

실습 12

IPC: Interprocess Communication Clients and Signal

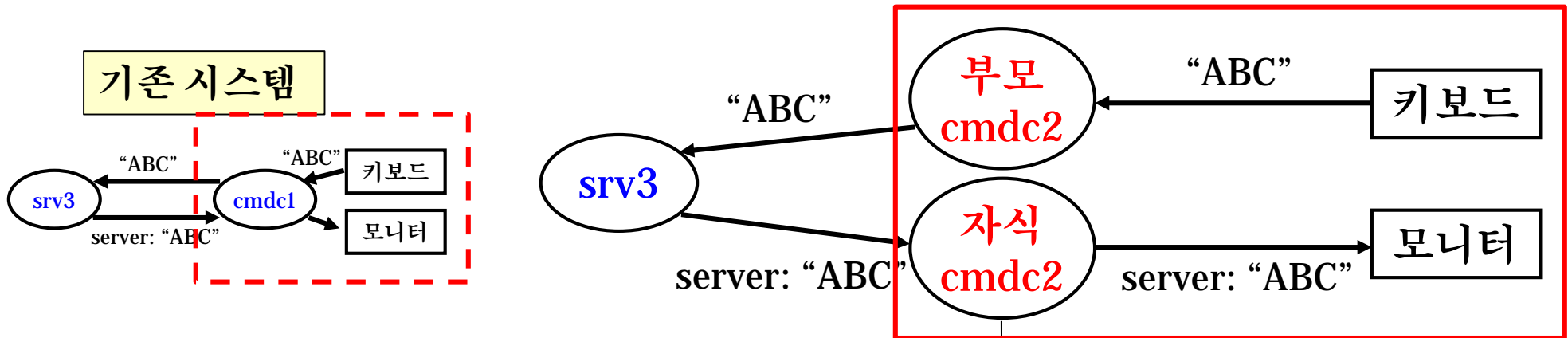
Jaehong Shim

Dept. of Computer Engineering



실습 12-1

cmdc2와 srv3



Cmdc가 두 개의 프로그램으로 분리됨

차이점: 기존의 cmdc는 혼자서 키보드 입력 또는 서버에서 들어오는 메시지를 동시에 처리해야 했음; 반응속도 떨어짐

기존 cmdc를 두 개의 cmdc로 분리하여 서로 입력 단자를 하나씩 전담함; 각 cmdc는 해당 입력 단자만 처리하면 되는 이점이 있음

cmdc2.c 파일 만들기

- ~/up/IPC 디렉토리에서 기존 파일을 복사한다.

```
$ cd ~/up/IPC
```

```
$ cp cmdc1.c cmdc2.c
```

```
$ ls
```

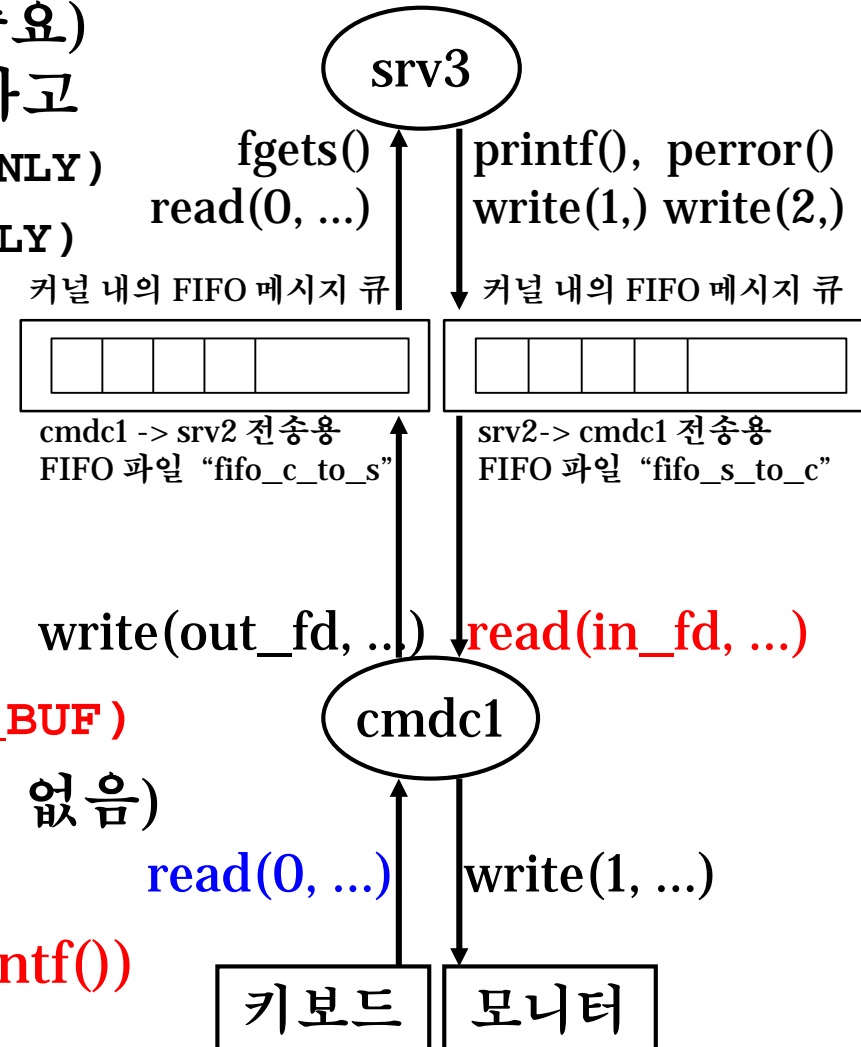
```
cmdc1.c  cmdc2.c  srv1.c  srv2.c  srv3.c  Makefile
```

- Makefile의 TARGETS에 cmdc2를 추가

```
TARGETS = cmdc1 cmdc2 srv1 srv2 srv3
```

cmdc1 클라이언트 프로그램 분석

- 두 개의 FIFO 파일을 open하여 (순서 중요)
서버와 통신할 FIFO 메시지 큐에 접속하고
 - `out_fd = open(fifo_c_to_s, O_WRONLY)`
 - `in_fd = open(fifo_s_to_c, O_RDONLY)`
- 키보드에서 명령어를 입력 받기 (대기)
 - `read(0, cmd_line, SZ_STR_BUF)`
- 이를 서버에 전송 (전송은 대기 없음)
 - `write(out_fd, cmd_line, len)`
- 그 후 서버에서 결과를 수신 (대기)
 - `read(in_fd, cmd_line, SZ_STR_BUF)`
- 수신한 결과를 화면에 출력(출력은 대기 없음)
 - `write(1, cmd_line, len)`
- 서버가 두 번 이상 메시지를 보내면 (`printf()`)
`cmdc`는 몇 번에 걸쳐 받아야 할까?



srv3.c: main() 수정

- 클라이언트로부터 받은 메시지를 에코 해 주고
- 일정시간 대기 (일부러 두 메시지의 전송 시간을 다르게 하기 위해)
- 추가로 메시지를 하나 더 전송함
 - 클라이언트에서 이 두 메시지를 순서적으로 잘 받을 수 있을까?
- 아래 문장 추가할 것

```
// srv3.c의 main() 함수
while (1) {
    ...
    /* 클라이언트로 메시지 송신하는 문장 */
    printf("server: %s", cmd_line);
    sleep(1); // 1초간 대기: 두 메시지의 전송시간을 다르게 하기 위해
    printf("sleep: %s", cmd_line);
    // 위 두 printf()에 의해 전송되는 메시지는 서로 전송시간이 다름
}
```

