



윤성우의 열혈 TCP/IP 소켓 프로그래밍

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

Chapter 10. 멀티프로세스 기반의 서버 구현



Chapter 10-1. 프로세스의 이해와 활용

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

다중 접속 서버의 구현 방법들

- 멀티프로세스 기반 서버 다수의 프로세스를 생성하는 방식으로 서비스 제공
- 멀티플렉싱 기반 서버 입출력 대상을 묶어서 관리하는 방식으로 서비스 제공
- 멀티쓰레딩 기반 서버 클라이언트의 수만큼 쓰레드를 생성하는 방식으로 서비스 제공

다중 접속 서버란 둘 이상의 클라이언트에게 동시에 접속을 허용하여, 동시에 둘 이상의 클라이언트에게 서비스를 제공하는 서버를 의미한다.

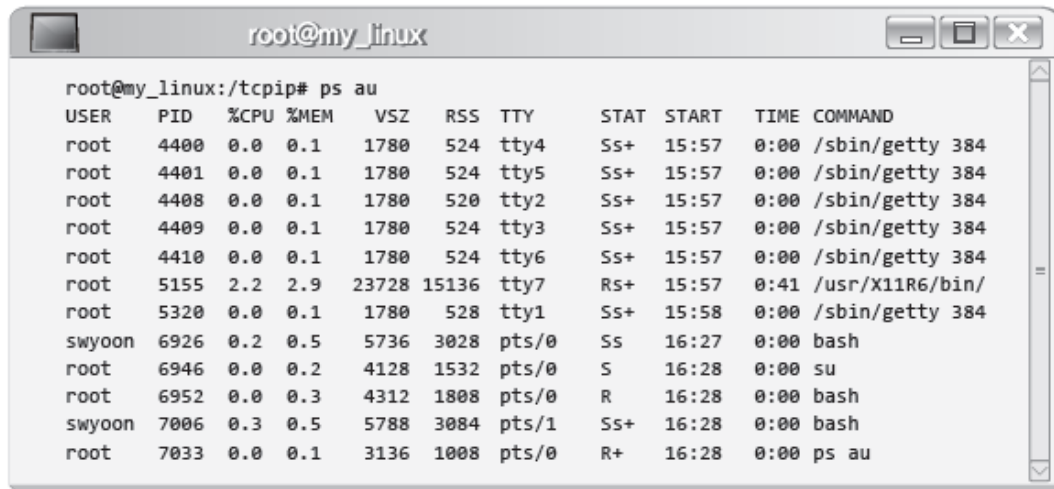
프로세스와 프로세스의 ID

▶ 프로세스란?

- ▶ 간단하게는 실행 중인 프로그램을 뜻한다.
- ▶ 실행 중인 프로그램에 관련된 메모리, 리소스 등을 총칭하는 의미이다.
- ▶ 멀티프로세스 운영체제는 둘 이상의 프로세스를 동시에 생성 가능하다.

▶ 프로세스 ID

- ▶ 운영체제는 생성되는 모든 프로세스에 ID를 할당한다.



```

root@my_linux:/tcpip# ps au
USER      PID    %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      4400    0.0  0.1   1780    524 tty4      Ss+   15:57   0:00 /sbin/getty 384
root      4401    0.0  0.1   1780    524 tty5      Ss+   15:57   0:00 /sbin/getty 384
root      4408    0.0  0.1   1780    520 tty2      Ss+   15:57   0:00 /sbin/getty 384
root      4409    0.0  0.1   1780    524 tty3      Ss+   15:57   0:00 /sbin/getty 384
root      4410    0.0  0.1   1780    524 tty6      Ss+   15:57   0:00 /sbin/getty 384
root      5155    2.2  2.9 23728 15136 tty7      Rs+   15:57   0:41 /usr/X11R6/bin/
root      5320    0.0  0.1   1780    528 tty1      Ss+   15:58   0:00 /sbin/getty 384
swyoon    6926    0.2  0.5   5736   3028 pts/0     Ss    16:27   0:00 bash
root      6946    0.0  0.2   4128   1532 pts/0     S     16:28   0:00 su
root      6952    0.0  0.3   4312   1808 pts/0     R     16:28   0:00 bash
swyoon    7006    0.3  0.5   5788   3084 pts/1     Ss+   16:28   0:00 bash
root      7033    0.0  0.1   3136   1008 pts/0     R+    16:28   0:00 ps au
  
```

fork 함수의 호출을 통한 프로세스의 생성

```
#include <unistd.h>
```

```
pid_t fork(void);
```

→ 성공 시 프로세스 ID, 실패 시 -1 반환

fork 함수가 호출되면, 호출한 프로세스가 복사되어
fork 함수 호출 이후를 각각의 프로세스가 독립적으로
실행하게 된다.

