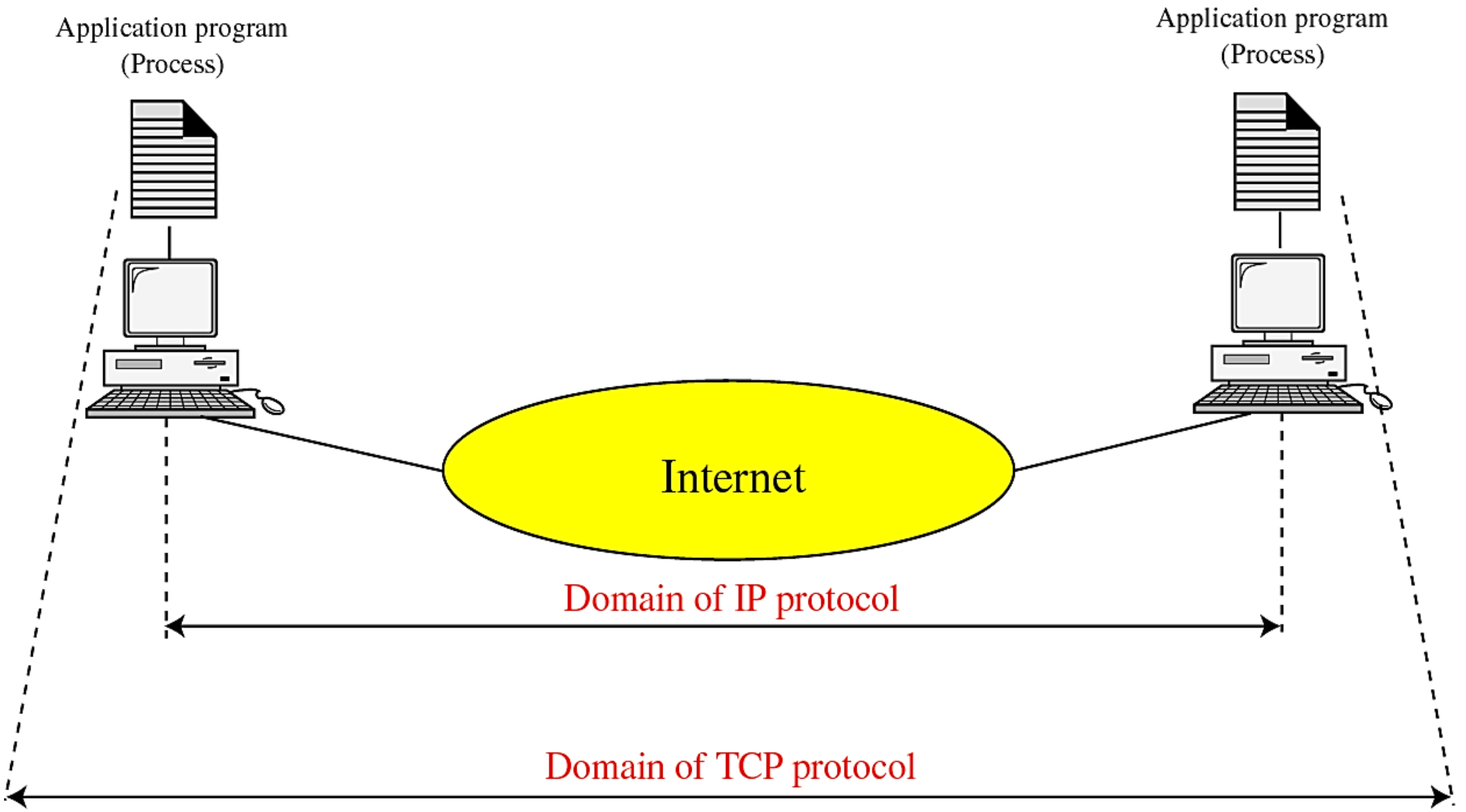
- What is inter-process communication?
- **pipe()**

ⓔ 프로세스끼리는 서로의 정보를 몰라.

Process-to-Process Communication

ⓔ 각 ID와 각 메모리 공간을 갖았다.

- ❑ Host-to-host communication and process-to-process communication

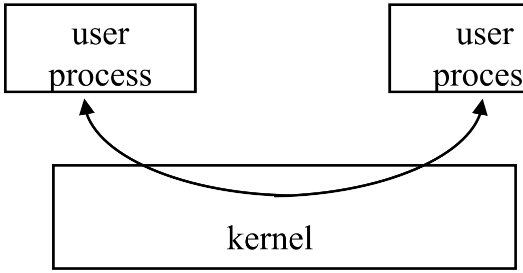


Interprocess Communication (IPC) : 프로세스들 간에 서로 데이터를 주고받는 행위

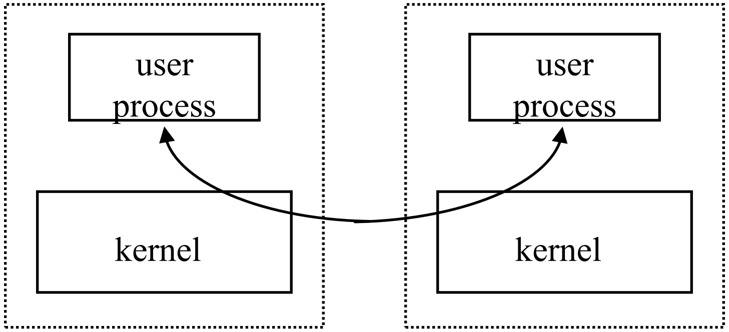
(일반적으로 프로세스는 서로서로 영향을 줄 수 X

- In general, processes cannot influence each other
  - Each process is executed in isolation from the others (isolated memory) : 각각의 프로세스는 서로 독립되어 실행
  - For example, one process cannot write into the memory of another
    (한 프로세스는 다른 프로세스에 메모리에 쓸 수 X)