프로그램 실행 결과

□ 새로 make 한 후 **srv3**와 **cmdc1** 실행 (**cmdc2** 아님)

server: one
sleep: one
두 개가 연속 출력 되어야 함

사용자의 입력은 "tree"인데 앞 전의 "two"를 받음

```
jhshim1@esl: ~/up/IPC
jhshim1:[~/up/IPC] $ cmdc1
one
server: one
two
sleep: one
tree
server: two
sleep: two
four
server: tree
sleep: tree
```

사용자 입력

서버 응답

비정상 출력

```
jhshim1@esl: ~/up/IPC
jhshim1:[~/up/IPC] $ cmdc3
one
server: one
sleep: one
two
server: two
sleep: two
three
server: three
sleep: three
```

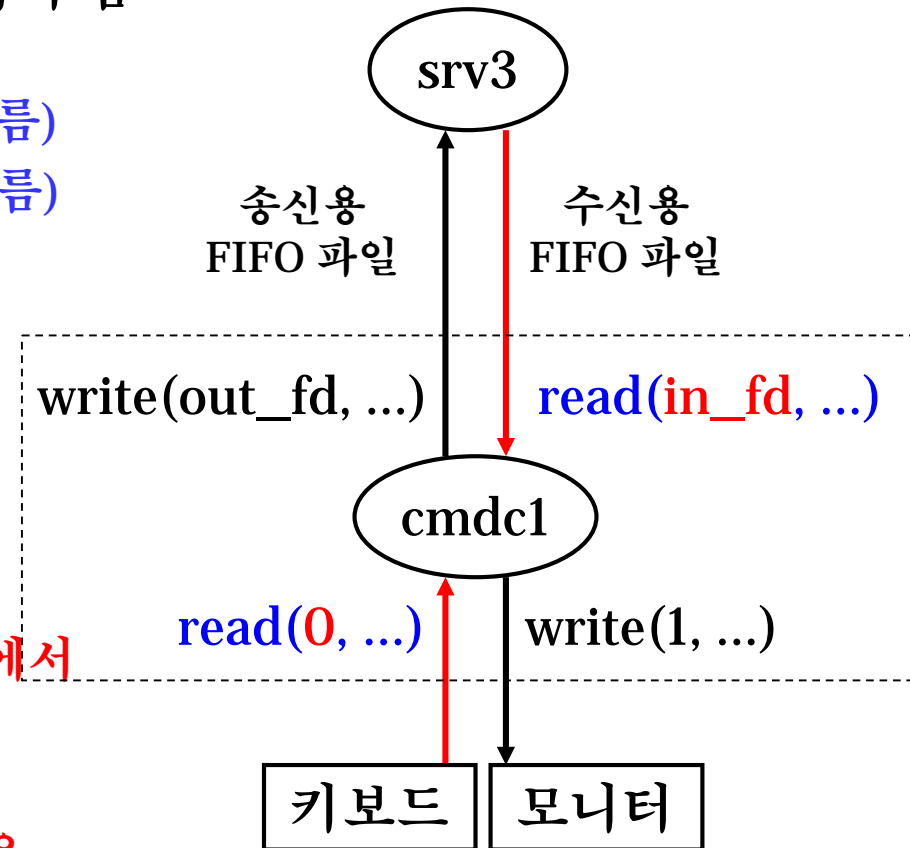
정상 출력

□ 이유

- 서버는 두 번에 걸쳐 printf()로 전송했으나 cmdc는 한 번만 서버로부터 받고 바로 다음 사용자 입력을 받음 (두 번째 메시지 못 받음, cmdc의 버퍼에 저장되어 있음)
- 못 받은 두 번째 메시지는 cmdc가 다음 번에 사용자 메시지를 보내고 난 후 받음

cmdc1 클라이언트 프로그램의 문제점

- ❑ cmdc1는 두 곳에는 들어오는 입력을 받아야 함
(어디에서 먼저 데이터가 들어 올까?)
 - 키보드 입력 데이터 (언제 입력 될지 모름)
 - Srv3가 전송한 데이터 (언제 수신 될지 모름)
- ❑ **read(0, ...) 함수의 한계**
 - 키보드에 도착한 데이터가 있으면 데이터를 가지고 바로 리턴
 - 도착한 데이터가 없으면 데이터가 도착할 때까지 **블록킹 상태에서 계속 대기**
 - 1. 문제는 블록킹 상태에서 대기하는 중에 srv3에서 보낸 데이터가 도착하면, in_fd에서 이를 즉시 받아 처리할 수가 없음
 - 반대 상황도 마찬가지임
 - 2. 만약 서버가 여러 번 메시지를 보낼 경우 cmdc는 몇 번 읽어야 할까? (문제는 언제, 몇 개를 보낼지 모른다는 것)



cmdc1 문제의 해결 방안

□ 프로세스 기반 해결방안

- fork() 함수를 이용 자식 프로세스를 생성
- 부모 프로세스: 키보드에서 읽어 -> 서버(cmd)로 전송
- 자식 프로세스: 서버에서 보낸 데이터 읽어 -> 화면에 출력

