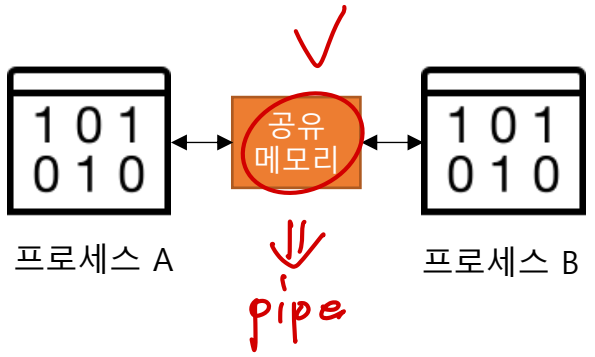
Chapter 11

프로세스간 통신
(Inter Process Communication)

IPC

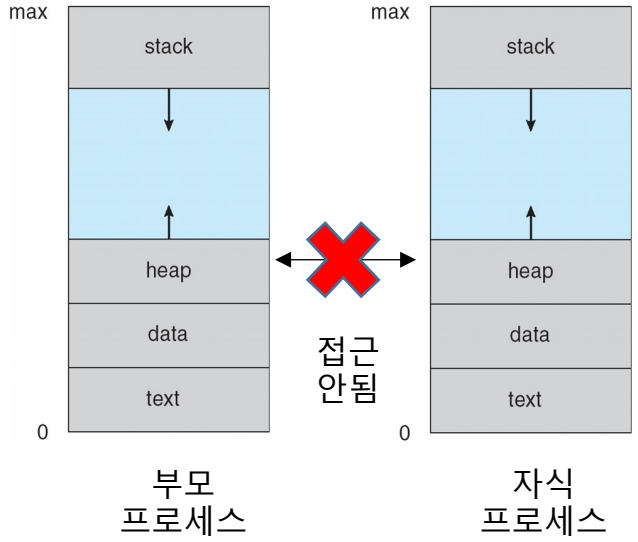
프로세스간 통신의 기본 이해

- 프로세스간 통신
  - 두 프로세스 사이에서 데이터 전달
    - 두 프로세스가 공유하는 메모리가 존재해야 됨



- 프로세스간 통신의 어려움
  - 모든 프로세스는 자신만의 메모리 공간을 독립적으로 구성
    - fork()함수 호출로 생성된 자식 프로세스도 부모 프로세스와 메모리를 공유하지 않음
  - 프로세스 A는 프로세스 B의 메모리 공간에 접근이 불가능

- 파이프(pipe)
  - 운영체제가 별도의 메모리 공간을 마련
  - 파이프를 이용한 프로세스간 통신이 가능
    - 파이프는 운영체제에 속한 자원

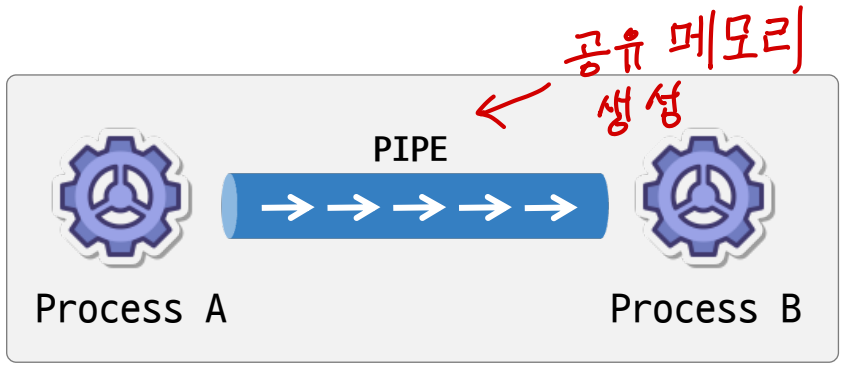


파이프 기반의 프로세스간 통신

- 파이프 생성 함수:pipe()

```
#include <unistd.h>

int pipe(int filedes[2]);
-> 성공 시 0, 실패 시 -1 반환
```