✓ Parent Process

```
int gval=10;
int main(void)
{
    int lval=20;
    lval+=5;
    gval++;
    pid_t pid=fork();
    if(pid == 0)
        gval++;
    else
        lval++;
    . . . . .
}
```

pid는 자식
프로세스 ID

실행

COPY

✓ Child Process

```
// gval은 11로 복사
int main(void)
{
    // lval은 25로 복사
    . . . . .
    pid_t pid=fork();
    if(pid == 0)
        gval++;
    else
        lval++;
    . . . . .
}
```

pid는 0!

실행

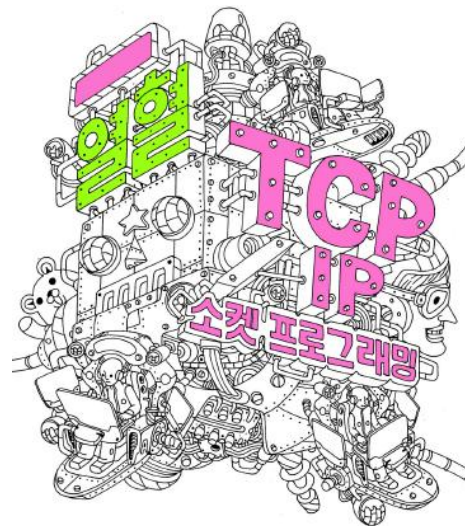
복사
발생지점

fork 함수 호출 이후의 반환 값은 다음과 같다. 따라서
반환 값의 차를 통해서 부모 프로세스와 자식 프로세
스의 프로그램 흐름을 구분하게 된다.

- 부모 프로세스 fork 함수의 반환 값은 자식 프로세스의 ID
- 자식 프로세스 fork 함수의 반환 값은 0

fork 함수를 호출한 프로세스는 **부모 프로세스**,

fork 함수의 호출을 통해서 생성된 프로세스는 **자식 프로세스**



Chapter 10-2. 프로세스 & 좀비 프로세스

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

좀비 프로세스의 이해



▶ 좀비 프로세스란?

- ▶ 실행이 완료되었음에도 불구하고, 소멸되지 않은 프로세스
- ▶ 프로세스도 main 함수가 반환되면 소멸되어야 한다.
- ▶ 소멸되지 않았다는 것은 프로세스가 사용한 리소스가 메모리 공간에 여전히 존재한다는 의미이다.

▶ 좀비 프로세스의 생성 원인

- ▶ 자식 프로세스가 종료되면서 반환하는 상태 값이 부모 프로세스에게 전달되지 않으면 해당 프로세스는 소멸되지 않고 좀비가 된다.

- 인자를 전달하면서 exit를 호출하는 경우
- main 함수에서 return문을 실행하면서 값을 반환하는 경우

자식 프로세스의 종료 상태 값이 운영체제에 전달되는 경로

좀비 프로세스의 생성 확인

```
pid_t pid=fork();
if(pid==0) // if Child Process
{
    puts("Hi, I am a child process");
}
else
{
    printf("Child Process ID: %d \n", pid);
    sleep(30); // Sleep 30 sec.
}

if(pid==0)
    puts("End child process");
else
    puts("End parent process");
return 0;
```

예제 zombie.c의 일부

```
root@my_linux:/tcpip# gcc zombie.c -o zombie
root@my_linux:/tcpip# ./zombie
Hi, I am a child process
End child process
Child Process ID: 10977
```

실행 결과

자식 프로세스의 종료 값을 반환 받을 부모 프로세스가 소멸되면, 좀비의 상태로 있던 자식 프로세스도 함께 소멸되기 때문에 부모 프로세스가 소멸되기 전에 좀비의 생성을 확인해야 한다.

```
root@my_linux:/tcpip# ps au
USER  PID  %CPU %MEM  VSZ   RSS TTY      STAT START   TIME COMMAND
. . . . .
root   10976 0.0  0.0   1628   368 pts/0    S+   20:26   0:00 ./zombie
root   10977 0.0  0.0     0     0 pts/0    Z+   20:26   0:00 [zom] <defunct>
. . . . .
```


좀비 프로세스의 소멸: wait 함수의 사용

예제 zombie.c의 일부

자식 프로세스
생성 종료

```
int status;
pid_t pid=fork();

if(pid==0)
{
    return 3;
}
else
{
```

자식 프로세스
생성 종료

```
    if(pid==0)
    {
        exit(7);
    }
    else
```

부모 프로세스
실행 영역

```
{
    printf("Child PID: %d \n", pid);
    wait(&status);
    if(WIFEXITED(status))
        printf("Child send one: %d \n", WEXITSTATUS(status));

    wait(&status);
    if(WIFEXITED(status))
        printf("Child send two: %d \n", WEXITSTATUS(status));
    sleep(30); // Sleep 30 sec.
}
}
```

```
root@my_linux:/tcpip# gcc wait.c -o wait
root@my_linux:/tcpip# ./wait
Child PID: 12337
Child PID: 12338
Child send one: 3
Child send two: 7
```

실행 결과

• WIFEXITED

자식 프로세스가 정상 종료한 경우 '참(true)'을 반환한다.