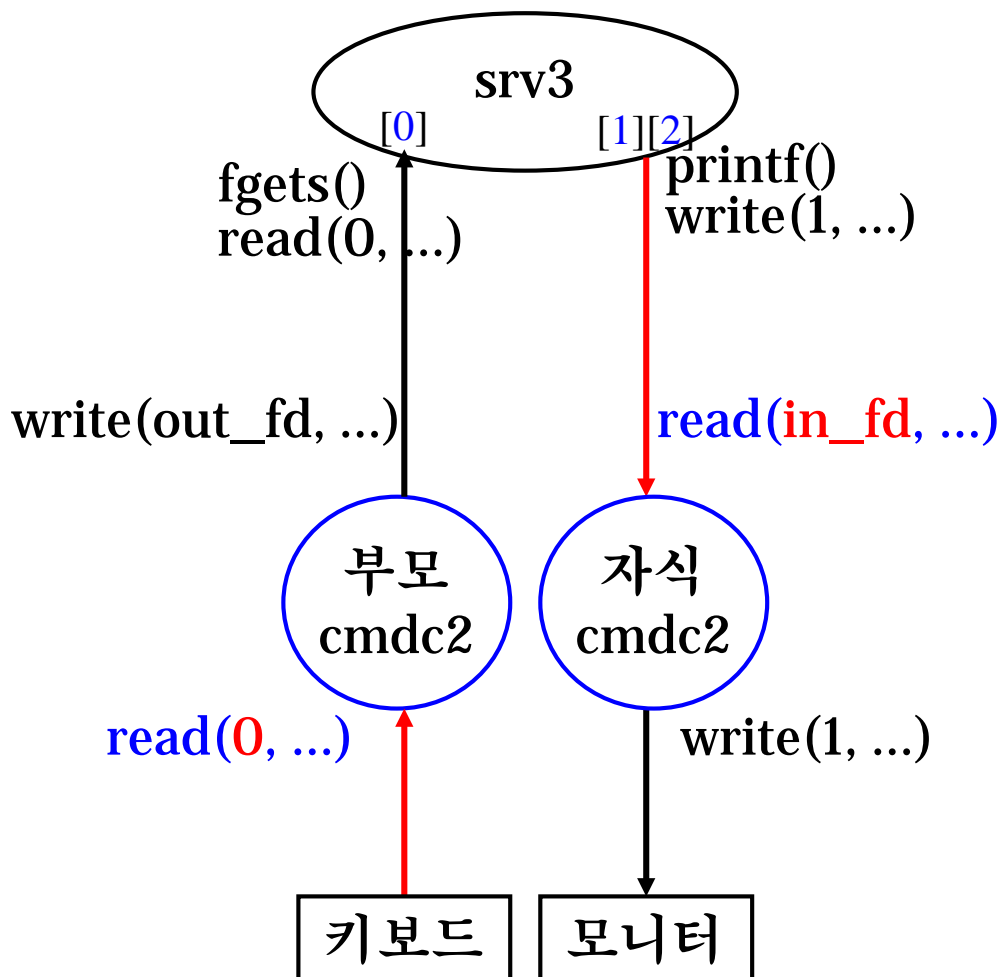
□ 쓰레드 기반 해결방안

- main 쓰레드에서 두 개의 쓰레드를 새로 생성
- SendToServerThread: 키보드에서 읽어 -> 서버(cmd)로 전송
- RecvFromServerThread: 서버에서 보낸 데이터 읽어 -> 화면에 출력

□ Select() 기반 해결방안

- 입력을 받을 두 개의 파일 기술자 (0, in_fd)를 미리 등록하고
- select() 함수를 호출하여 대기함
- select()는 둘 중 하나에서 데이터가 도착하면 리턴함
- 어느 쪽에서 데이터가 들어 왔는지 확인한 후 read()를 이용하여 읽음

프로세스 기반 해결방안 (프로세스의 자기복제)



- ❑ 부모 cmdc2가 자기 복제(fork())하여 자식 cmdc2를 생성
- ❑ 부모
 - 키보드에서 대기하고 있다가 사용자가 데이터를 입력하면 이를 읽어 서버로 전송
- ❑ 자식
 - in_fd에서 서버가 전송한 데이터를 대기하고 있다가 데이터가 도착하면 이를 수신하여 화면에 출력함
 - 서버가 계속 데이터를 보내면 이를 연속으로 받아 화면에 정상 출력하는 것이 가능함
- ❑ 부모와 자식 프로세스에서 사용하지 않는 파일 핸들은 close() 함

문제 해결: cmdc2.c 수정

// 파일 앞쪽에 헤드 파일 삽입: \$ man wait 해서 찾아 볼 것
// single_process() 함수 전체 삭제하고 그 위치에 아래 함수 배치

```
void dual_process(void)
```

```
{
```

```
    pid_t pid;
```

```
    // 두개의 프로세스로 분리하여 각 프로세스가 입출력을 따로 담당함
```

```
    if ((pid = fork()) < 0)
```

```
        perror("fork");
```

```
        // 자기 복제 실패한 경우
```

```
    else if (pid > 0) {
```

```
        // 부모 프로세스: 키보드 입력 받아 서버에 전송
```

```
        close(in_fd);
```

```
        // 사용하지 않기 때문에 닫음
```

```
        input_send_loop();
```

```
        // 반복하여 키보드에서 입력 받아 서버로 전송
```

```
        wait(NULL);
```

```
        // 부모가 먼저 끝난 경우, 자식이 종료할 때까지 기다림
```

```
    }
```

```
    else {
```

```
        // 자식 프로세스: 서버가 보내 준 메시지 받아 화면에 출력
```

```
        close(out_fd);
```

```
        // 사용하지 않기 때문에 닫음
```

```
        recv_output_loop();
```

```
        // 반복하여 서버에서 받은 데이터를 모니터(화면)로 출력
```

```
        printf("child: exit\n");
```

```
}
```

```
}
```

```
int main()
```

```
{
```

```
    connect_to_server();
```

```
    // 두개의 프로세스로 분리하여
```

```
    // 각각 입출력을 따로 담당함
```

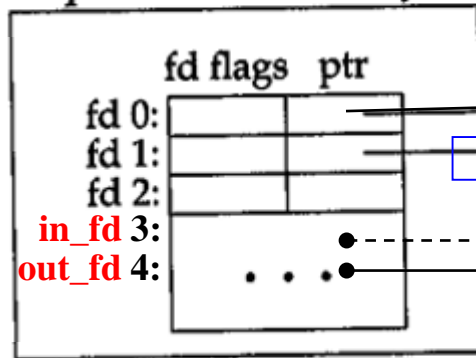
```
    dual_process(); // 수정할 것
```

```
    dis_connect();
```

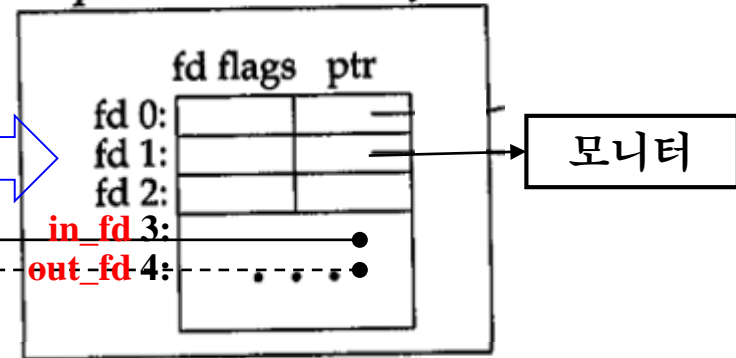
```
}
```

fork() : 부모의 파일 핸들 테이블을 자식에게 복사

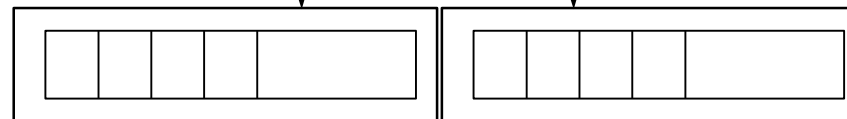
부모 cmdc2 프로세스 구조체
process table entry



자식 cmdc2 프로세스 구조체
process table entry



cmdc1 -> srv2 전송용
FIFO 파일 "fifo_c_to_s"

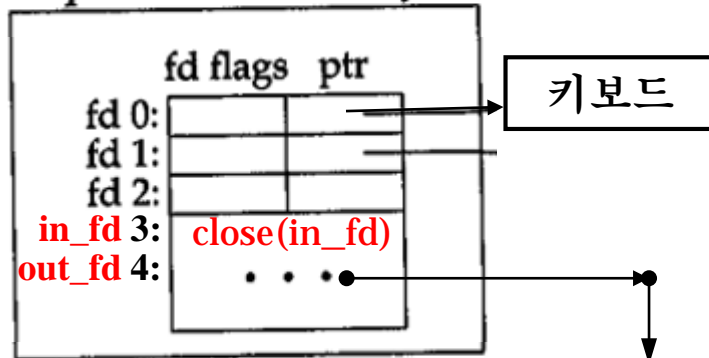


srv2 -> cmdc1 전송용
FIFO 파일 "fifo_s_to_c"