□ IPC between Two Processes at the same system



□ IPC between Two Systems



Classifications of IPC

여럿 프로세스가 서로서로 소통할 수 있게 만들 수 O

- But you can let several processes communicate with each other via the following methods:
  - Signals: Send a signal (SIGHUP, SIGINT, SIGKILL, etc.) to another process
  - 터널 같은 애 (data 전송 통로)

    PIPE: a communication channel to transfer data
  - FIFOs, Message Queues, Shared memory (the same memory area is accessible to multiple processes), Semaphores (e.g., to regulate access to shared memory)
- Generally, all these IPCs work only between processes of the same computer

PIPE

CH11

PIPE

- ❒ A specific type of file that IPC is supported by OS
  - ❍ Temporal file that is managed by OS, unlike general file
  - ❍ It's used to transfer data between processes not to store data

- ❒ IPC using a PIPE

  보내는애는 PIPE에 write하고
  받는애는 그걸 읽어요.

  - ❍ Sender writes to a PIPE and receiver reads from that
  - ❍ Supports a stream channel
  - ❍ Sent data has a sending order

PIPE

- ❐ Acts as a conduit/channel allowing two processes to communicate : 두 프로세스가 통신할 수 있는 도관/채널 역할

- ❐ Some questions?

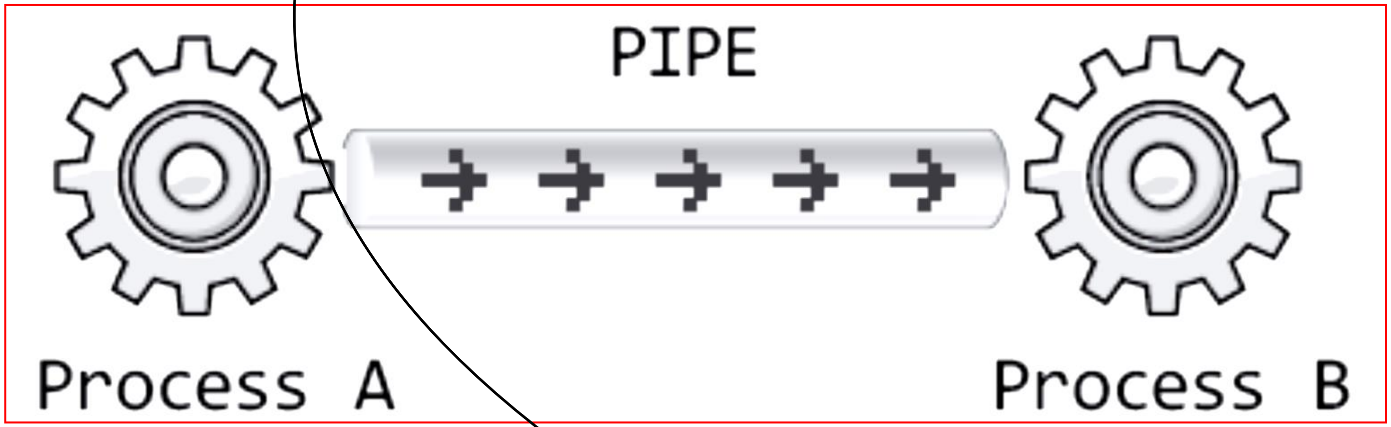
  단방향 통신 vs 양방향 통신

  - Is communication **unidirectional** or **bidirectional**?
  - Must there exist a **relationship** (i.e., parent-child) between the communicating processes?

  소통할 때 관계가 있어야하나?

pipe()-unnamed PIPE

- A simple, **unnamed** pipe provides a one-way flow of data : 동일한 부모 프로세스를 갖는 프로세스들 간에 '단방향 통신' 지원. (named pipe는 any process)
  - Can be thought as a special file that can store a limited amount of data in a first-in-first-out (FIFO) manner, exactly akin to a queue : (큐랑 비슷해) FIFO 구조임 (First in First Out)



→ 제한된 양의 데이터를 FIFO하고 저장할 수 있는 특수 파일로 생각하면 됨.

- Other variations:
  - Stream pipes
  - FIFOs

pipe()-unnamed PIPE

- An unnamed pipe is created by calling **pipe()**, which returns an array of 2 file descriptors (int)
  - The file descriptors are for reading and writing, respectively → read, write

- Unnamed pipes can only be used between **related process**, such as parent/child, or child/child processes : parent/child나 child/child같이
  서로가 관련있는 프로세스 사이에서만 사용가능