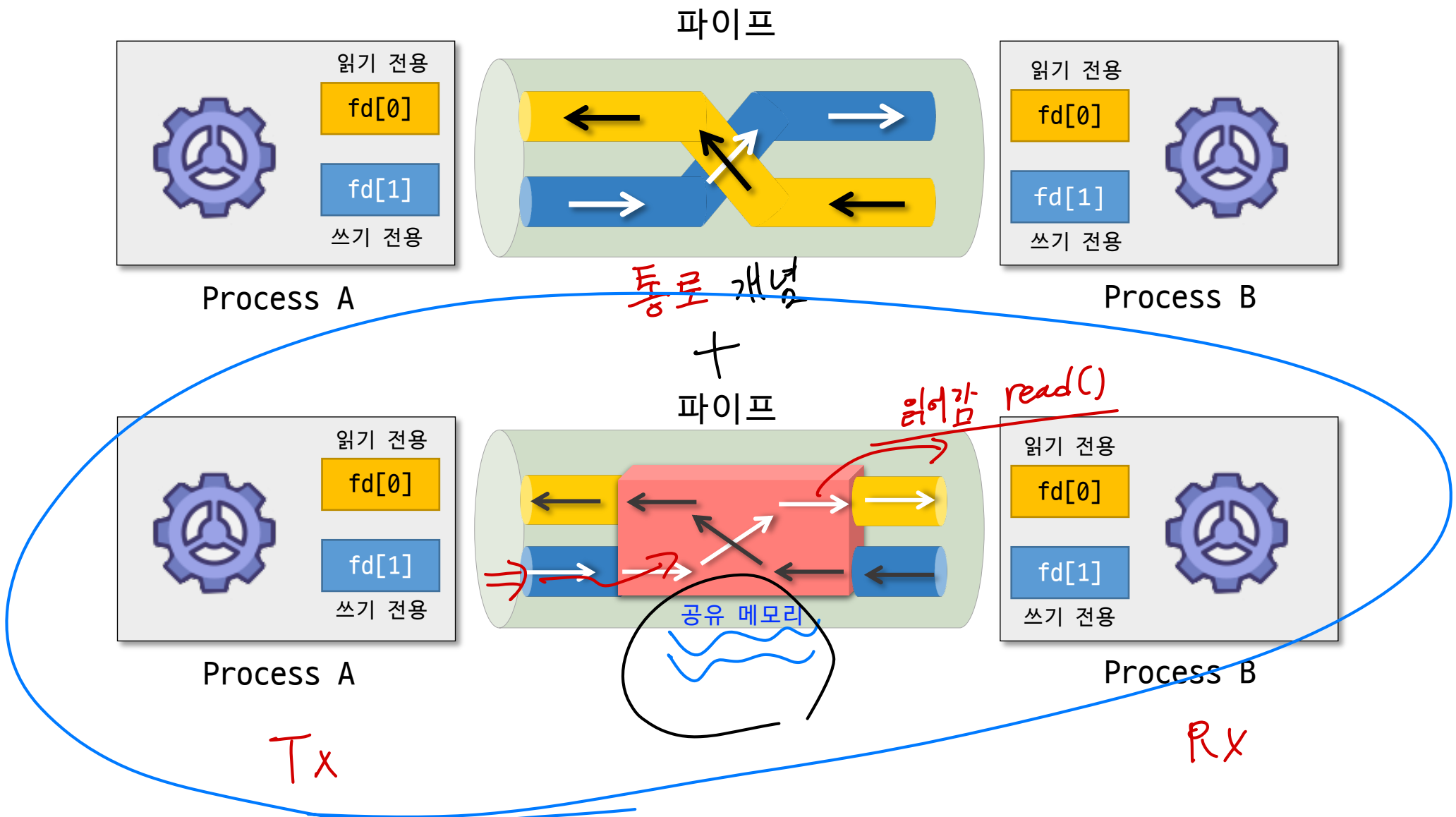
  - filedes[0]: 데이터 수신용 파일 디스크립터 저장 (읽기 전용)
  - filedes[1]: 데이터 송신용 파일 디스크립터 저장 (쓰기 전용)