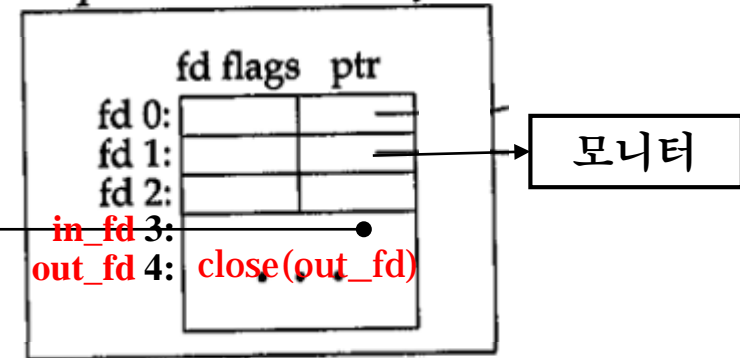
srv3

불필요한 파일 핸들 close()

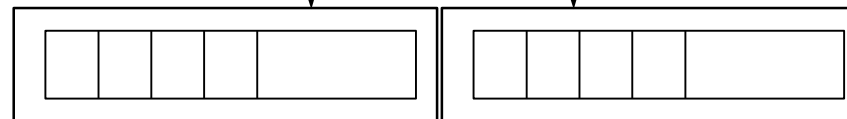
부모 cmdc2 프로세스 구조체
process table entry



자식 cmdc2 프로세스 구조체
process table entry



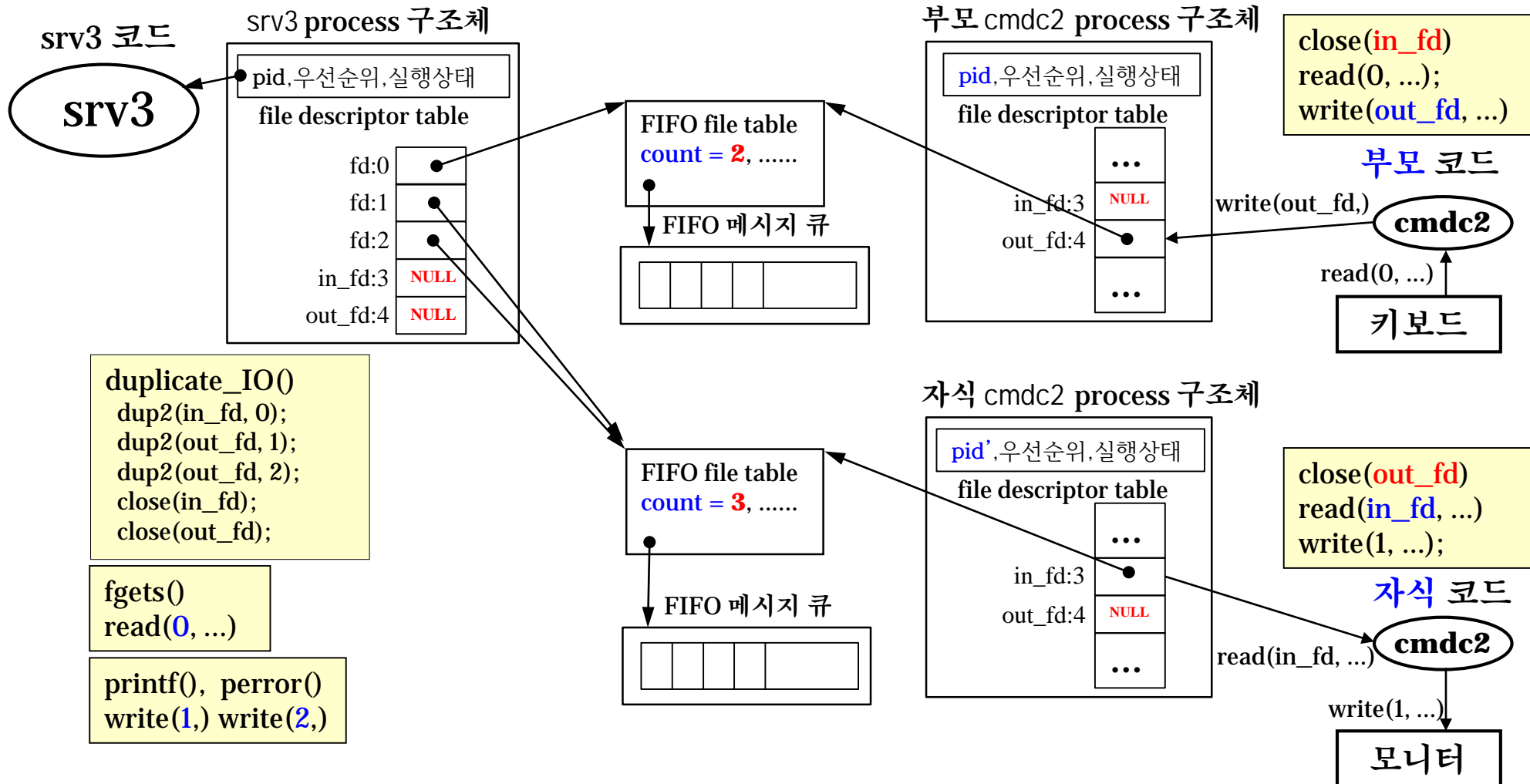
cmdc1 -> srv2 전송용
FIFO 파일 "fifo_c_to_s"



srv2 -> cmdc1 전송용
FIFO 파일 "fifo_s_to_c"

srv3

fork() 이후의 커널(운영체제) 내부의 모습



문제 해결: cmdc2.c 수정

// dual_process() 함수 바로 앞에 배치

// 부모 cmdc2: 반복하여 키보드에서 입력 받아 서버로 전송

void input_send_loop(void)

```
{
    while (1) {
        if (input_send() <= 0) // 키보드 입력 후
                               // 서버로 전송
            break;
    }
}
```

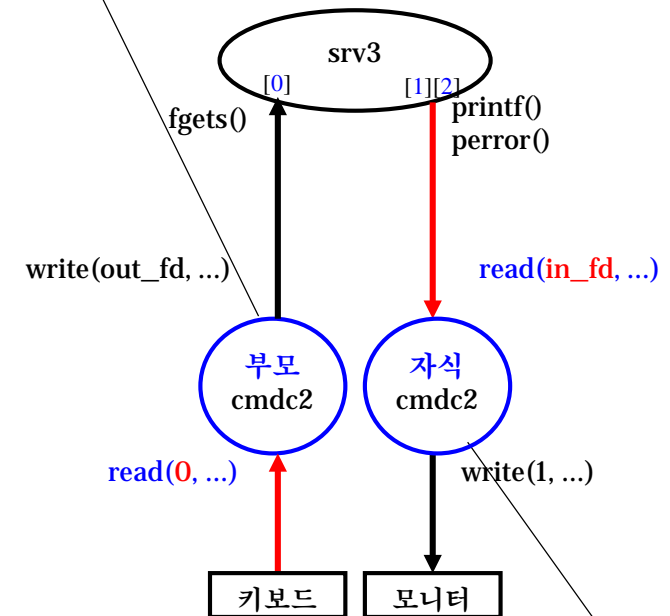
// 자식 cmdc2: 반복하여 서버에서 받은 데이터를
// 모니터(화면)로 출력

void recv_output_loop(void)

```
{
    while (1) { // 서버로부터 받고 화면에 출력
        if (recv_output() <= 0)
            break;
    }
}
```

기존 input_send() 함수

```
len = read(0, cmd_line, SZ_STR_BUF);
if (len <= 0) return len;
if (write(out_fd, cmd_line, len) != len)
    return -1; // 서버 종료한 경우
return len;
```