• WEXITSTATUS

자식 프로세스의 전달 값을 반환한다.

wait 함수의 경우 자식 프로세스가 종료되지 않은 상황에서는 반환하지 않고 **블로킹 상태에 놓인다**는 특징이 있다.

좀비 프로세스의 소멸2: waitpid 함수의 사용

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int * statloc, int options);
```

➔ 성공 시 종료된 자식 프로세스의 ID(또는 0), 실패 시 -1 반환

- pid 종료를 확인하고자 하는 자식 프로세스의 ID 전달, 이를 대신해서 -1을 전달하면 wait 함수와 마찬가지로 임의의 자식 프로세스가 종료되기를 기다린다.
- statloc wait 함수의 매개변수 statloc과 동일한 의미로 사용된다.
- options 헤더파일 sys/wait.h에 선언된 상수 WNOHANG을 인자로 전달하면, 종료된 자식 프로세스가 존재하지 않아도 블로킹 상태에 있지 않고, 0을 반환하면서 함수를 빠져 나온다.

wait 함수는 블로킹 상태에 빠질 수 있는 반면, waitpid 함수는 **블로킹 상태에 놓이지 않게끔** 할 수 있다는 장점이 있다.



waitpid 함수의 예

예제 waitpid.c

```
int main(int argc, char *argv[])
{
    int status;
    pid_t pid=fork();

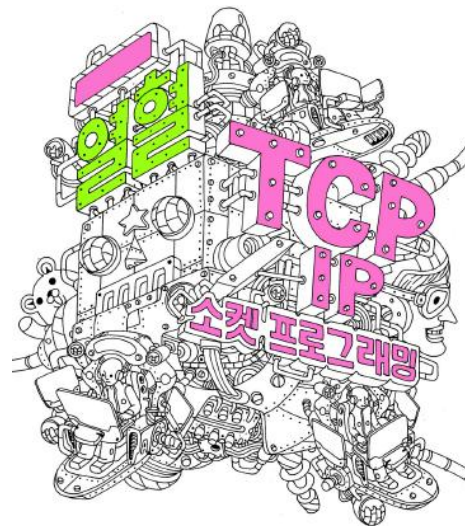
    if(pid==0)
    {
        sleep(15);
        return 24;
    }
    else
    {
        while(!waitpid(-1, &status, WNOHANG))
        {
            sleep(3);
            puts("sleep 3sec.");
        }

        if(WIFEXITED(status))
            printf("Child send %d \n", WEXITSTATUS(status));
    }
    return 0;
}
```

```
root@my_linux:/tcip# gcc waitpid.c -o waitpid
root@my_linux:/tcip# ./waitpid
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
Child send 24
```

실행 결과

waitpid 함수 호출 시 첫 번째 인자로 **-1**, 세 번째 인자로 **WNOHANG**가 전달되었으니, 임의의 프로세스가 소멸되기를 기다리되, 종료된 자식 프로세스가 없으면 0을 반환하면서 함수를 빠져나온다.



Chapter 10-3. 시그널 핸들링

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

시그널과 시그널 등록의 이해

▶ 시그널이란?

- ▶ 특정 상황이 되었을 때 운영체제가 프로세스에게 해당 상황이 발생했음을 알리는 일종의 메시지를 가리켜 시그널이라 한다.

등록 가능한 시그널의 예

• SIGALRM	alarm 함수호출을 통해서 등록된 시간이 된 상황
• SIGINT	CTRL+C가 입력된 상황
• SIGCHLD	자식 프로세스가 종료된 상황

▶ 시그널 등록이란?

- ▶ 특정 상황에서 운영체제로부터 프로세스가 시그널을 받기 위해서는 해당 상황에 대해서 등록의 과정을 거쳐야 한다.

시그널과 시그널 함수

```
#include <signal.h>
```

```
void (*signal(int signo, void (*func)(int)))(int);
```

➔ 시그널 발생시 호출되도록 이전에 등록된 함수의 포인터 반환

시그널 등록에 사용되는 함수

- 함수 이름 signal
- 매개변수 선언 int signo, void(*func)(int)
- 반환형 매개변수형이 int이고 반환형이 void인 함수 포인터

시그널 등록의 예

signal(SIGCHLD, mychild);	자식 프로세스가 종료되면 mychild 함수를 호출해 달라!
signal(SIGALRM, timeout);	alarm 함수호출을 통해서 등록된 시간이 지나면 timeout 함수호출!
signal(SIGINT, keycontrol);	CTRL+C가 입력되면 keycontrol 함수를 호출해 달라!

시그널 등록되면, 함께 등록된 함수의 호출을 통해서 운영체제는 시그널의 발생을 알린다.



시그널 핸들링 예제

예제 signal.c