PIPE 기반 프로세스 통신 모델

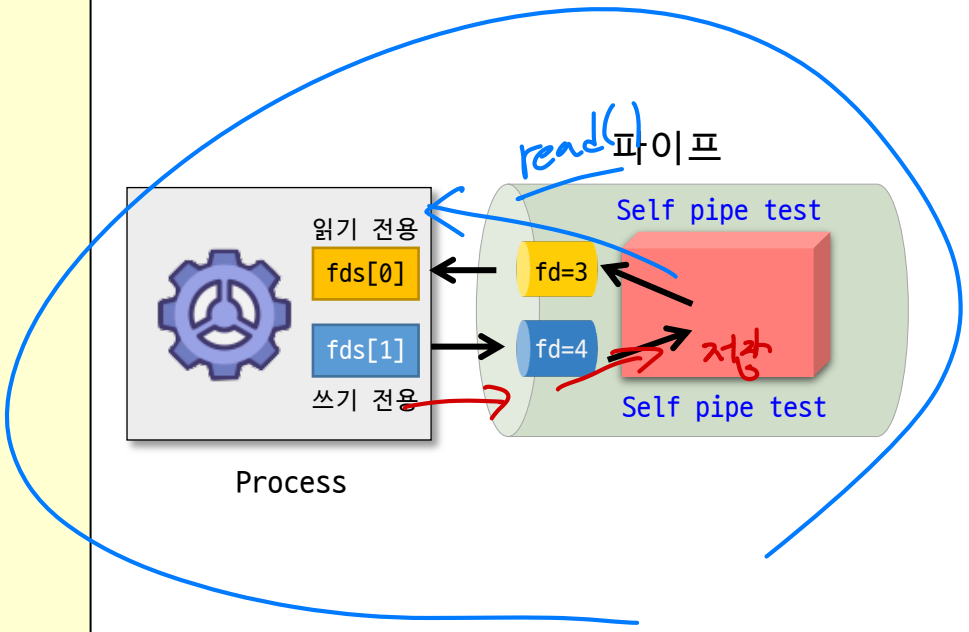
- pipe() 함수가 호출되면, 운영체제는 서로 다른 프로세스가 함께 접근할 수 있는 공유 메모리 공간을 생성
- 공유 메모리 공간(pipe)에 접근하기 위해 파일 디스크립터를 사용
- 프로세스는 pipe의 파일 디스크립터만 가지고 있음

파이프 통신 개념 그림


파이프
읽기 전용
fd[0]
fd[1]
쓰기 전용
Process A
Process B
통로 개념
+
파이프
읽어감 read()
공유 메모리
Tx
Rx

하나의 프로세스에서 파이프 사용: pipe0.c

```c
#include <stdio.h>
#include <unistd.h>
#define BUF_SIZE 30

int main(int argc, char *argv[])
{
    int fds[2];
    char str[]="Self pipe test";
    char buf[BUF_SIZE];
    pid_t pid;

    pipe(fds); // 파이프 생성
    printf("fds[0]=%d, fds[1]=%d\n", fds[0], fds[1]);

    // fds[1]을 통해 쓰기
    write(fds[1], str, sizeof(str));

    // fds[0]을 통해 읽기
    read(fds[0], buf, BUF_SIZE);
    puts(buf);
    return 0;
}
```