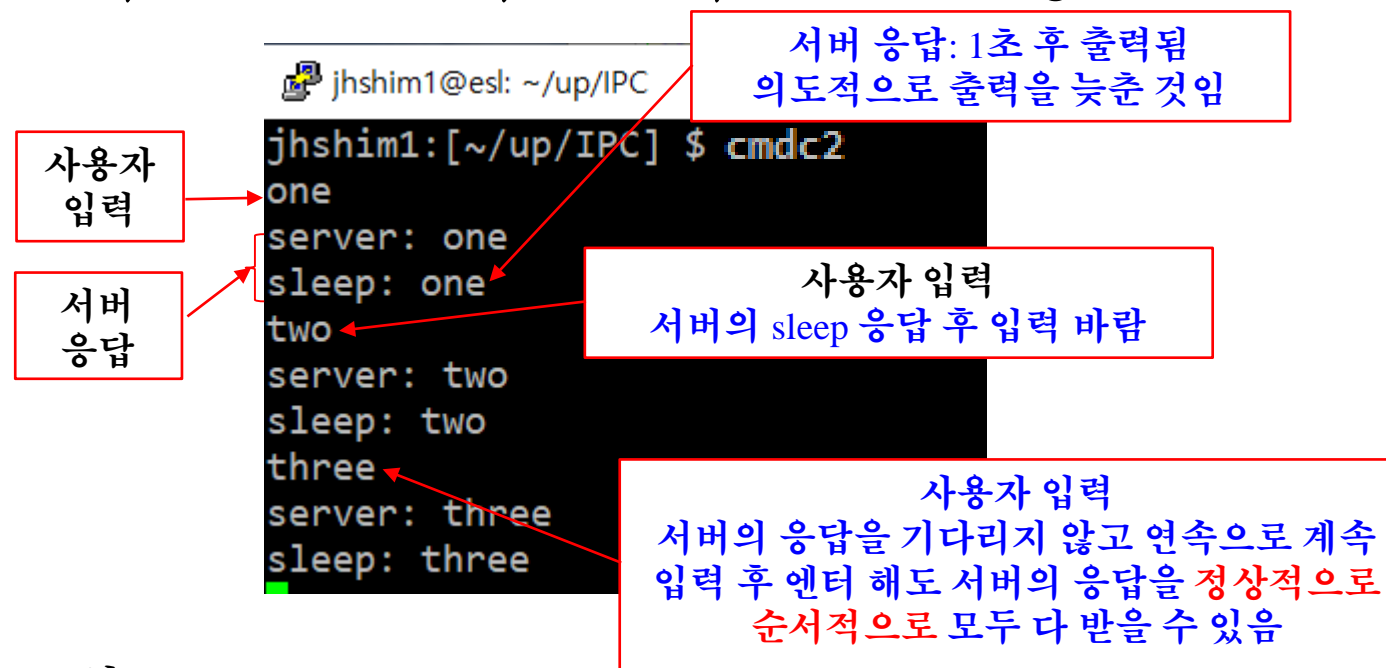


기존 recv_output() 함수

```
len = read(in_fd, cmd_line, SZ_STR_BUF);
if (len <= 0) return len; // 서버 종료한 경우
if (write(1, cmd_line, len) != len)
    return -1;
return len;
```

프로그램 실행 결과

□ 새로 make 한 후 srv3와 cmdc2 실행

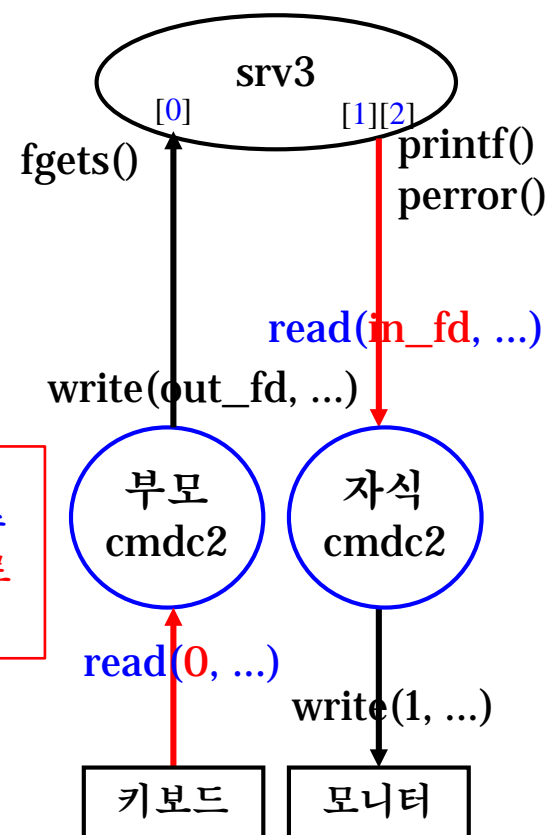


□ 부모 cmdc2

- 사용자가 입력한 것을 서버로 전송함

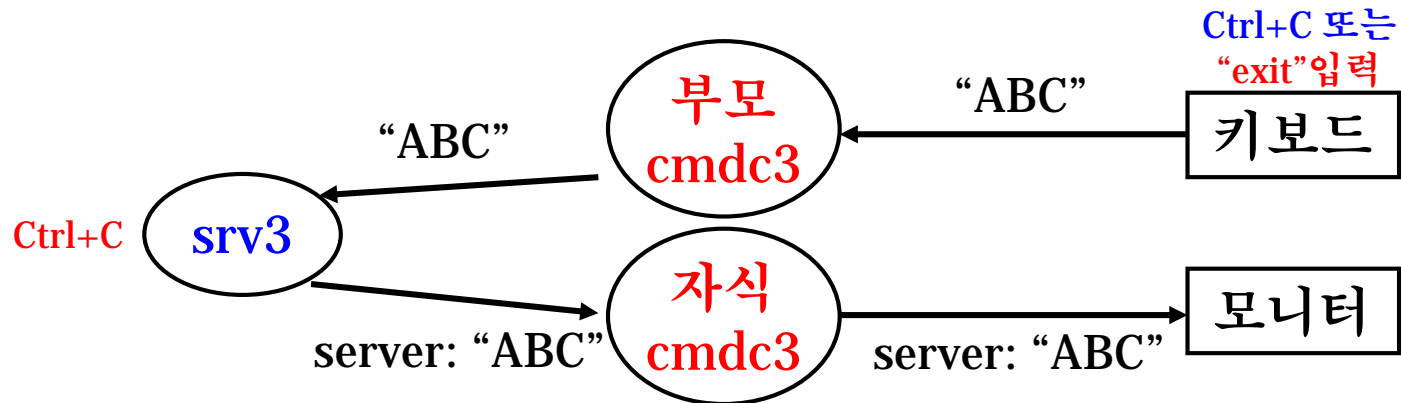
□ 자식 cmdc2

- 서버의 에코 메시지 "servr: one"를 바로 출력
- 1초 후에 서버의 "sleep: one" 메시지를 받아 출력



실습 12-2

cmdc3와 srv3



문제점: srv3에서 Ctrl+C를 입력하거나 또는 cmdc2에서 "exit[enter]"를 입력하면 부모 cmdc2가 정상 종료하지 않음

해결방안: 부모 cmdc가 자식 종료 신호를 받아 정상 종료하게 함

서버와 클라이언트의 종료 문제점

□ 두 프로세스를 한번에 종료시키는 세가지 방법

1. cmdc2에서 Ctrl+C를 누름

- cmdc2이 종료되고 이어 srv3이 종료됨 (모두가 정상 종료됨)

