41865 Unix Systems

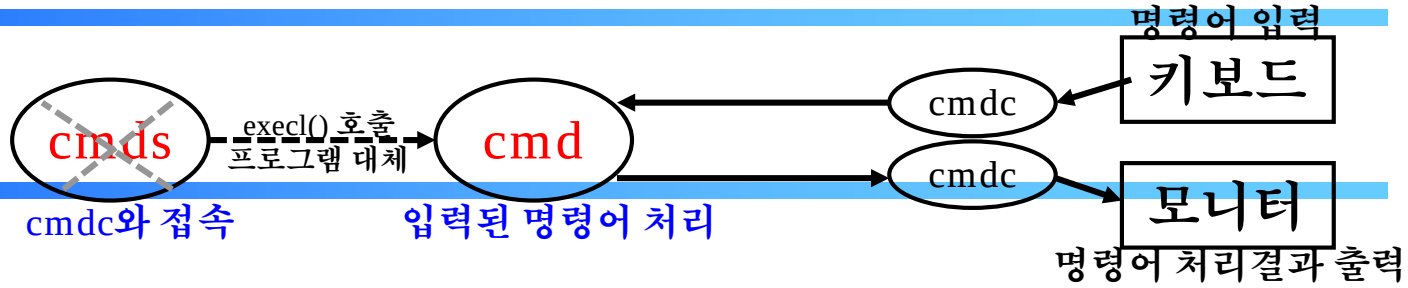
실습 13
IPC: Interprocess Communication
cmds and cmdc

Jaehong Shim

Dept. of Computer Engineering

CHOSUN UNIVERSITY 1946
조선대학교
CHOSUN UNIVERSITY

목적



- 기존의 cmd을 서버와 클라이언트, 총 3개의 프로그램으로 구성한다.
- cmds: cmdc와 접속한 후 기존의 cmd로 교체하는 서버 프로그램
  - cmdc와 통신 접속 후 표준입출력을 메시지 큐(fifo)로 변경
  - execl() 호출하여 cmds 프로세스를 cmd로 프로그램 교체
- cmd: 기존에 구현했던 명령어 해석기 cmd
  - 클라이언트가 보낸 명령어를 수신하고 (fgets() 함수 이용)
  - 해당 명령어를 처리한 후 (기존의 cmd가 처리하던 내용),
  - 결과를 다시 클라이언트에게 전송한다. (printf() 함수 이용)
- cmdc: 클라이언트 프로그램
  - 키보드 입력(명령어)을 읽어 서버로 전송하고,
  - 서버로부터 처리결과를 수신하여 화면에 출력한다.
  - 기존의 cmdc3 프로그램이 하던 역할

새로 시작하기

- ~/up/IPC 디렉토리에서 기존 파일을 복사한다.

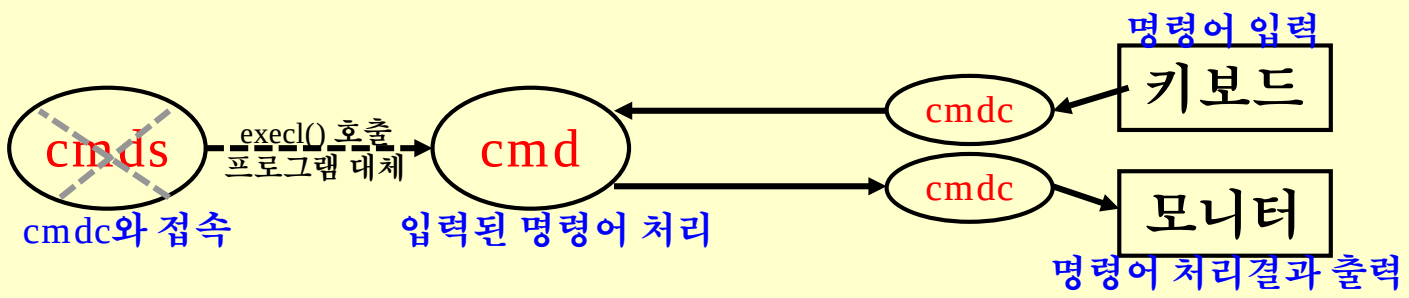
```
$ cd ~/up/IPC
$ cp srv3.c  cmds.c
$ cp cmdc3.c  cmdc.c
$ ls
  cmdc.c  cmds.c  Makefile   기타 기존 파일들
```