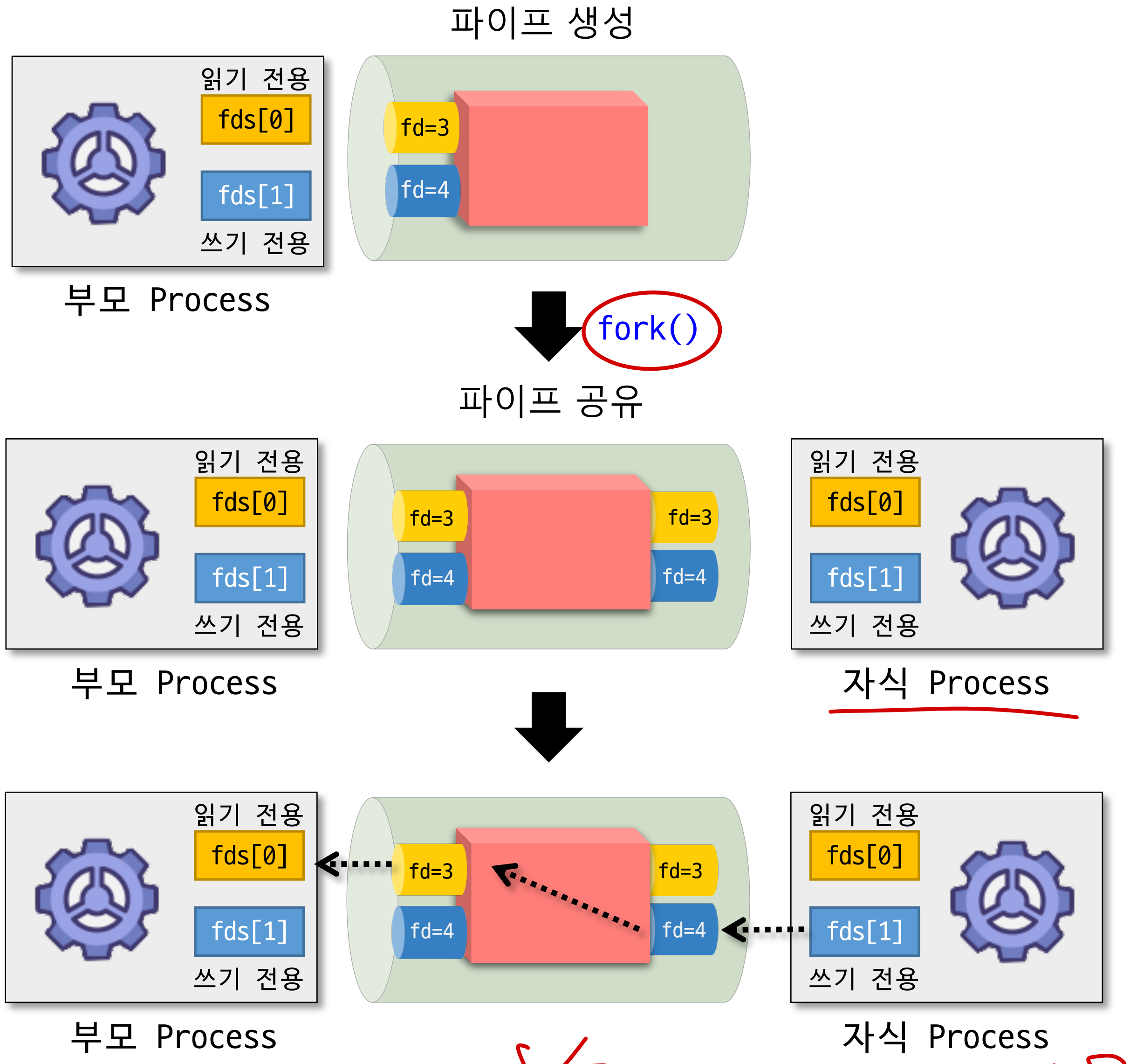

```
$ gcc pipe0.c -o pipe0
$ ./pipe0
fds[0]=3, fds[1]=4
Self pipe test
```

파이프 생성 및 파이프 공유: pipe1.c

- 파이프 생성
  - pipe(fds)

- fork() 함수 호출
- 파이프 공유
  - 파일 디스크립터 복사

- 데이터 전송
  - 자식: 쓰기
  - 부모: 일기



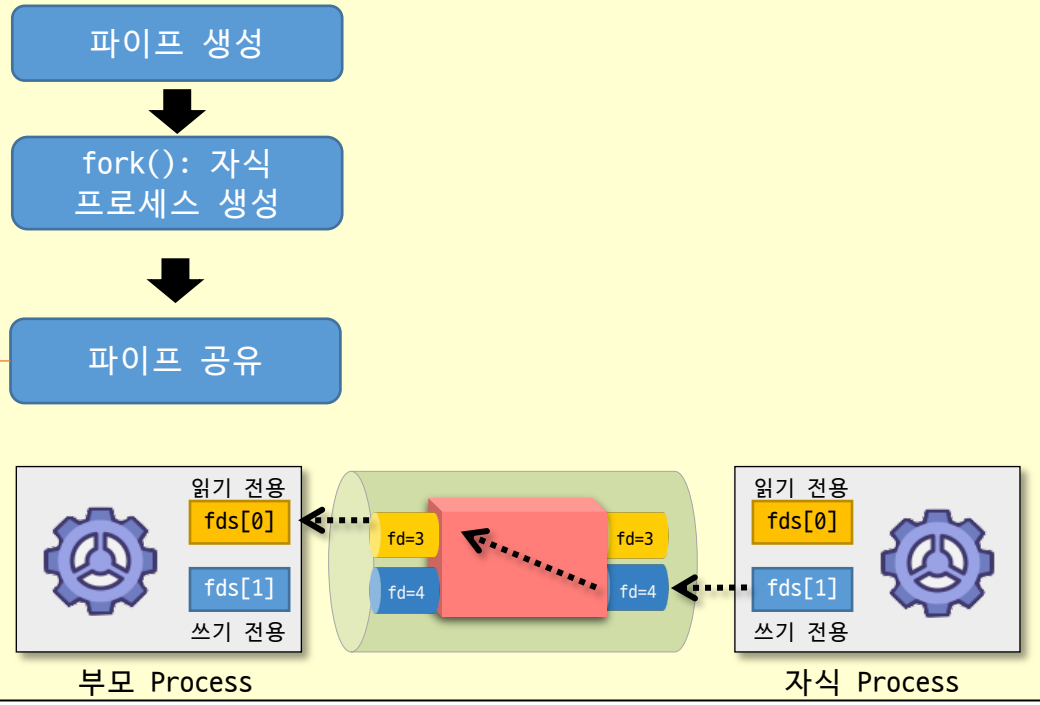
✓ 상당히 위험한 코드.
why?