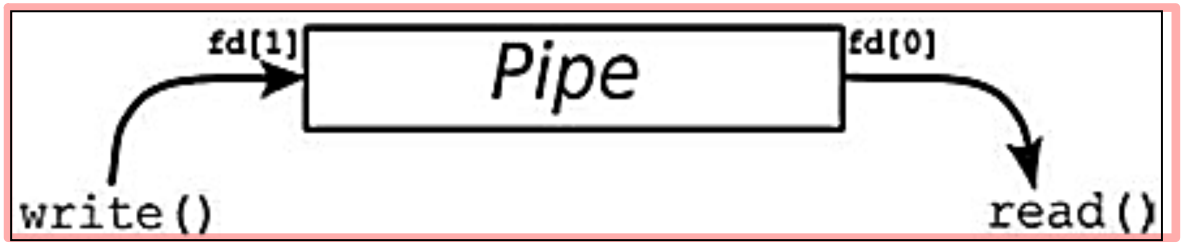
- Unnamed pipes can exist only as long as the processes using them : unnamed 파이프는 프로세스가 사용하는 동안에만 존재할 수 O

pipe() System Call (unnamed)

Creates a unidirectional pipe.

```
#include < unistd.h>
int pipe (int pipefd[2]);
                                   Return 0 on Success; -1 on Failure;
```

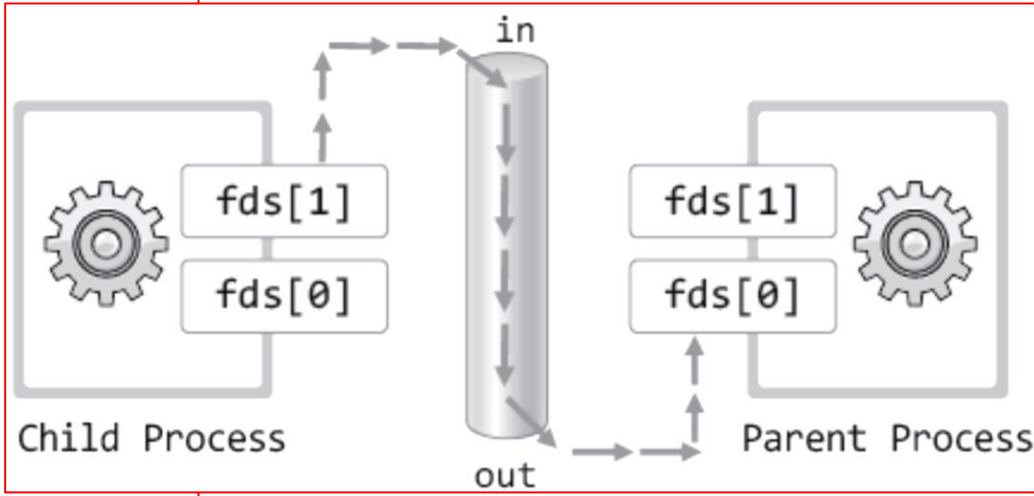
- If successful, the **pipe** system call will **return two** integer file descriptors, **pipefd[0]** and **pipefd[1]**
  - ***pipefd[0]*** is the **read** end from the pipe
  - ***pipefd[1]*** is the **write** end to the pipe
- Example (pipe1.c)
  - Parent/child processes communicating via unnamed pipe



Example #1-pipe1.c

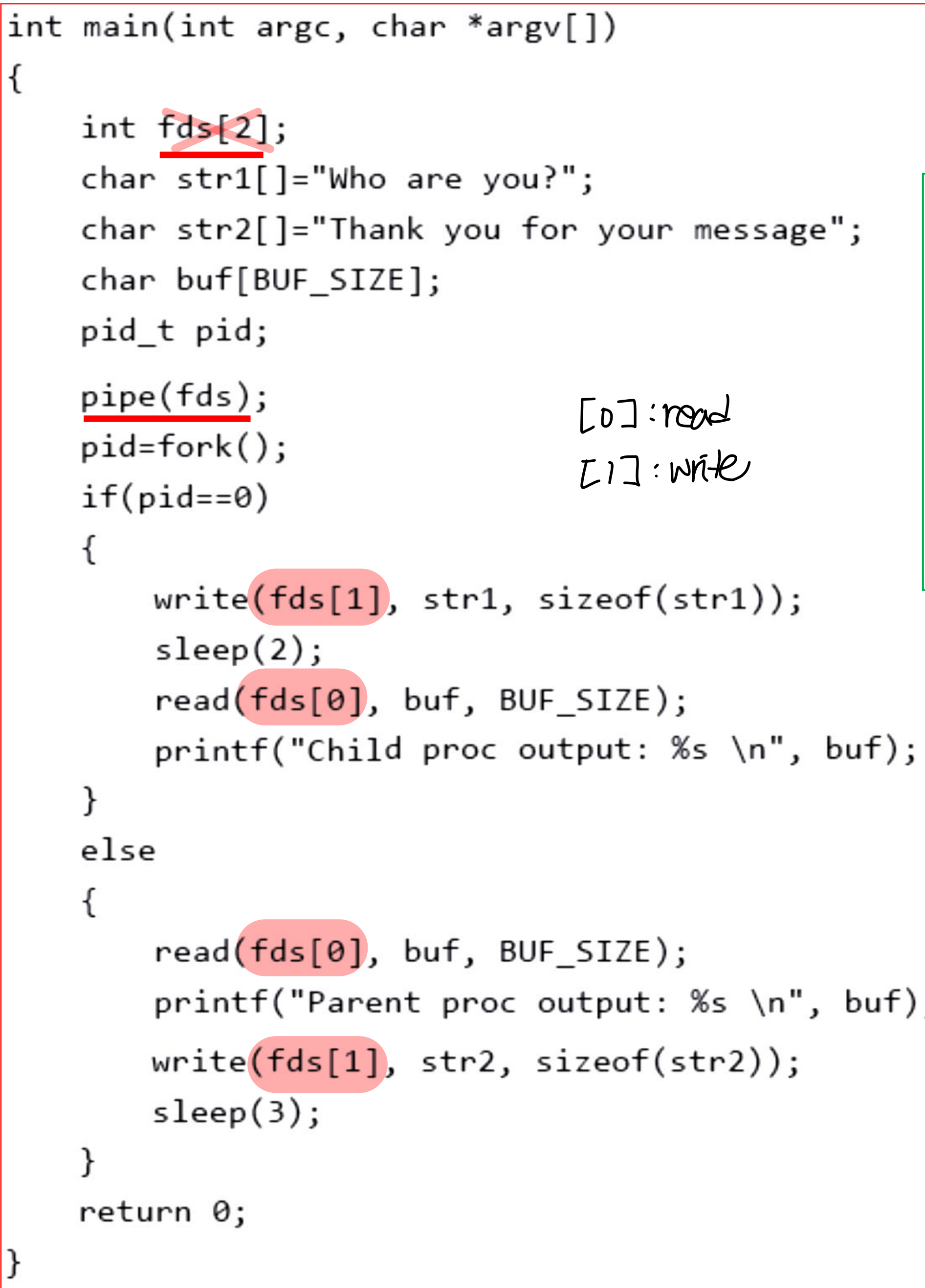
→unidirectional communication

```
int main(int argc, char *argv[])
{
    int fds[2];
    char str[]="Who are you?";
    char buf[BUF_SIZE];
    pid_t pid;

    pipe(fds);                  [0]:read
                                [1]:write
    pid=fork();
    if(pid==0) : child process
    {
        write(fds[1], str, sizeof(str));
    }
    else : parent process
    {
        read(fds[0], buf, BUF_SIZE);
        puts(buf);
    }
    return 0;
}
```



```
root@my_linux:/tcpip# gcc pipe1.c -o pipe1
root@my_linux:/tcpip# ./pipe1
Who are you?
```