- Makefile의 TARGETS에 cmdc와 cmds를 추가

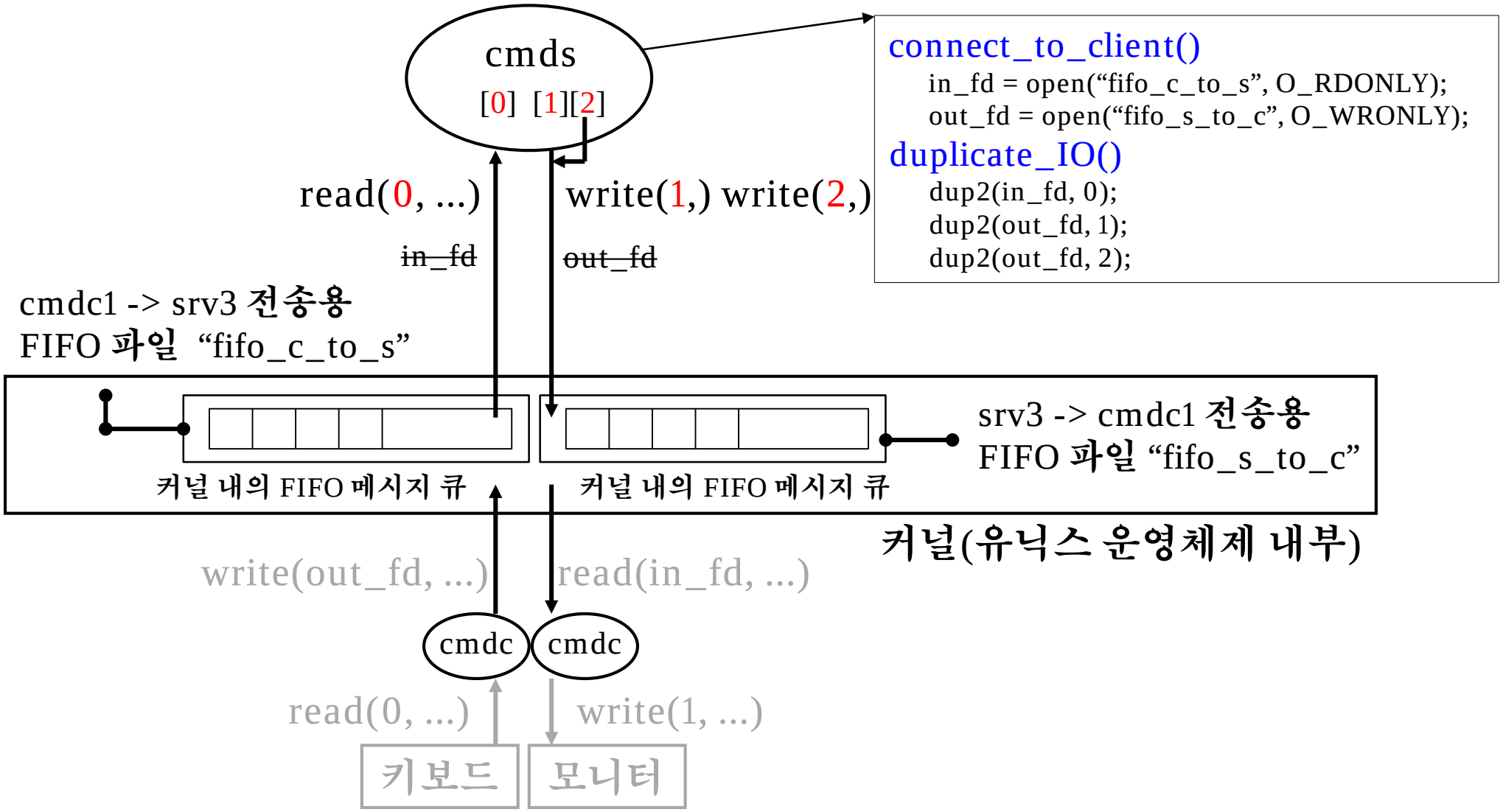
```
TARGETS = cmdc1 cmdc2 cmdc3 srv1 srv2 srv3 cmdc cmds
```

cmds.c: main() 함수를 다음과 같이 수정

```
int main(int argc, char *argv[])
{
   connect_to_client();    // 클라이언트 프로그램 cmdc와 연결한다.
   duplicate_IO();         // 표준 입출력 파일 핸들을 변경한다.

  // cmds 프로그램을 기존의 cmd 프로그램으로 교체
  // 프로그램은 교체 되어도 cmdc와 접속한 0, 1, 2 표준입출력은 그대로
  // 존속하며 cmdc와의 접속(연결) 상태도 그대로 유지하고 있음

   execl("../cmd/cmd", "cmd", NULL);

  while (1) { // while 문 이하 전체를 삭제할 것
      . . .
  }
  dis_connect();
}
```