✓ 공간 빼서 만들어서 둘 사이 공유한다!

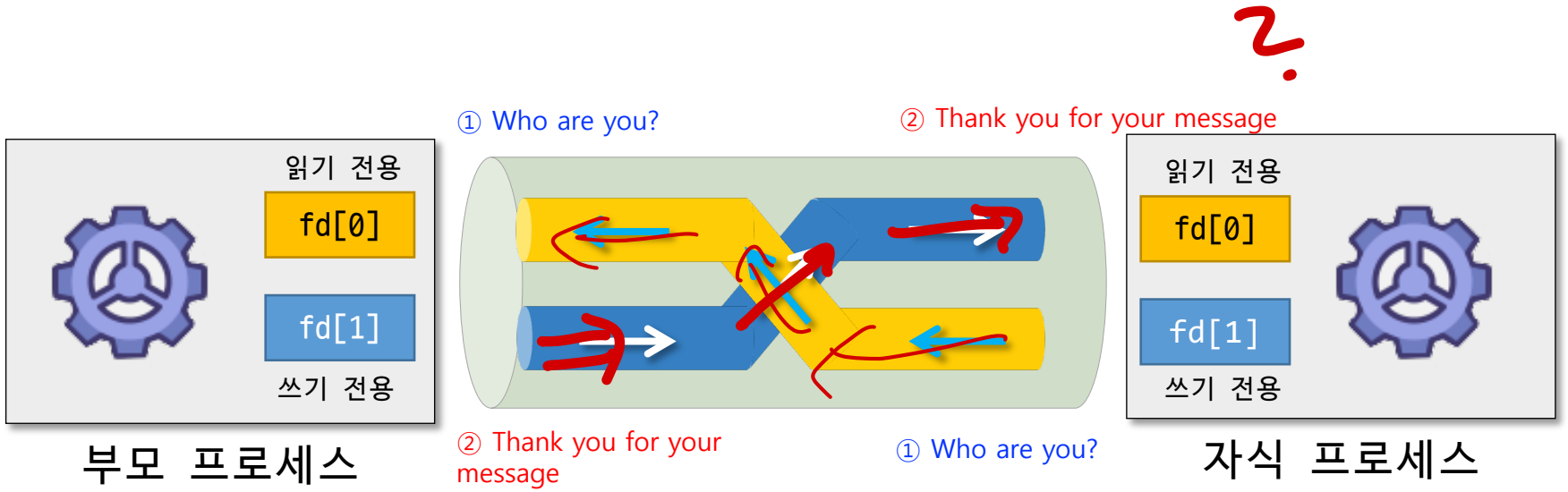
파이프 생성 및 통신 예제: pipe1.c

```c
#include <stdio.h>
#include <unistd.h>
#define BUF_SIZE 30

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

    pipe(fds); // 파이프 생성
    printf("fds[0]=%d, fds[1]=%d\n", fds[0], fds[1]);

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