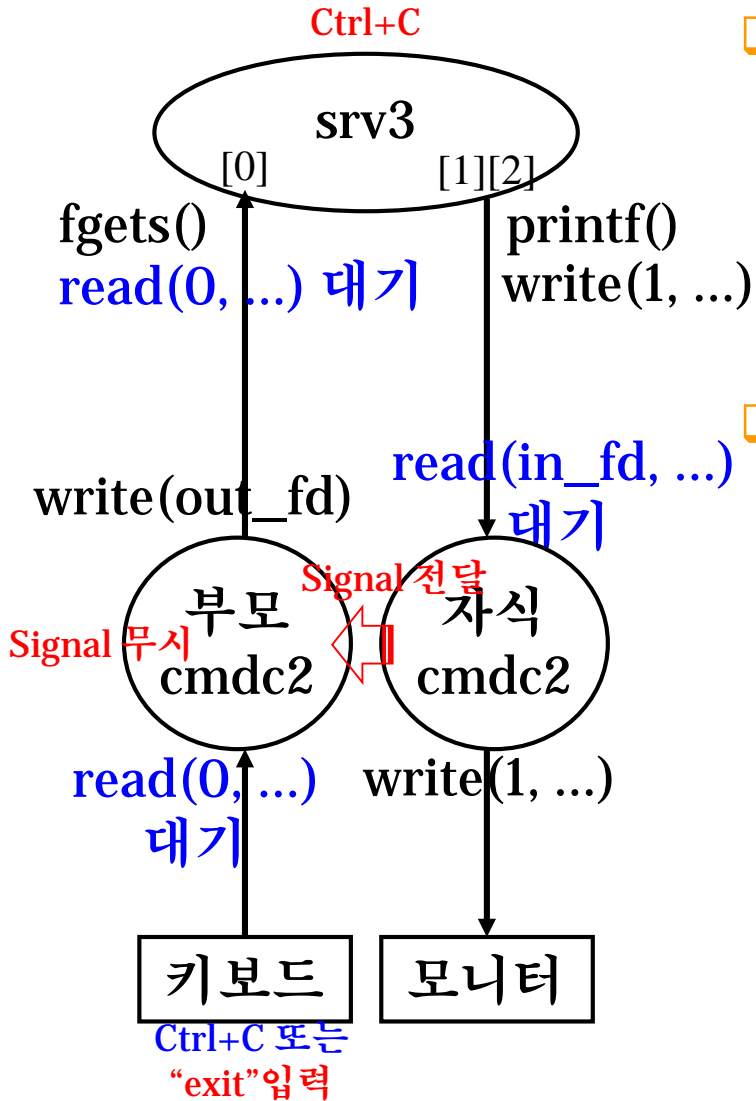
2. cmdc2에서 “exit[enter]”입력 또는 srv3에서 Ctrl+C 누름

- 서버와 클라이언트 모두 종료함
- 그러나 “exit[enter]”입력 후 또는 srv3에서 Ctrl+C 누른 후 부모 cmdc2는 바로 종료하지 않음
- 다시 한번 엔터를 입력해야 그제서야 종료함

```
// srv3.c
int main(int argc, char *argv[]) {
    while (1) {
        /* 클라이언트로부터 메시지 수신 */
        if (strncmp(cmd_line, "exit", 4) == 0)
            break; // 서버는 여기서 종료
    }
}
```

cmdc2의 종료할 때의 문제점

```
// srv3.c
int main(int argc, char *argv[]) {
    while (1) {
        if (fgets(..., stdin) == NULL)
            break;
        if (strncmp(cmd_line, "exit", 4) == 0)
            break; // 서버는 여기서 종료
    }
}
```



□ 종료절차 1

- cmdc2에서 입력시 **Ctrl+C** 입력
- 부모 cmdc2 종료 -> cmdc2의 out_fd 자동 close -> 이와 연결된 서버의 표준 입력 fgetc()가 NULL 리턴 -> srv3 종료 -> srv3측 송수신 FIFO close됨 -> 자식 cmdc2의 read() 함수 리턴 0 -> 자식 cmdc2 종료

□ 종료절차 2

- cmdc2에서 “exit” 입력 또는 srv3에서 Ctrl+C
- srv3 종료 -> srv3측 FIFO 용 0, 1, 2 파일 핸들 자동 close됨 -> 이와 연결된 자식 cmdc2의 read() 함수 리턴 0 -> 자식 cmdc2 종료 -> 부모에게 SIGCHLD (자식 종료)신호 전달 -> 부모는 이 신호를 무시
- 문제점: 부모 cmdc2는 즉시 종료하지 않고
다음 키 입력 받아 전송 시 종료
- 해결방안: 자식 cmdc2가 종료할 때 부모 cmdc2에게 전달된 종료 신호(SIGCHLD signal)를 부모가 무시하지 않고 catch하여 스스로 정상 종료하게 함

□ Signal

- Software interrupts
- 실행 중인 프로세스에게 전달되는 하나의 숫자 (시그널 도착)
- 운영체제는 이 숫자와 맵핑된 함수를 실행시킴 (시그널 처리)
 - 이 함수는 디폴트 함수를 사용해도 되고 필요하면 사용자가 자신의 함수를 만든 후 `signal()` 함수를 호출하여 자신의 함수를 미리 등록할 수도 있음
 - 또는 도착한 신호를 무시할 수도 있음

□ 키보드에서

- **Ctrl+C**를 누를 경우 시그널 **SIGINT**(#defin으로 정의된 상수 값임)가 현재 실행되는 프로세스에게 전달되고, 이 시그널이 도착하면 그 프로세스는 죽음(디폴트 시그널 처리 함수 실행 결과)
- **Ctrl+Z**를 누를 경우 **SIGTSTP** 시그널이 현재 실행되는 프로세스에게 전달되고, 이 시그널이 도착하면 그 프로세스는 실행 중지됨
- **Ctrl+**를 누를 경우 **SIGQUIT** 시그널이 도착하는데 그러면 그 프로세스는 부수적인 결과물(core 파일)을 생산하고 죽음

signal.c 파일 생성

```
$ cd ~/up/IPC
```

□ ~/up/IPC에 있는 Makefile의 TARGETS에 **signal**을 추가

```
TARGETS = cmdc1 cmdc2 srv1 srv2 signal
```

□ ~/up/IPC에 signal.c 소스 파일을 만든 후 아래 코드를 입력하라.

```
#include <stdio.h>                                // printf(), gets()

int main(int argc, char *argv[])
{
    while (1) {
        char buf[128];
        fgets(buf, 128, stdin);
        printf("%s", buf);
    }
}
```

signal 프로그램 실행하기

- make 한 후 signal을 실행함
 - 옆의 실행 결과처럼 Ctrl+C, Ctrl+\\, Ctrl+Z를 눌러본다.
 - Ctrl+Z의 경우 프로그램의 실행을 일시 중지시키는 것이므로 %1을 통해 signal 프로그램의 실행을 복구시켜 주어야 한다.

```
jhshim1@esl: ~/up/IPC
jhshim1:[~/up/IPC] $
jhshim1:[~/up/IPC] $ signal
This is test.
This is test.
^C
jhshim1:[~/up/IPC] $ signal
^\\끝 내 기 (core dumped)
jhshim1:[~/up/IPC] $ signal
^Z
[1]+  정 지 됨                  signal
jhshim1:[~/up/IPC] $ jobs
[1]+  정 지 됨                  signal
jhshim1:[~/up/IPC] $ %1
signal
This is test.
This is test.
^C
jhshim1:[~/up/IPC] $
```