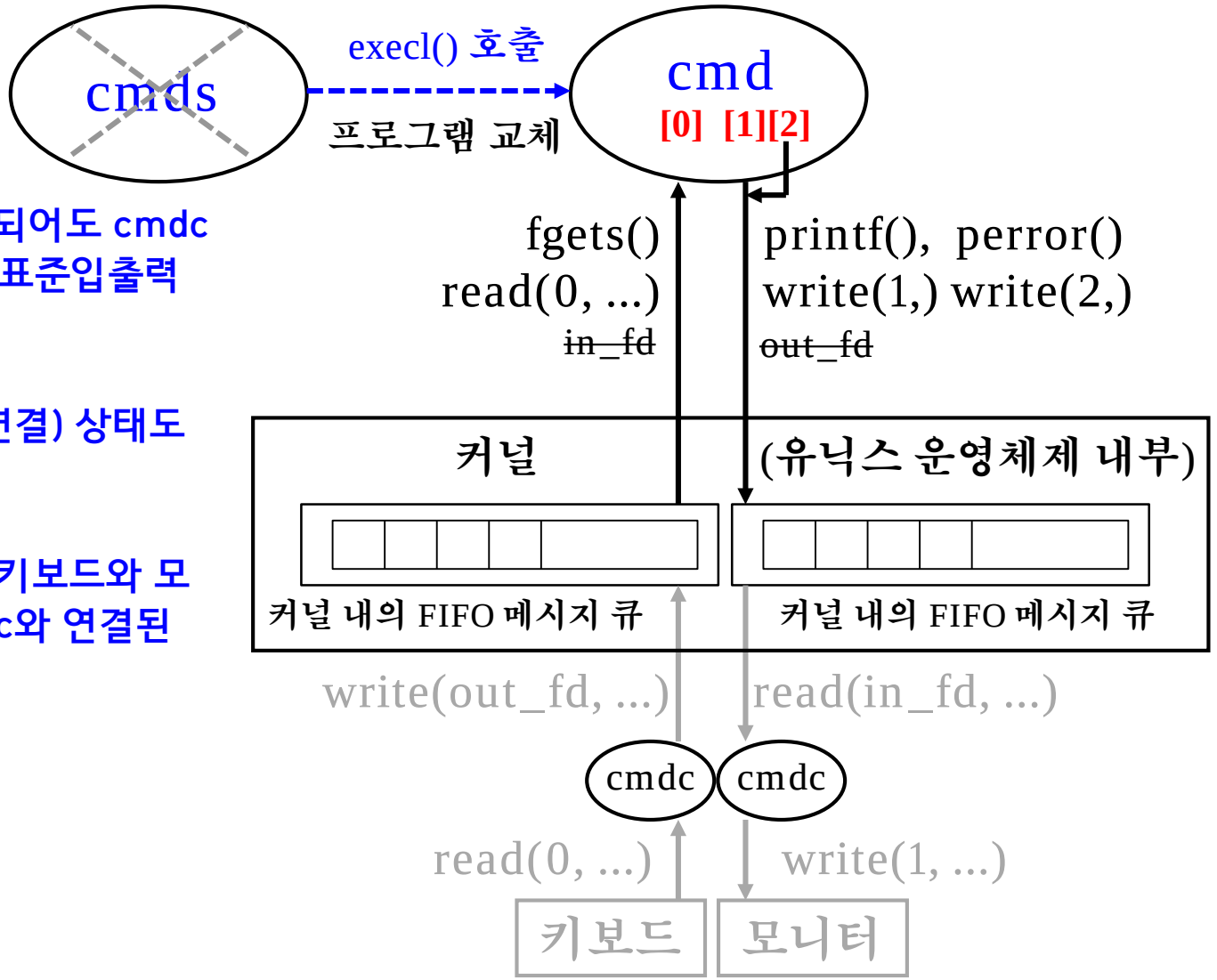


cmds의 역할: cmdc와 접속 및 표준입출력을 메시지 큐로 교체

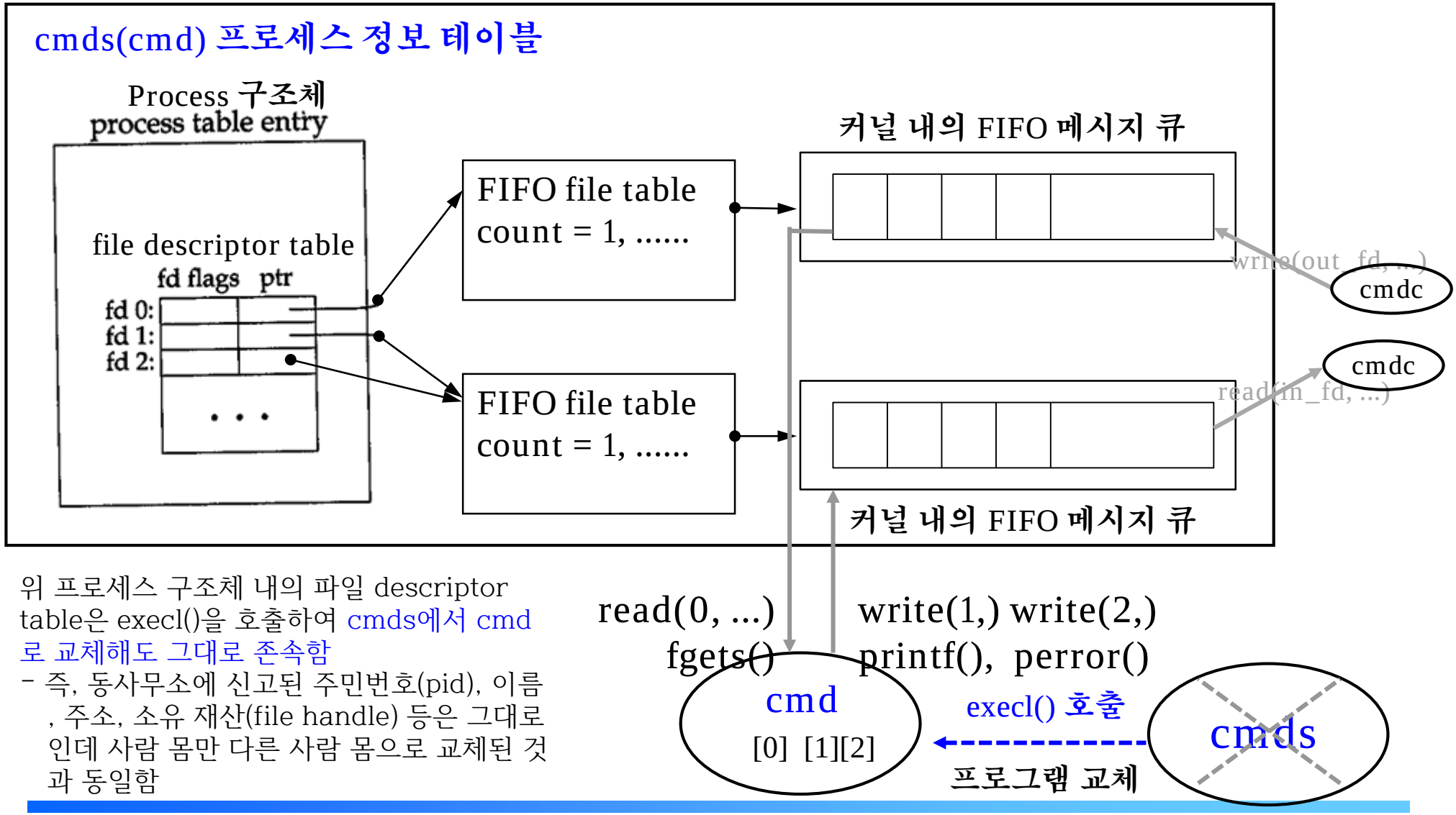


cmds 프로그램에서 cmd 프로그램으로 교체한 후의 상황

- 프로그램은 교체 되어도 cmdc와 접속한 0, 1, 2 표준입출력은 그대로 존속
- cmdc와의 접속(연결) 상태도 그대로 유지
- cmd의 입출력은 키보드와 모니터가 아닌 cmdc와 연결된 메시지 큐임



커널(운영체제 내부) 모습



위 프로세스 구조체 내의 파일 descriptor table은 execl()을 호출하여 cmds에서 cmd로 교체해도 그대로 존속함

- 즉, 동사무소에 신고된 주민번호(pid), 이름, 주소, 소유 재산(file handle) 등은 그대로인데 사람 몸만 다른 사람 몸으로 교체된 것과 동일함

cmdc.c: main() 함수

- 아래 두 문장을 삭제할 것

```
static void sig_child(int sig)
{
   printf("parent: sig_child: exit\n");
   exit(0); // 자식 cmdc3가 종료하면 시그널을 받아 부모 cmdc3가 여기서 종료
}
void dual_process(void) {
    if ((pid = fork()) < 0)
       ...
    else if (pid > 0) {    // 부모 프로세스: 키보드 입력 받아 서버에 전송
       ...
    } else { // 자식 프로세스
       ...
       printf("child: exit\n");
}   }
```