```
$ gcc pipe1.c -o pipe1
$ ./pipe1
fds[0]=3, fds[1]=4
Who are you?
```



하나의 파이프를 이용한 양방향 통신



- 하나의 파이프를 이용한 양방향 통신
  - ✓ 자식과 부모 프로세스가 교대로 read/write 과정을 수행
  - ✓ 타이밍이 매우 중요
  - ✓ 타이밍 관리는 불가능함

하나의 파이프를 이용한 양방향 통신: pipe2.c (실습)

```c
#include <stdio.h>
#include <unistd.h>
#define BUF_SIZE 30

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
    {
        sleep(2);
        read(fds[0], buf, BUF_SIZE);
        printf("Parent proc output: %s \n", buf);
        write(fds[1], str2, sizeof(str2));
        sleep(3);
    }

    return 0;
}
```

⇒ 파이프 생성 + 파이프 공유

TX

sleep(2) 주석 처리 후 결과 확인

단순히 sleep() 함수 호출로 두 프로세스의 타이밍 관리

① Who are you?

② Thank you for your message

sleep(3)
- 부모 프로세스가 먼저 종료되는 것을 막기 위함

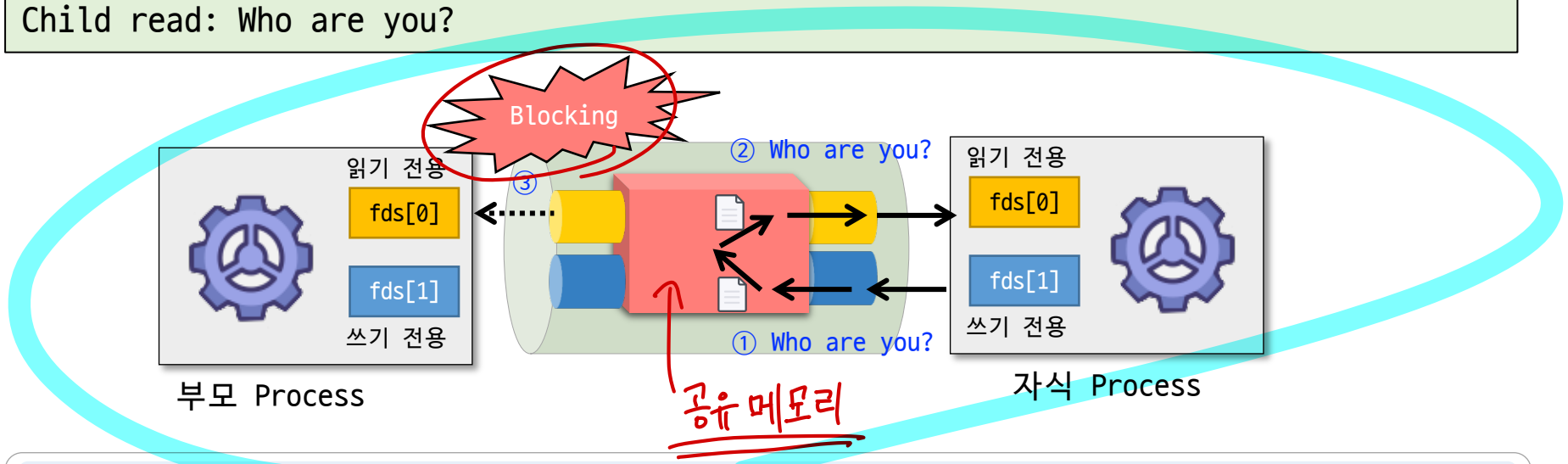
실행 결과