시그널 처리 함수 작성

□ 기존 signal.c 파일에 아래 내용을 main() 함수 앞에 삽입하라.

```
// Ctrl+\ 를 눌렀을 때 호출되는 시그널 함수
static void sig_quit(int sig)
{
    signal(SIGINT, SIG_DFL); // ctrl+c 를 눌렀을 때 원래대로 프로그램 죽게 설정함
    // 즉, SIGINT의 디폴트 시그널 처리 함수를 호출하게 함
    printf("  sig_quit: signal(SIGINT, SIG_DFL) called.\n");
}

// Ctrl+Z 를 눌렀을 때 호출되는 시그널 함수
static void sig_stop(int sig) {
    printf("  sig_stop\n");
}
```

시그널 처리 함수 등록

- ❑ main() 함수 내에 아래 내용을 삽입하라.
- ❑ 필요한 헤드 파일도 include 시켜라.

```
// $ man -s2 signal 해서 필요한 헤드파일을 signal.h 앞에 include 시켜라.
```

```
int main(int argc, char *argv[])
{
    // 각 시그널을 처리하는 함수를 미리 등록함
    signal(SIGINT, SIG_IGN); // Ctrl+C 를 눌렀을 때 시그널을 무시함
    signal(SIGQUIT, sig_quit); // Ctrl+\ 를 눌렀을 때 sig_quit()을 호출함
    signal(SIGTSTP, sig_stop); // Ctrl+Z 를 눌렀을 때 sig_stop()을 호출함
    // 이제 위 시그널이 도착하면 등록된 시그널 처리 함수가 자동 실행됨

    while (1) {
        ... // 기존 코드
    }
}
```

시그널(신호) 잡기

- ❑ make 한 후 signal을 실행함
 - 옆의 실행 결과처럼
Ctrl+C, Ctrl+\, Ctrl+Z를
눌러본다.
 - 프로그램이 죽거나 일시
정지 하지 않음
- ❑ 시그널이 도착하면 등록된
시그널 처리 함수가 실행됨
- ❑ Ctrl+\ 를 누를 경우,
SIGINT(Ctrl+C) 시그널을
디폴트 시그널 처리 함수로
다시 재설정함
 - 따라서 이후 Ctrl+C를 누를
경우 원래대로 프로그램
종료함

```
jhshim1@esl: ~/up/IPC
jhshim1:[~/up/IPC] $
jhshim1:[~/up/IPC] $ signal
^CThis is test.
This is test.
^C^Z  sig_stop
^Z  sig_stop
This is test
This is test
^\  sig_quit: signal(SIGINT, SIG_DFL) called.
This is test
This is test
^C
jhshim1:[~/up/IPC] $
```


cmdc3.c 파일 만들기

- ~/up/IPC 디렉토리에서 기존 파일을 복사한다.

```
$ cd ~/up/IPC
```

```
$ cp cmdc2.c cmdc3.c
```

```
$ ls
```

```
... cmdc2.c cmdc3.c ... srv3.c Makefile
```

- Makefile의 TARGETS에 cmdc3를 추가

```
TARGETS = cmdc1 cmdc2 cmdc3 srv1 srv2 srv3
```

cmdc3.c: 시그널 처리 함수 작성

```
// $ man -s2 signal 해서 필요한 헤드파일을 cmdc3.c 앞쪽에 include 시킨다.  
  
// void dual_process(void) 함수 바로 앞에 아래 함수를 추가한다.  
  
// *****  
// SIGCHLD 시그널 처리 함수  
// 자식 cmdc3가 종료했을 때 부모 cmdc3에게 SIGCHLD 시그널이 도착  
// SIGCHLD 시그널이 부모에게 도착하면 사전에 등록된 함수인 sig_child()가 호출됨  
// *****  
  
static void sig_child(int sig)  
{  
    printf("parent: sig_child: exit\n");  
    exit(0); // 자식 cmdc3가 종료하면 시그널을 받아 부모 cmdc3가 여기서 종료  
}
```

cmdc3.c: 시그널 처리 함수 등록

기존 input_send() 함수

```
len = read(0, cmd_line, SZ_STR_BUF);
if (len <= 0) return len;
if (write(out_fd, cmd_line, len) != len)
    return -1;
return len;
```

```
void dual_process(void)
```

```
{
```

```
    if ((pid = fork()) < 0)
```

```
        ...
```

```
    else if (pid > 0) {        // 부모 프로세스: 키보드 입력 받아 서버에 전송
```

```
        // SIGCHLD 시그널을 처리하는 함수(sig_child)를 미리 등록함
```

```
        // 자식 프로세스(cmdc3)가 종료했을 때 부모 프로세스에게 SIGCHLD 시그널이 전달되고
```

```
        // 이 시그널이 부모에게 도착하면 사전에 등록된 함수인 sig_child()가 호출됨
```

```
        // 대부분 자식이 먼저 죽게 된다. 따라서 부모에게 SIGCHLD 신호가 도착하며,
```

```
        // 부모는 대부분 sig_child()에서 종료하게 됨
```

```
        signal(SIGCHLD, sig_child); // 자식이 먼저 죽으면, 호출할 함수 등록
```

```
        close(in_fd);           // 사용하지 않기 때문에 닫음
```

```
        input_send_loop();      // 반복하여 키보드에서 입력 받아 서버로 전송
```

```
        wait(NULL);            // 부모가 먼저 끝난 경우, 자식이 종료할 때까지 기다림
```

```
    } else { // 자식 프로세스
```

```
        ...
```

```
        printf("child: exit\n");
```

```
    }
```

```
static void sig_child(int sig)
```

```
{// 자식이 종료하면 시그널을 받아 부모 cmdc3 여기서 종료
```