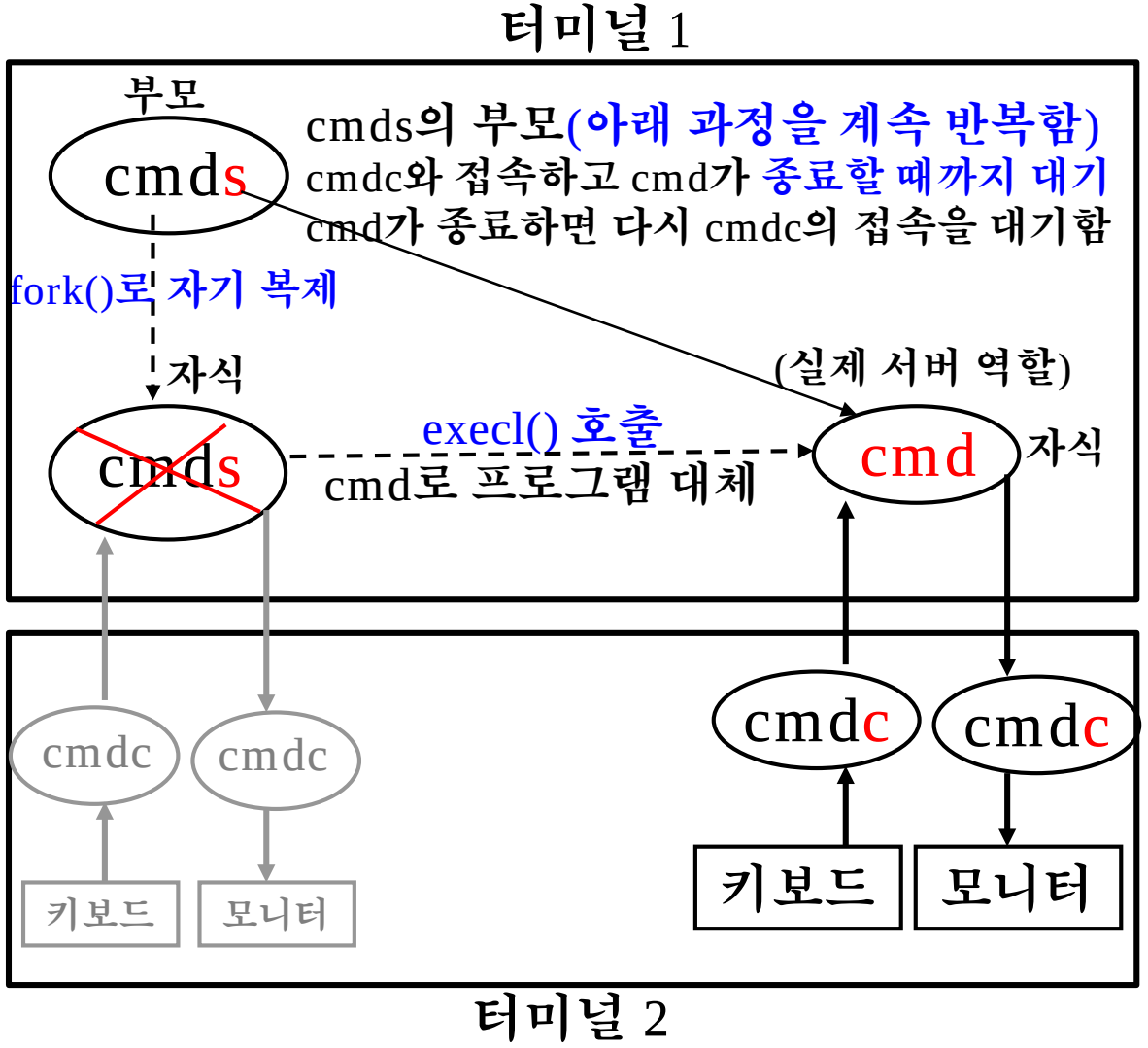
서버(cmds)와 클라이언(cmdc)의 실행

- make 실행한 후 터미널을 두 개 실행한다.
- 한쪽 터미널에서 서버 cmds을 먼저 실행
- 다른 쪽 터미널에서 클라이언트 cmdc를 실행
- cmdc에서 오른쪽 명령어들을 입력
  - 입력 받은 명령어를 서버에 보냄
  - 서버에서 보내준 처리된 명령어의 결과를 받음
  - 받은 처리 결과가 cmdc 화면에 출력되어야 함
  - 명령어가 처리되는 도중에 cmds의 화면에는 아무 것도 출력되지 않음

```
$ cat cmdc.c
$ cd ../cmd
$ ls -l
$ cd ../IPC
$ uname  -a
$ pwd
$ date
$ whoami
$ cp  cmdc.c  test.c
$ mv test.c tmp.c
$ ls
$ rm tmp.c
$ id
$ cal
$ cal 2016
$ ps
$ who
$ clear
$ whereis gcc
$ cd ../pr4
$ make clean
$ make
$ cd ../IPC
$ exit
```

연속형 서버: cmds의 전반적인 실행 과정

- 기존의 서버인 cmds는 cmdc가 종료하면 함께 종료한다. 즉, 한번만 서비스하고 종료한다.
- 이번에는 서버가 종료하지 않고 연속적으로 계속 cmdc의 접속을 계속 받아주는 연속형 서버 프로그램으로 변경하여 보자.



cmds.c의 main() 수정 - 1

- 기존의 cmds.c의 main() 함수를 아래와 같이 수정하라.

```
int main(int argc, char *argv[])
{
   printf("cmds: Server is Ready.\n");

   while (1) {
        // 클라이언트 프로그램의 접속을 기다린 후
        // 접속 요청이 오면 cmdc와 연결한다.
        connect_to_client();

        // fork()를 호출하여 현재 실행 중인 cmds를 복제하여
        // 자식 프로세스 cmds를 생성한다.
        pid_t pid = fork(); // pid: 프로세스 id

        if (pid < 0) // 에러 발생
            print_err_exit("cmds: fork()");
```

connect_to_client()
in_fd = open("fifo_c_to_s", O_RDONLY);
out_fd = open("fifo_s_to_c", O_WRONLY);