Example #2-pipe2.c (bidirectional)

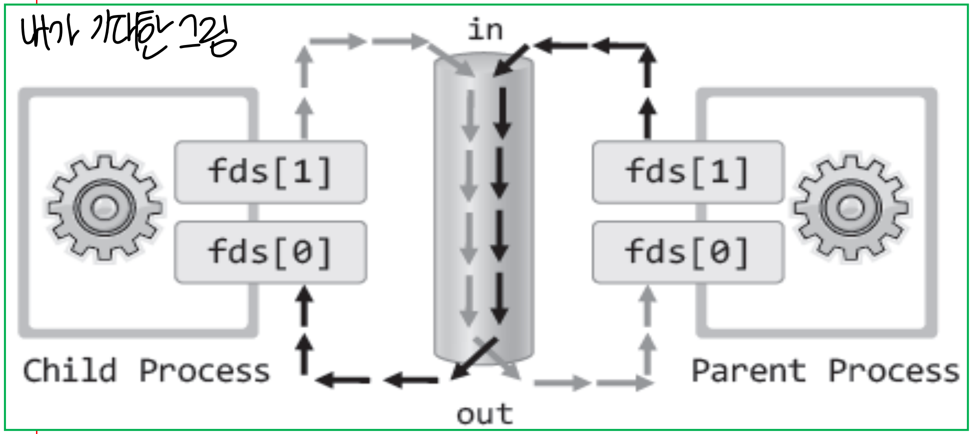


```
int main(int argc, char *argv[])
{
    int fds[2];
    char str1[]="Who are you?";
    char str2[]="Thank you for your message";
    char buf[BUF_SIZE];
    pid_t pid;
    pipe(fds);
    pid=fork();
    if(pid==0)
    {
        write(fds[1], str1, sizeof(str1));
        sleep(2);
        read(fds[0], buf, BUF_SIZE);
        printf("Child proc output: %s \n", buf);
    }
    else
    {
        read(fds[0], buf, BUF_SIZE);
        printf("Parent proc output: %s \n", buf);
        write(fds[1], str2, sizeof(str2));
        sleep(3);
    }
    return 0;
}
```

[0] : read

[1] : write

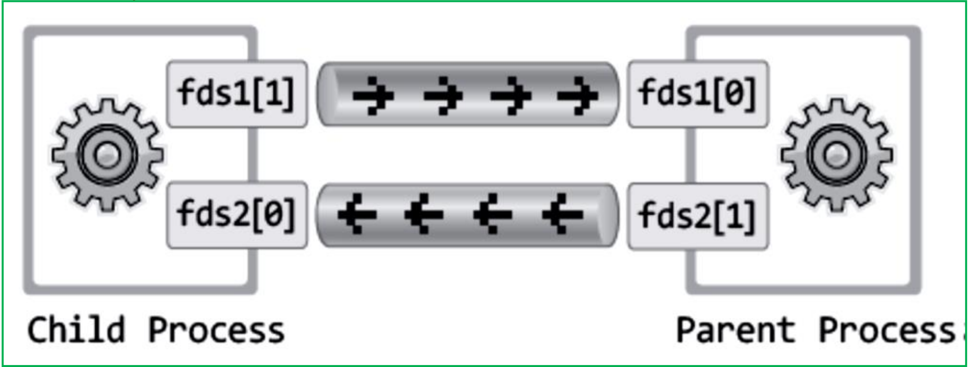
Bad Example!! Why?