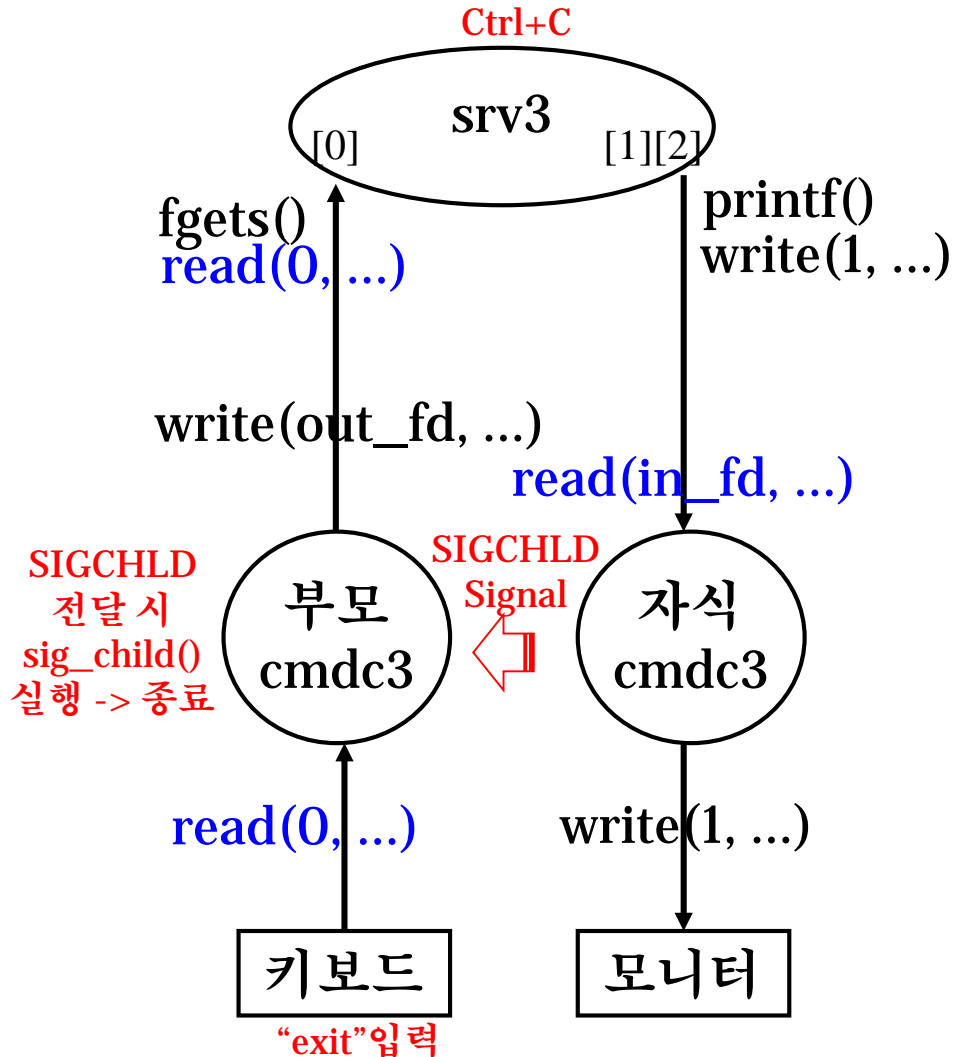
```
    printf("parent: sig_child: exit\n");
```

```
    exit(0);
```

```
}
```

프로그램 실행 및 종료 확인

```
// srv.C: int main(int argc, char *argv[]) {  
    while (1) { /* 클라이언트로부터 메시지 수신 후 */  
        if (strcmp(cmd_line, "exit", 4) == 0)  
            break; // 서버는 여기서 종료  
    }  
}
```

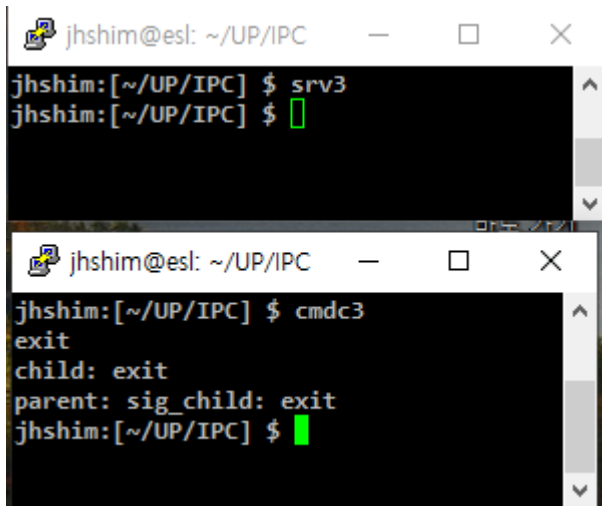


- ❑ make 실행한 후 srv3와 cmdc3 실행
- ❑ 종료 절차 1
 - cmdc3에서 입력시 **Ctrl+C** 입력
 - 정상 종료되는지 확인 (원래 정상이었음)
- ❑ 종료 절차 2
 - cmdc3에서 **"exit"** 입력
 - srv3 종료 -> srv3측 FIFO 용 0, 1, 2 파일 핸들 자동 close됨 -> 이와 연결된 자식 cmdc3의 read() 함수 리턴 0 -> 자식 cmdc3 종료 -> 부모 cmdc3에 SIGCHLD 전달 -> 부모의 sig_child() 함수 실행 -> 부모 종료 => 세 프로세스 모두 종료하는지 확인할 것
- ❑ 종료 절차 3
 - 또는 srv3에서 **Ctrl+C**
 - srv3 종료 -> 위 종료절차 2 과정 수행

서버와 클라이언트의 정상 종료

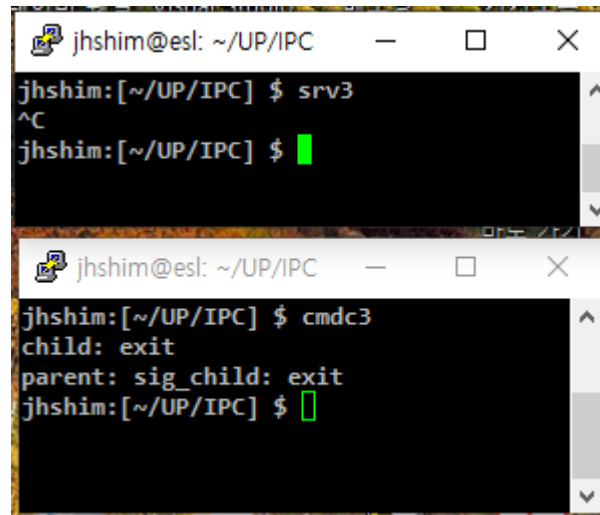
□ make 한 후 srv3, cmdc3 실행

1. cmdc3에서
“exit[enter]” 입력
○ 모두 종료



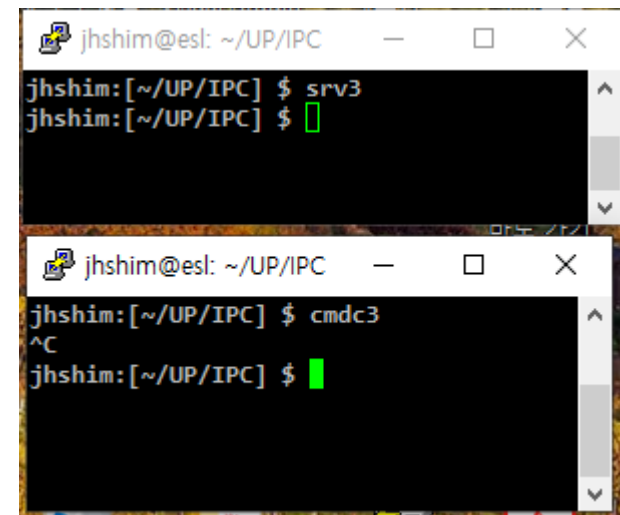
```
jhshim@esl: ~/UP/IPC
jhshim:[~/UP/IPC] $ srv3
jhshim:[~/UP/IPC] $
jhshim:[~/UP/IPC] $ cmdc3
exit
child: exit
parent: sig_child: exit
jhshim:[~/UP/IPC] $
```

2. srv3에서 Ctrl+C 입력
○ 모두 종료



```
jhshim@esl: ~/UP/IPC
jhshim:[~/UP/IPC] $ srv3
^C
jhshim:[~/UP/IPC] $
jhshim:[~/UP/IPC] $ cmdc3
child: exit
parent: sig_child: exit
jhshim:[~/UP/IPC] $
```

3. cmdc3에서 Ctrl+C 입력
○ 모두 종료



```
jhshim@esl: ~/UP/IPC
jhshim:[~/UP/IPC] $ srv3
jhshim:[~/UP/IPC] $
jhshim:[~/UP/IPC] $ cmdc3
^C
jhshim:[~/UP/IPC] $
```