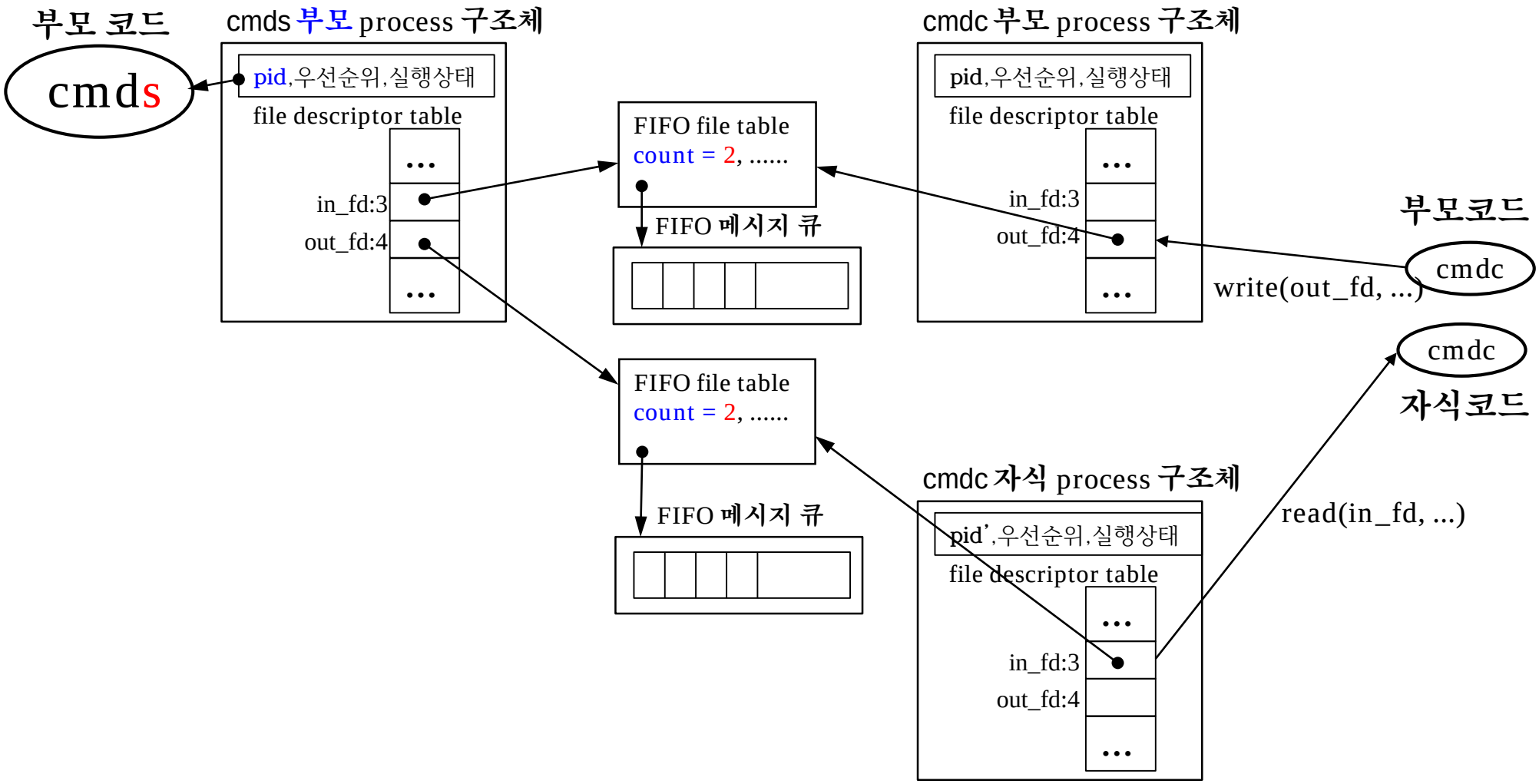
cmds.c의 main() 수정 - 2

- man 명령어를 이용하여 fork(), execl(), waitpid() 함수들의 헤드 파일을 include 하라.