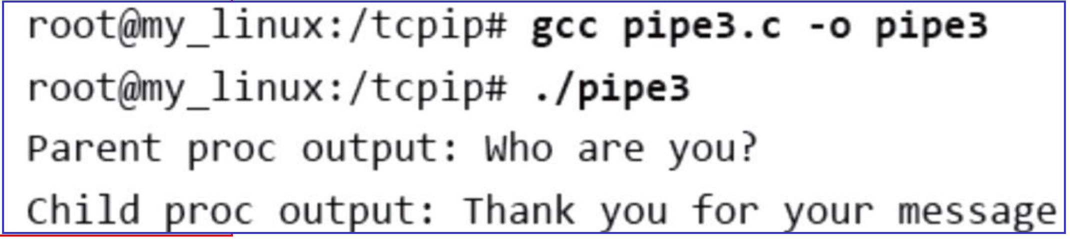


내가 기대한 그림

Example #3-pipe3.c (bidirectional)-Good!

```
int main(int argc, char *argv[])
{
    int fds1[2], fds2[2];
    char str1[]="Who are you?";
    char str2[]="Thank you for your message";
    char buf[BUF_SIZE];
    pid_t pid;

    pipe(fds1), pipe(fds2);
    pid=fork();
    if(pid==0)
    {
        write(fds1[1], str1, sizeof(str1));
        read(fds2[0], buf, BUF_SIZE);
        printf("Child proc output: %s \n", buf);
    }
    else
    {
        read(fds1[0], buf, BUF_SIZE);
        printf("Parent proc output: %s \n", buf);
        write(fds2[1], str2, sizeof(str2));
        sleep(3);
    }
    return 0;
}
```



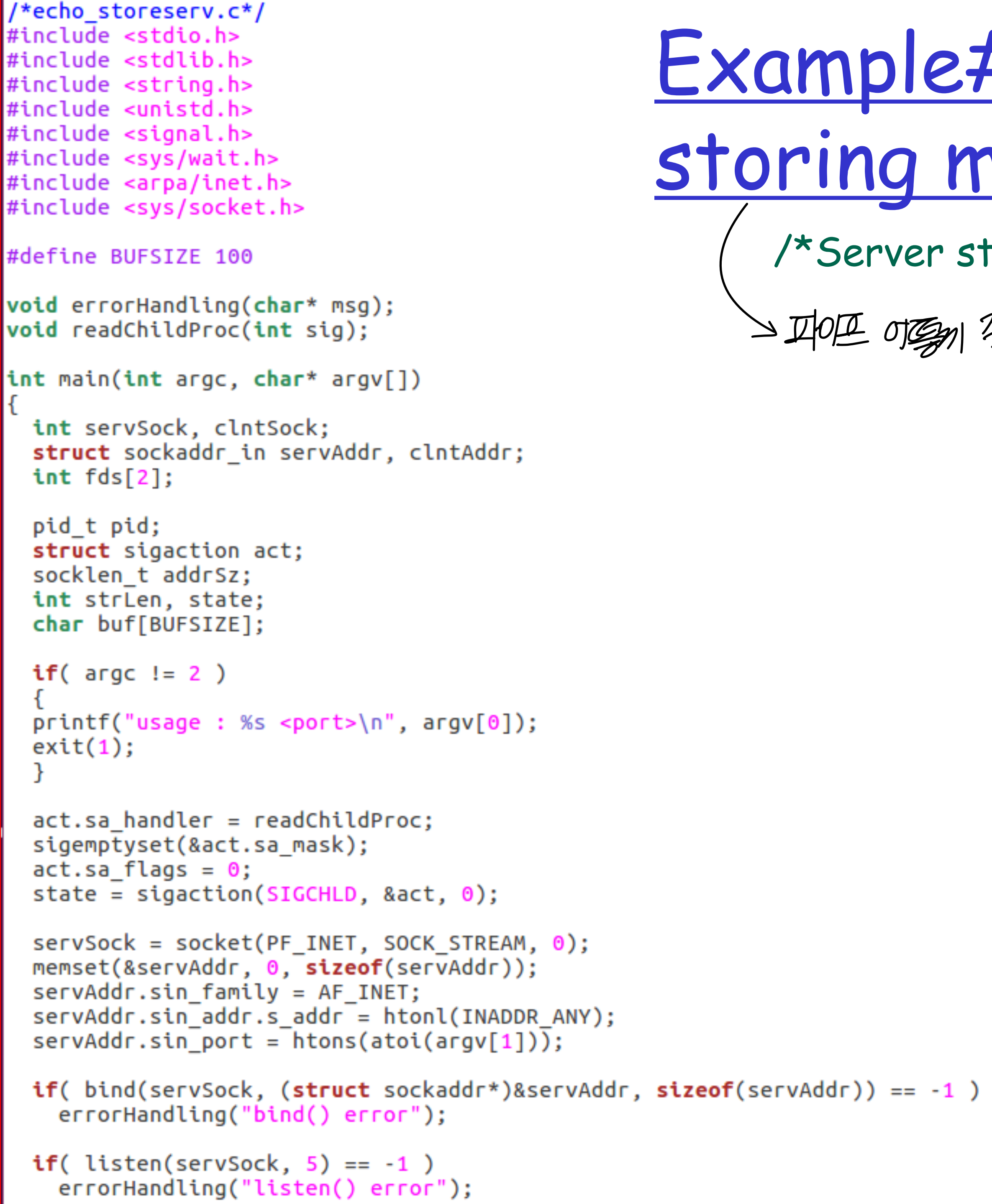


```
root@my_linux:/tcpip# gcc pipe3.c -o pipe3
root@my_linux:/tcpip# ./pipe3
Parent proc output: Who are you?
Child proc output: Thank you for your message
```