```
$ ./pipe2

Parent read: Who are you?
Child read: Thank you for your message
```

실행 결과: sleep(2) 주석 처리

- 비정상 동작: sleep(2) 주석 처리(18라인)

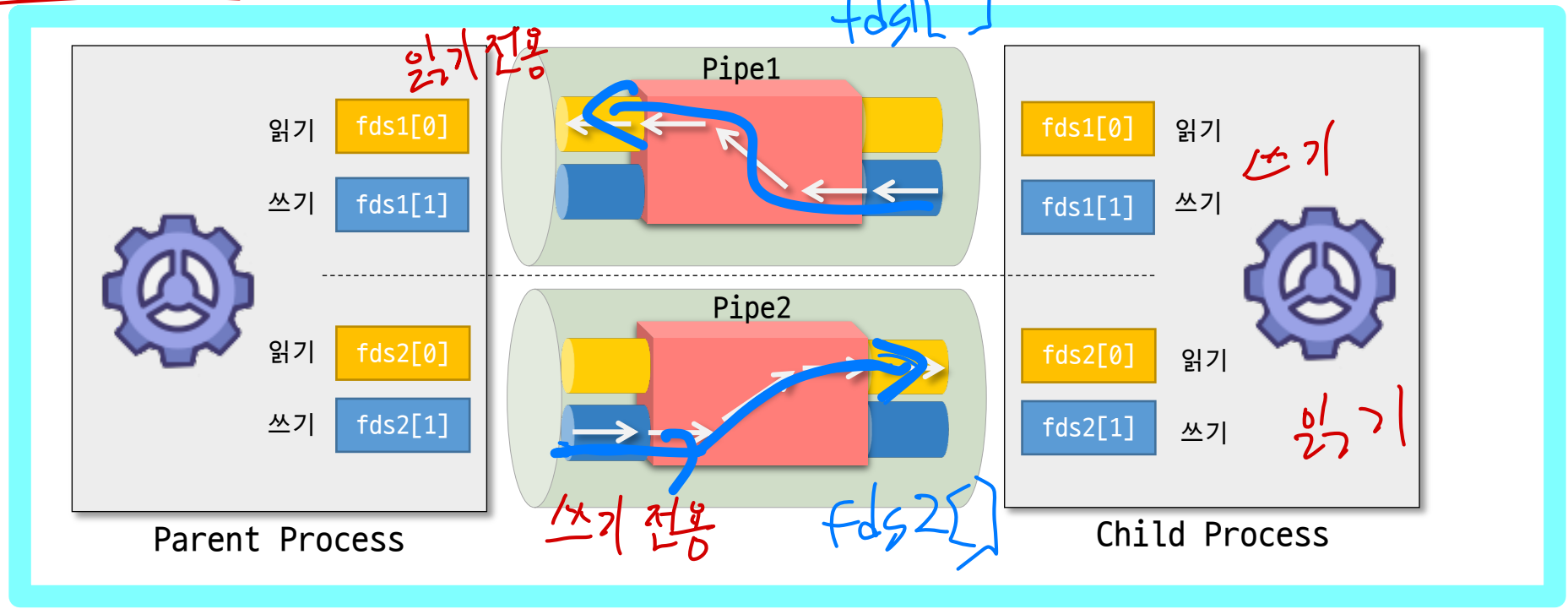
```
$ ./pipe2

Child read: Who are you?
```



- 파이프 데이터 전달
  - ✓ read() 함수를 먼저 호출하는 프로세스에게 데이터가 전달됨
  - ✓ 자식 process가 먼저 읽음
- 부모 프로세스
  - ✓ read() 함수를 호출하고 나서, 파이프에 데이터가 들어오기 만을 기다림
  - ✓ 자식 프로세스가 이미 읽었기 때문에 blocking 상태에 빠짐

양방향 통신을 위한 두 개의 파이프 사용

- 두 개의 파이프 사용
  - read()/write() 과정을 각각 다른 파이프에서 처리



- 양방향 통신을 위해 두 개의 파이프 생성
  - ✓ 쓰기 전용, 읽기 전용
  - ✓ 입출력 타이밍에 따라 데이터 흐름에 영향을 받지 않음

두 개의 파이프 사용 예제: pipe3.c (실습)

```c
#include <stdio.h>
#include <unistd.h>
#define BUF_SIZE 30

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;
}
```

두 개의 파이프 생성

read()와 write()함수가 서로 다른 파이프를 사용

```
$ ./pipe3

Parent proc read: Who are you?
Child Proc read: Thank you for your message.
```

양방향 pipe 통신 예제: pipe4.c #1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define BUF_SIZE 10

int main()
{
    int pipe_parent[2];
    int pipe_child[2];
    char wbuf[BUF_SIZE], rbuf[BUF_SIZE];
    int count = 100;
    int len = 0;
    pid_t pid;

    memset(wbuf, 0, sizeof(BUF_SIZE));
    memset(rbuf, 0, sizeof(BUF_SIZE));
```

```c
if(pipe(pipe_parent) == -1)
{
    printf("Parent pipe 생성 실패\n");
    exit(1);
}

if(pipe(pipe_child) == -1)
{
    printf("Child pipe 생성 실패\n");
    exit(1);
}

pid = fork();

if(pid == -1)
{
    printf("자식 프로세스 생성 실패\n");
    return -1;
}
```