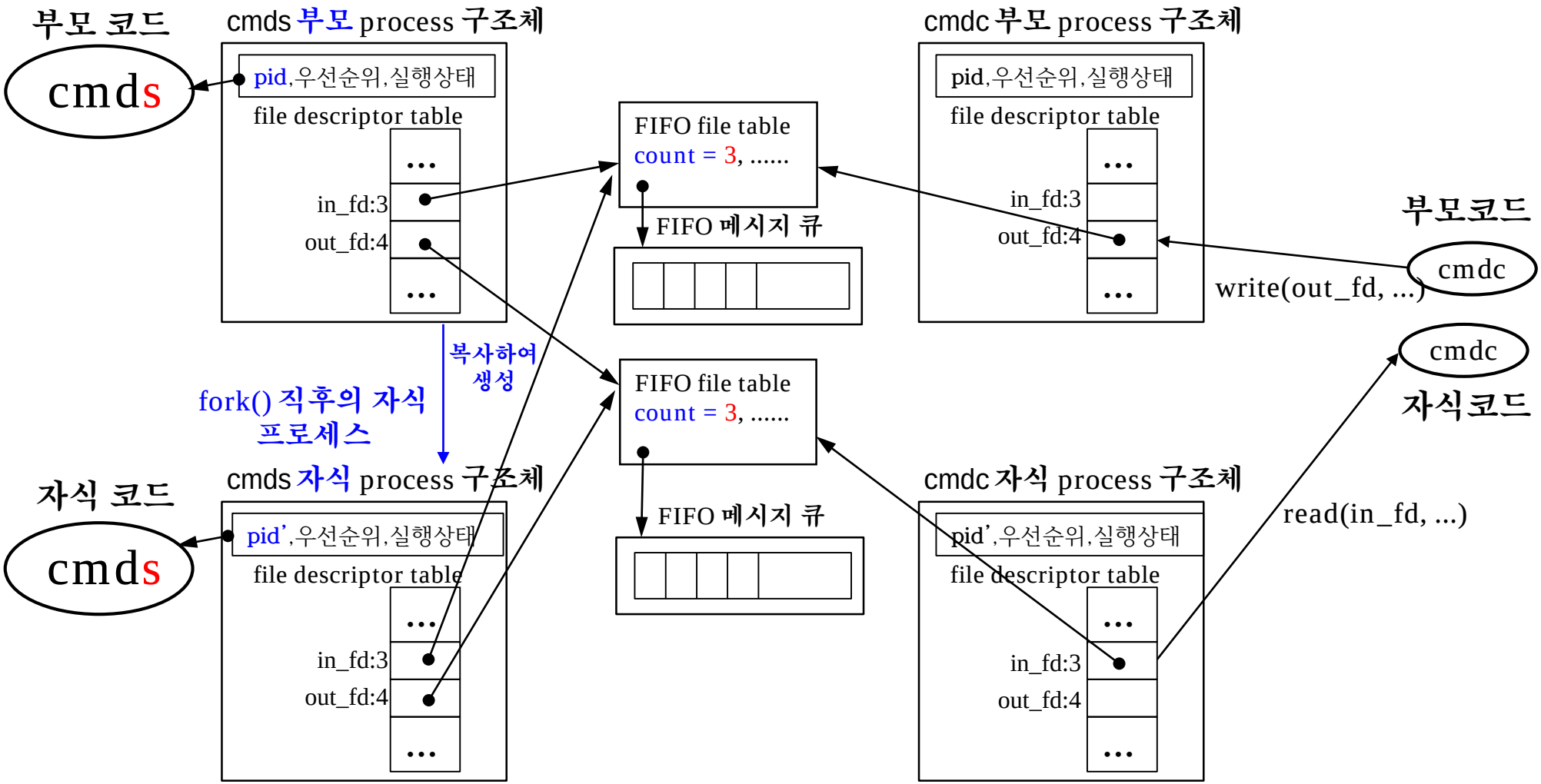
```
        // 자식 프로세스는 기존의 cmd 프로그램으로 프로세스 이미지(실행 함수들)로 교체함
        else if (pid == 0){ // 자식 프로세스: execl() 후 계속 cmdc와 통신함(연결유지)
            printf("client connected: cmd(pid %d) exec\n", getpid());
            duplicate_IO(); // 표준 입출력을 cmdc와 연결된 fifo로 교체함
            execl("../cmd/cmd", "cmd", NULL);
        }


       // 부모 프로세스는 자식(cmd)이 종료될 때까지 대기(자식은 cmdc와 접속되어 있음)
        else { // pid > 0 경우, 즉 부모 프로세스
            dis_connect(); // cmdc와 연결된 접속을 모두 닫음(자식이 대신 통신하므로)
            waitpid(pid, NULL, 0); // 자식(pid)이 종료될 때까지 대기
            printf("disconnected: cmd(pid %d) terminated\n", pid);
            sleep(2);                  // cmdc가 완전히 종료되길 잠시 기다림
        }
    }
}
```

```
duplicate_IO()
  dup2(in_fd, 0);
  dup2(out_fd, 1);
  dup2(out_fd, 2);
```