Example#4 Echo server storing message (1/2)

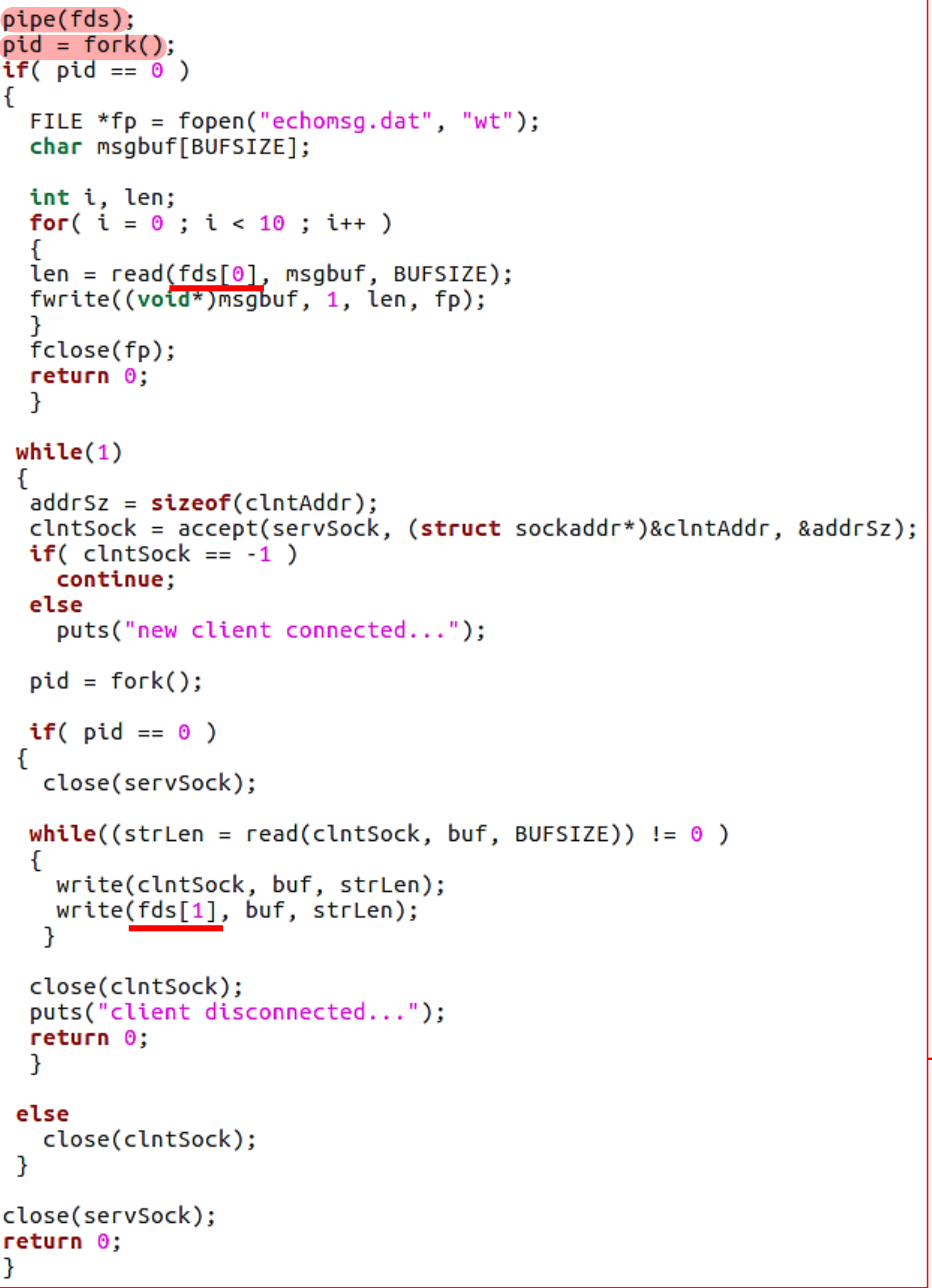
/*Server stores all strings sent by client*/