첫 번째 파이프 생성

두 번째 파이프 생성

파이프 공유

양방향 pipe 통신 예제: pipe4.c #2

```c
if(pid == 0)
{
    while(1)
    {
        count += 2;

        len = sprintf(wbuf, "%d", count);
        wbuf[len] = '\0 ' ;
        write(pipe_child[1], wbuf, strlen(wbuf));

        read(pipe_parent[0], rbuf, BUF_SIZE);
        sscanf(rbuf, "%d", &count);
        printf("<Child process> count= %d\n", count);
        sleep(1);
    }
}
```

자식 프로세스는 count 값을 2씩 증가

```c
else
{
    while(1)
    {
        read(pipe_child[0], rbuf, BUF_SIZE);
        sscanf(rbuf, "%d", &count);
        printf("[Parent process] count= %d\n", count);

        count -= 1;

        len = sprintf(wbuf, "%d", count);
        wbuf[len] = '\0 ' ;
        write(pipe_parent[1], wbuf, strlen(wbuf));
        sleep(1);
    }
}
return 0;
}
```

부모 프로세스는 count 값을 1씩 감소

실행 결과

```
$ ./pipe4
[Parent process] count= 102
<Child process> count= 101
[Parent process] count= 103
<Child process> count= 102
[Parent process] count= 104
<Child process> count= 103
[Parent process] count= 105
<Child process> count= 104
[Parent process] count= 106
<Child process> count= 105
[Parent process] count= 107
<Child process> count= 106
[Parent process] count= 108
<Child process> count= 107
[Parent process] count= 109
<Child process> count= 108
[Parent process] count= 110
<Child process> count= 109
[Parent process] count= 111
<Child process> count= 110
[Parent process] count= 112
<Child process> count= 111
[Parent process] count= 113
<Child process> count= 112
^C
```

메시지를 저장하는 에코 서버 동작

echo_storeserv.c 일부

```
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;
}
```

파이프 생성 및 자식 프로세스와 파이프를 공유

- 클라이언트 데이터를 파이프로 전달
- 수신 데이터를 파일로 저장

- accept() 함수 호출 후 fork() 함수를 호출하여 파이프 디스크립터를 복사
- 이를 이용하여 이전에 생성한 자식 프로세스에게 데이터를 전송

- 파이프를 생성하고 자식 프로세스를 생성
- 자식 프로세스가 파이프로부터 데이터를 읽어 파일에 저장

```
clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
if(clnt_sock==-1)
    continue;
else
    puts("new client connected...");

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 클라이언트의 연결 허용마다 자식 프로세스 생성

pipe

echo_storeserv.c #1

```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 BUF_SIZE 100
void error_handling(char *message);
void read_childproc(int sig);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_adr, clnt_adr;
    int fds[2];
    pid_t pid;
    struct sigaction act;
    socklen_t adr_sz;
    int str_len, state;
    char buf[BUF_SIZE];
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
```

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

    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    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");

    pipe(fds);
    pid=fork();
```

echo_storeserv.c #2

```
    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;
    }

    while(1)
    {
        adr_sz=sizeof(clnt_adr);
        clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr,
                           &adr_sz);
        if(clnt_sock==-1)
            continue;
        else
            puts("new client connected...");

        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);
    }
    close(serv_sock);
    return 0;
}

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

실행 결과

- 서버: echo_storeserv.c
- 클라이언트: echo_mpclient.c (10장)

서버

```
$ gcc echo_storeserv.c -o server
$ ./server 9190
new client connected...
new client connected...
removed proc id: 237469
client disconnected...
removed proc id: 237472
client disconnected...
removed proc id: 237533
```

파일 저장

echomsg.txt

```
Hi
Hello
Pipe test
One
Two
Threee
Three
hi
```

클라이언트 #1

```
$ gcc echo_mpclient.c -o client
$ ./client 127.0.0.1 9190
Hello
Message from server: Hello
Pipe test
Message from server: Pipe test
Two
Message from server: Two
Threee
Message from server: Threee
```

클라이언트 #2

```
$ ./client 127.0.0.1 9190
Hi
Message from server: Hi
One
Message from server: One
Three
Message from server: Three
```

Questions?