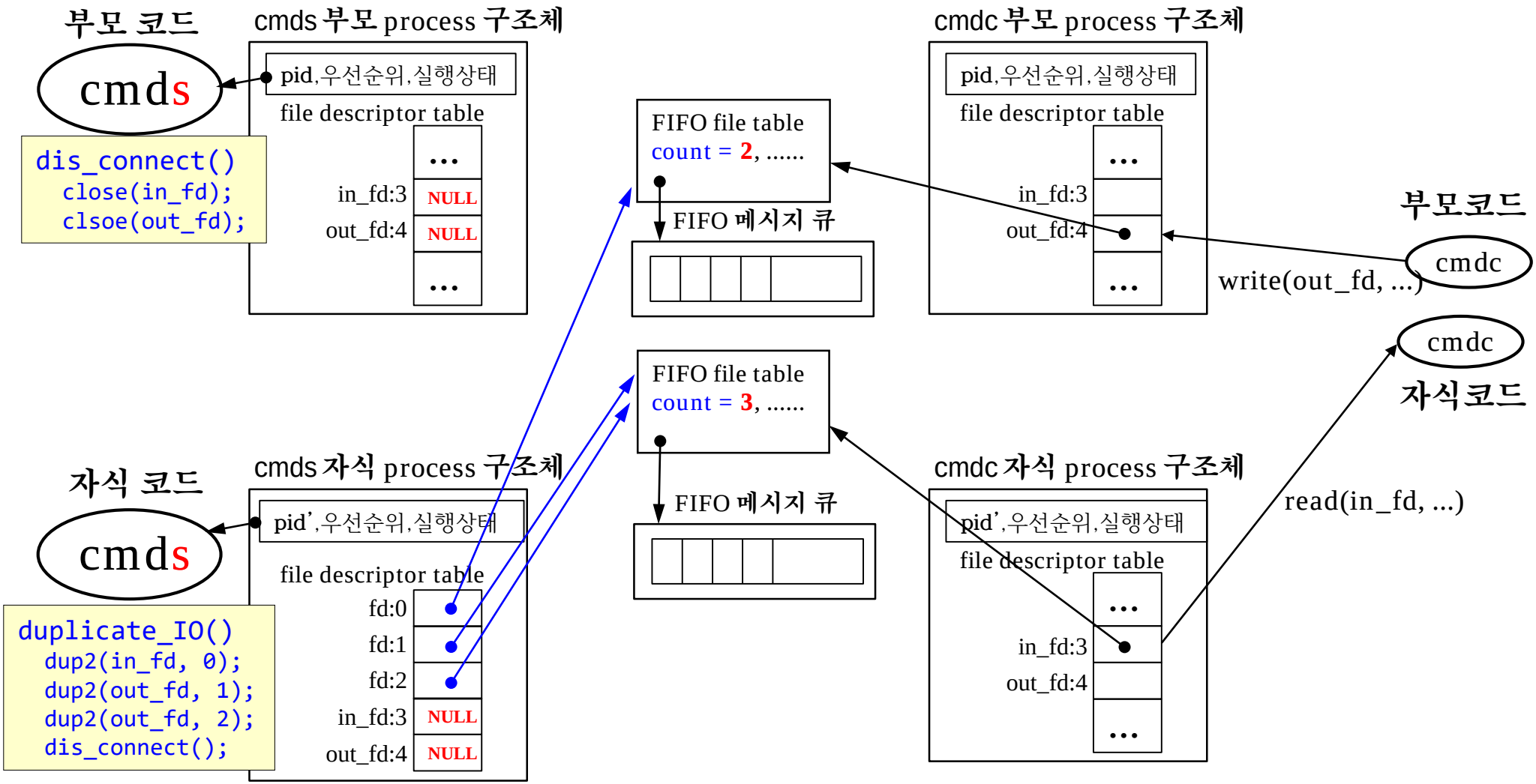
connect_to_client() 직후의 운영체제 내부 모습



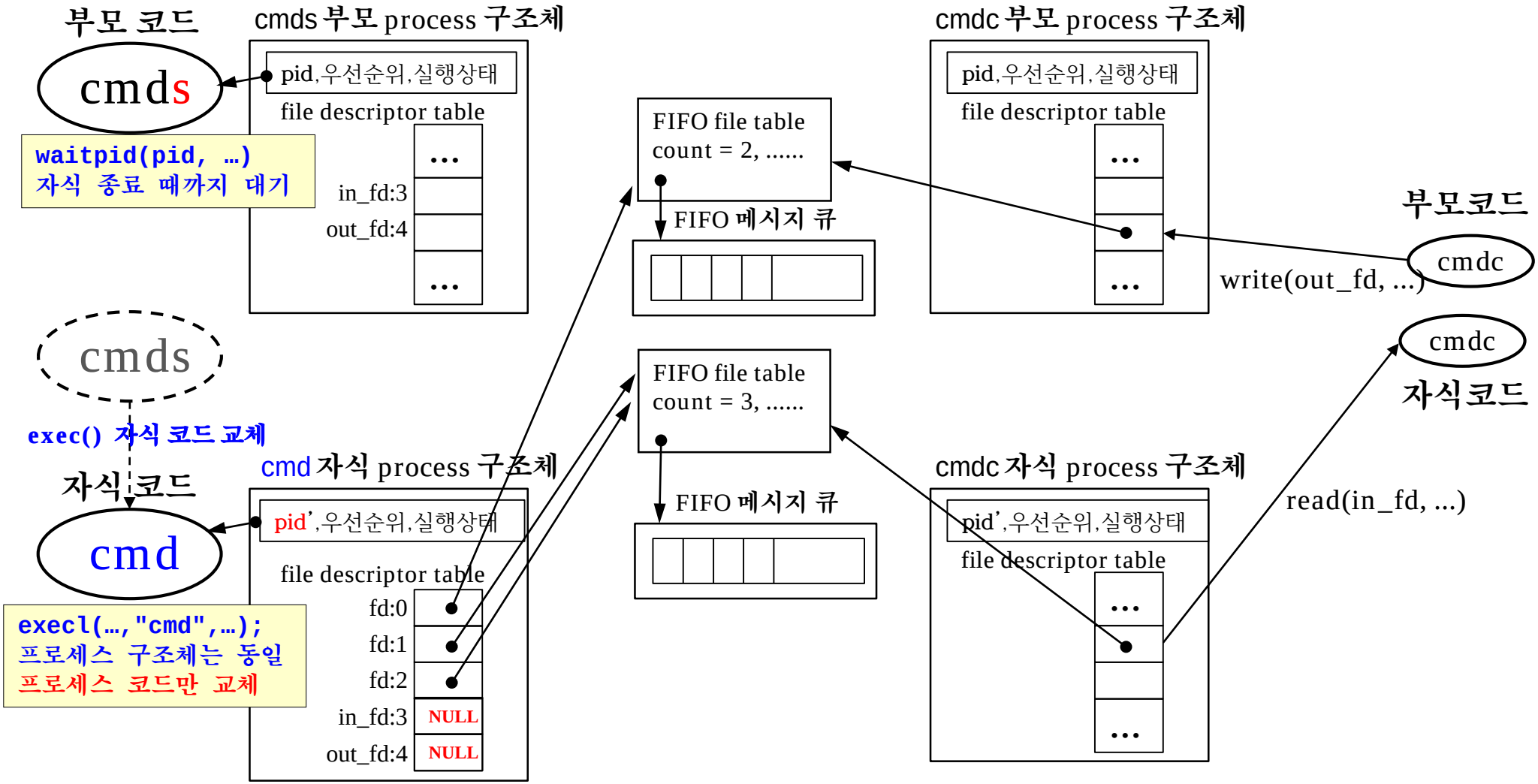
fork() 이후의 커널(운영체제 내부) 모습



dis_connect(), duplicate_IO() 이후의 커널 모습



waitpid(), exec() 이후의 커널 모습



cmds 및 cmdc의 실행

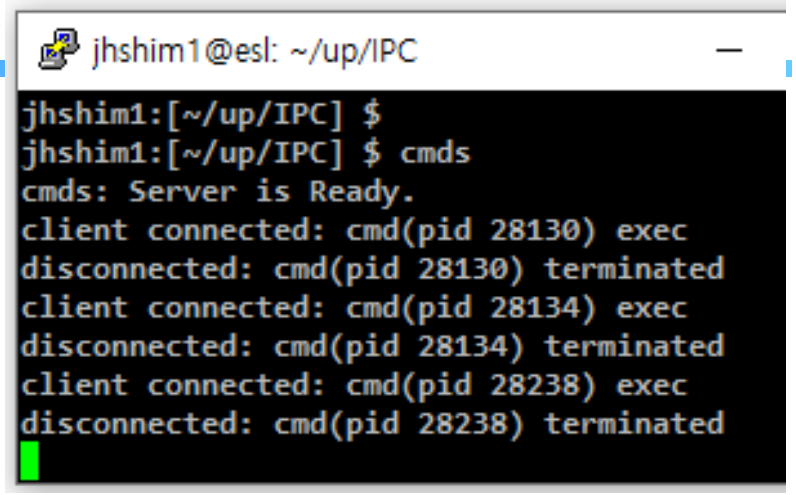
- 터미널 1에서 cmds를 먼저 실행
  - cmds를 종료하고자 할 때는 Ctrl+C로 종료

```
$ cmds
```

- 터미널 2(다른 터미널)에서 cmdc를 실행

```
$ cmdc
</home/……/up/IPC> 1:
```

  - 이후 여기에서 오른쪽 명령어들을 입력하면
  - 결과가 여기에 출력됨
  - cmdc에서 exit 명령어 입력하면 cmdc는 종료하지만, cmds는 여전히 살아 있음
  - 터미널 2(다른 터미널)에서 다시 cmdc를 실행
  - 여러 번 cmdc를 종료해도 cmds는 여전히 살아 있음




```
# cmdc 실행
$ ls -l
$ exit

# cmdc 재실행
$ date
$ exit

# cmdc 재실행
$ whoami
$ exit
```