→ 파이프 어떻게 적용할지

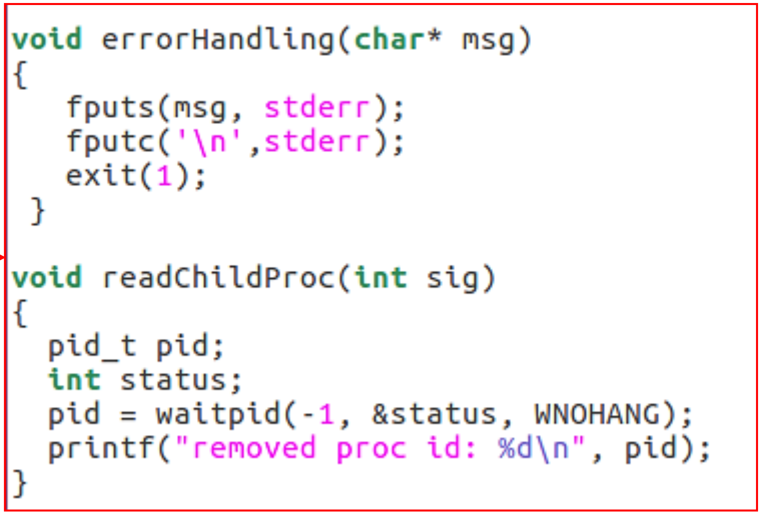


```c
/*echo_storeserv.c*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUFSIZE 100

void errorHandling(char* msg);
void readChildProc(int sig);

int main(int argc, char* argv[])
{
  int servSock, clntSock;
  struct sockaddr_in servAddr, clntAddr;
  int fds[2];

  pid_t pid;
  struct sigaction act;
  socklen_t addrSz;
  int strLen, state;
  char buf[BUFSIZE];

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

  act.sa_handler = readChildProc;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  state = sigaction(SIGCHLD, &act, 0);

  servSock = socket(PF_INET, SOCK_STREAM, 0);
  memset(&servAddr, 0, sizeof(servAddr));
  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(atoi(argv[1]));

  if( bind(servSock, (struct sockaddr*)&servAddr, sizeof(servAddr)) == -1 )
    errorHandling("bind() error");

  if( listen(servSock, 5) == -1 )
    errorHandling("listen() error");
```

Example#4 Echo server storing message (2/2)

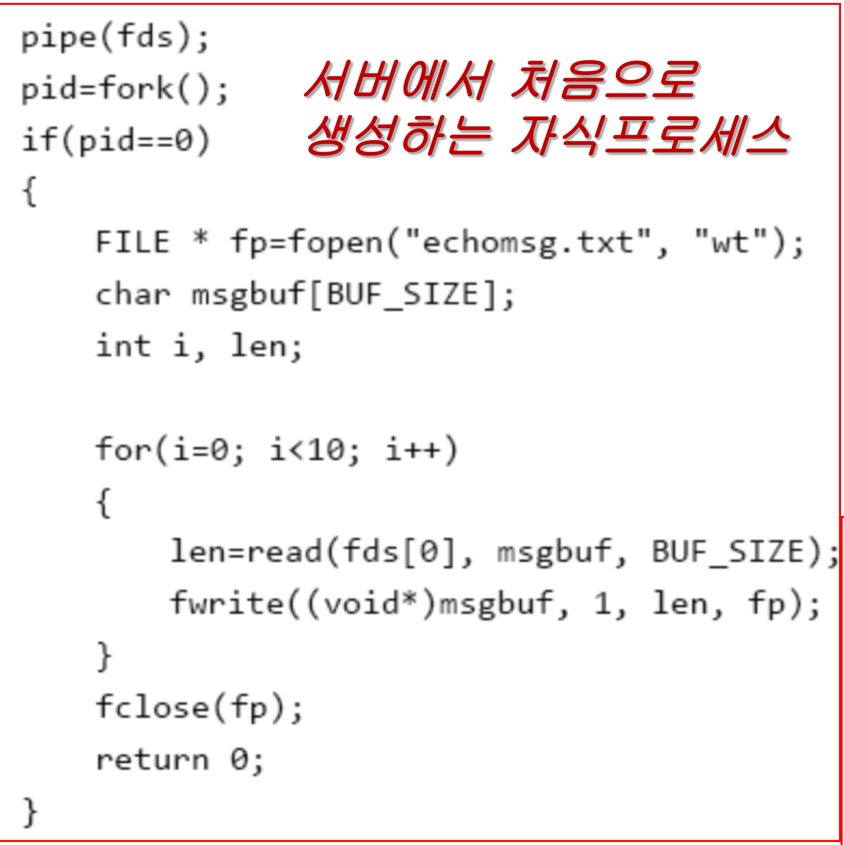


```
pipe(fds);
pid = fork();
if( pid == 0 )
{
  FILE *fp = fopen("echomsg.dat", "wt");
  char msgbuf[BUFSIZE];

  int i, len;
  for( i = 0 ; i < 10 ; i++ )
  {
  len = read(fds[0], msgbuf, BUFSIZE);
  fwrite((void*)msgbuf, 1, len, fp);
  }
  fclose(fp);
  return 0;
  }

 while(1)
 {
  addrSz = sizeof(clntAddr);
  clntSock = accept(servSock, (struct sockaddr*)&clntAddr, &addrSz);
  if( clntSock == -1 )
    continue;
  else
    puts("new client connected...");

  pid = fork();

  if( pid == 0 )
 {
   close(servSock);

  while((strLen = read(clntSock, buf, BUFSIZE)) != 0 )
  {
    write(clntSock, buf, strLen);
    write(fds[1], buf, strLen);
   }

  close(clntSock);
  puts("client disconnected...");
  return 0;
  }

 else
   close(clntSock);
 }

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



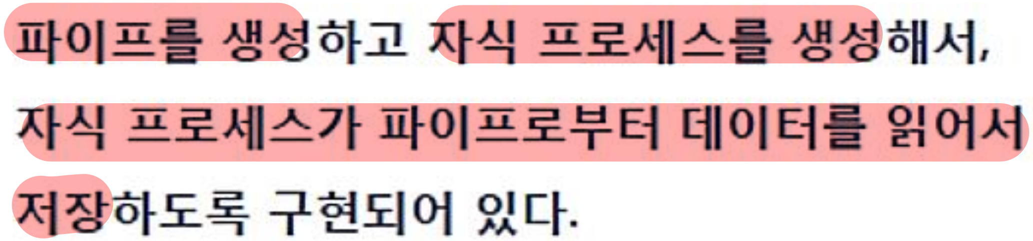
```
void errorHandling(char* msg)
{
   fputs(msg, stderr);
   fputc('\n',stderr);
   exit(1);
 }

void readChildProc(int sig)
{
  pid_t pid;
  int status;
  pid = waitpid(-1, &status, WNOHANG);
  printf("removed proc id: %d\n", pid);
}
```

Example#4 Echo server-additional explanations

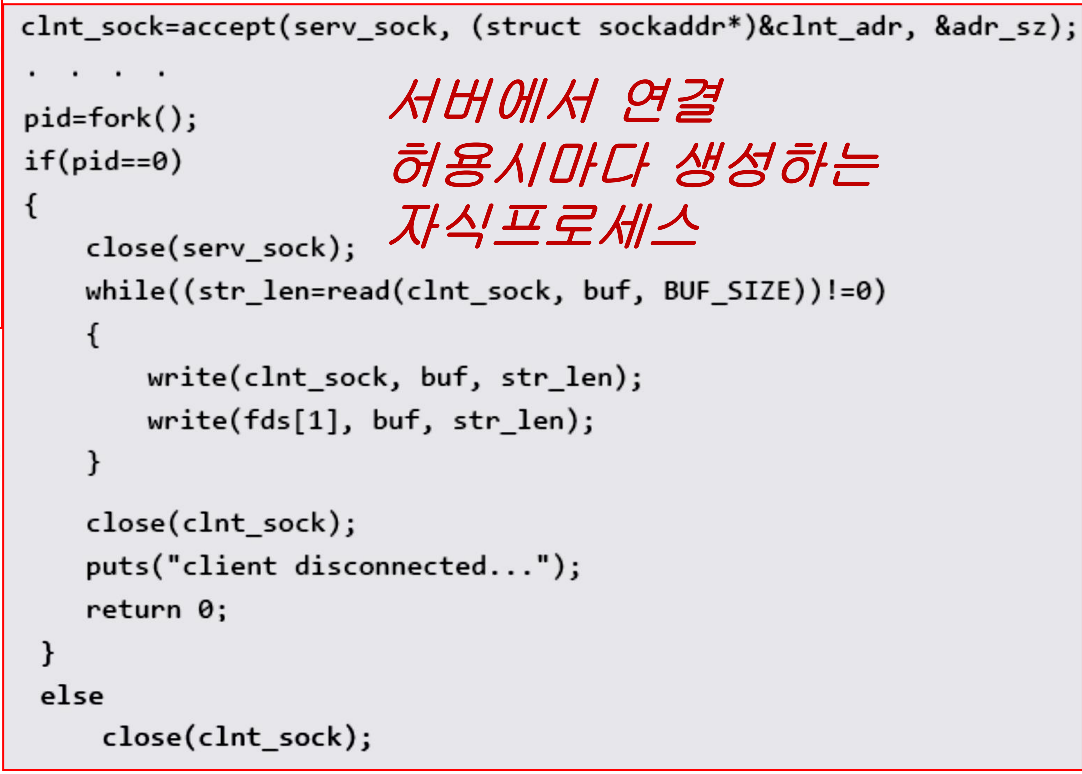


```
pipe(fds);
pid=fork();
if(pid==0)
{
    FILE * fp=fopen("echomsg.txt", "wt");
    char msgbuf[BUF_SIZE];
    int i, len;

    for(i=0; i<10; i++)
    {
        len=read(fds[0], msgbuf, BUF_SIZE);
        fwrite((void*)msgbuf, 1, len, fp);
    }
    fclose(fp);
    return 0;
}
```

서버에서 처음으로
생성하는 자식프로세스

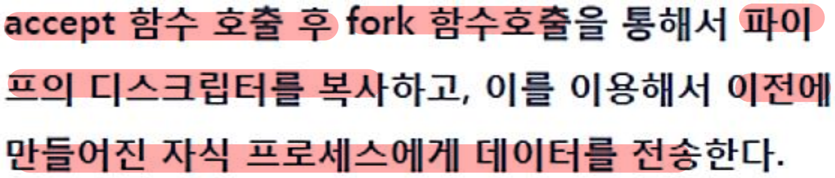


파이프를 생성하고 자식 프로세스를 생성해서, 자식 프로세스가 파이프로부터 데이터를 읽어서 저장하도록 구현되어 있다.



```
clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
. . . .
pid=fork();
if(pid==0)
{
    close(serv_sock);
    while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
    {
        write(clnt_sock, buf, str_len);
        write(fds[1], buf, str_len);
    }

    close(clnt_sock);
    puts("client disconnected...");
    return 0;
 }
 else
     close(clnt_sock);
```

서버에서 연결
허용시마다 생성하는
자식프로세스



accept 함수 호출 후 fork 함수호출을 통해서 파이프의 디스크립터를 복사하고, 이를 이용해서 이전에 만들어진 자식 프로세스에게 데이터를 전송한다.