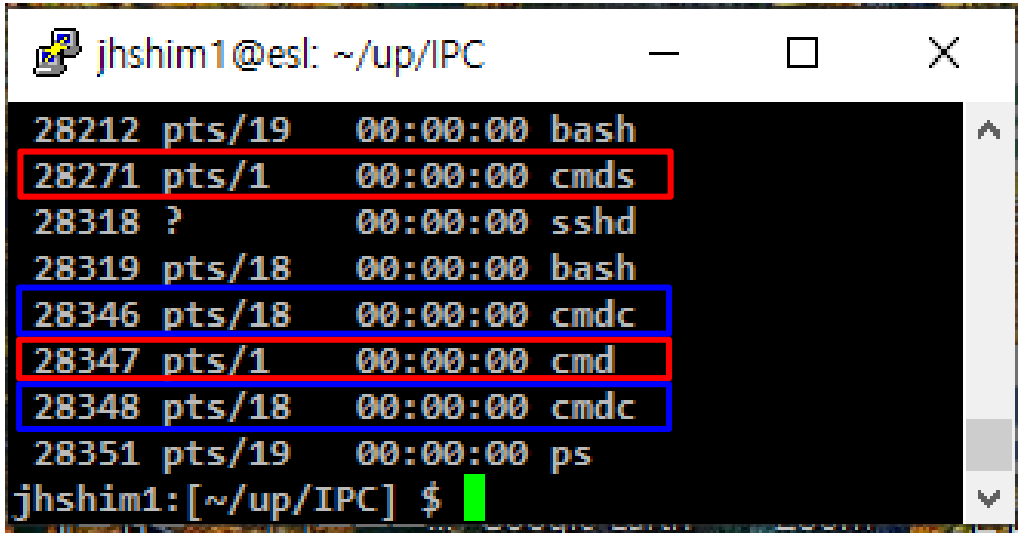
cmds, cmd, cmdc 프로그램들의 실행 확인

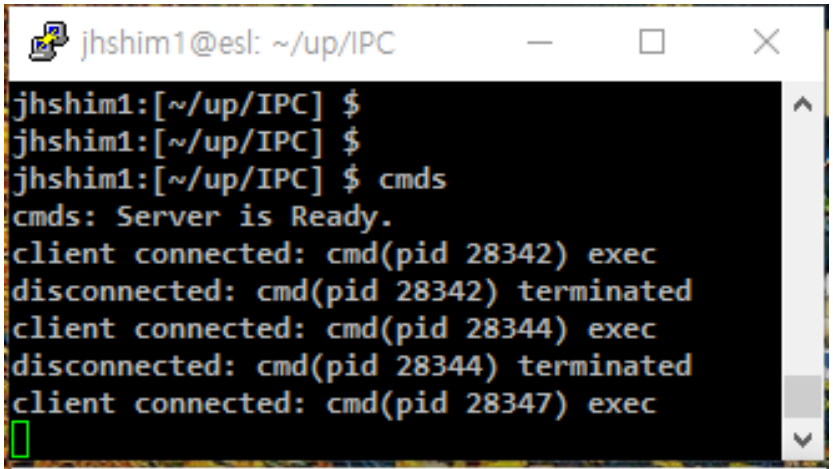
- 터미널 1에서 cmds를 먼저 실행
- 터미널 2(다른 터미널)에서 cmdc를 실행

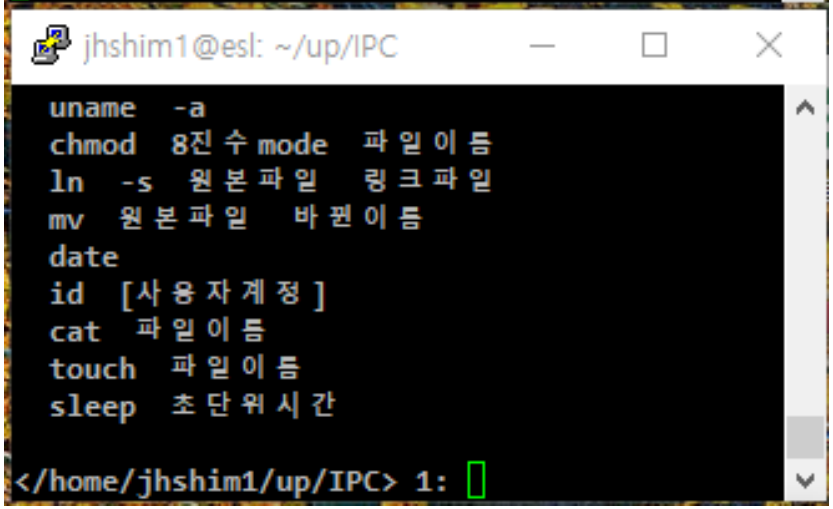
```
$ cmdc
</home/……/up/IPC> 1:
```

- 터미널 3에서 ps 명령어를 실행하면 아래처럼 현재 실행 중인 cmds, cmdc, cmd를 확인할 수 있음

```
$ ps -u 자신의계정
```







cmds의 백그라운드 실행

- 터미널 1에서 cmds를 먼저 실행

```
$ cmds &
$ jobs // 백그라운드로 실행되는 프로그램 보여 줌
       // cmds가 보여야 함
$ ps -f -u jhshim(자신의 계정이름)
       // 자신이 실행한 모든 프로세스 리스트를 보여줌
       // cmds가 보여야 함
```

- 터미널 2(다른 터미널)에서 cmdc를 연속적으로 실행

- 문제점
  - cmds의 터미널 1을 종료(터미널의 오른쪽 위쪽 x 버튼을 누름)하면 cmds도 함께 종료됨 => 터미널 2의 cmdc도 종료됨
  - 우리가 원하는 것은 터미널이 종료되어도 cmds가 종료되지 않고 계속 서비스를 해 주길 바람
  - 데몬 프로세스의 필요성 대두