```
void timeout(int sig)
{
    if(sig==SIGALRM)
        puts("Time out!");
    alarm(2);
}

void keycontrol(int sig)
{
    if(sig==SIGINT)
        puts("CTRL+C pressed");
}

int main(int argc, char *argv[])
{
    int i;
    signal(SIGALRM, timeout);
    signal(SIGINT, keycontrol);
    alarm(2);
    for(i=0; i<3; i++)
    {
        puts("wait...");
        sleep(100);
    }
    return 0;
}
```

이 예제에서 보이는 signal 함수는 운영체제 별로 동작방식의 차이를 보이기 때문에 이어서 설명하는 sigaction 함수를 대신 사용한다. signal 함수는 과거 프로그램과의 호환성을 유지하기 위해서 제공된다.

```
root@my_linux:/tcip# gcc signal.c -o signal
root@my_linux:/tcip# ./signal
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

실행 결과

시그널이 발생하면, sleep 함수의 호출을 통해서 블로킹 상태에 있던 프로세스가 깨어난다. 그래서 이 예제의 경우 코드의 내용대로 300초의 sleep 시간을 갖지 않는다.

sigaction 함수

```
#include <signal.h>
```

```
int sigaction(int signo, const struct sigaction * act, struct sigaction * oldact);
```

➔ 성공 시 0, 실패 시 -1 반환

- signo signal 함수와 마찬가지로 시그널의 정보를 인자로 전달.
- act 첫 번째 인자로 전달된 상수에 해당하는 시그널 발생시 호출될 함수(시그널 핸들러)의 정보 전달.
- oldact 이전에 등록되었던 시그널 핸들러의 함수 포인터를 얻는데 사용되는 인자, 필요 없다면 0 전달.

```
struct sigaction
{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

sigaction 구조체 변수를 선언해서, 시그널 등록 시 호출될 함수의 정보를 채워서 위의 함수 호출 시 인자로 전달한다. **sa_mask**의 모든 비트는 0, **sa_flags**는 0으로 초기화! 이들은 시그널관련 정보의 추가 전달에 사용되는데, 좀비의 소멸을 목적으로는 사용되지 않는다.

sigaction 함수의 호출 예

예제 sigaction.c

```
void timeout(int sig)
{
    if(sig==SIGALRM)
        puts("Time out!");
    alarm(2);
}

int main(int argc, char *argv[])
{
    int i;
    struct sigaction act;
    act.sa_handler=timeout;
    sigemptyset(&act.sa_mask);
    act.sa_flags=0;
    sigaction(SIGALRM, &act, 0);
    alarm(2);

    for(i=0; i<3; i++)
    {
        puts("wait...");
        sleep(100);
    }
    return 0;
}
```

실행 결과

```
root@my_linux:/tcpip# gcc sigaction.c -o sigaction
root@my_linux:/tcpip# ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

시그널 핸들링을 통한 좀비 프로세스의 소멸

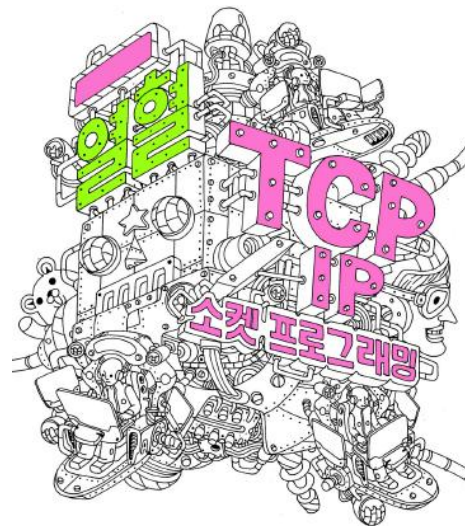
```
int main(int argc, char *argv[])
{
    pid_t pid;
    struct sigaction act;
    act.sa_handler=read_childproc;
    sigemptyset(&act.sa_mask);
    act.sa_flags=0;
    sigaction(SIGCHLD, &act, 0);
    . . . . .
```

SIGCHID에 대해서 시그널 핸들링을 등록하였으니, 이 때 등록된 함수 내에서 좀비의 소멸을 막으면 좀비 프로세스는 생성되지 않는다.

```
void read_childproc(int sig)
{
    int status;
    pid_t id=waitpid(-1, &status, WNOHANG);
    if(WIFEXITED(status))
    {
        printf("Removed proc id: %d \n", id);
        printf("Child send: %d \n", WEXITSTATUS(status));
    }
}
```

가장 기본적인 좀비의 소멸 코드로 함수가 정의되어 있다.



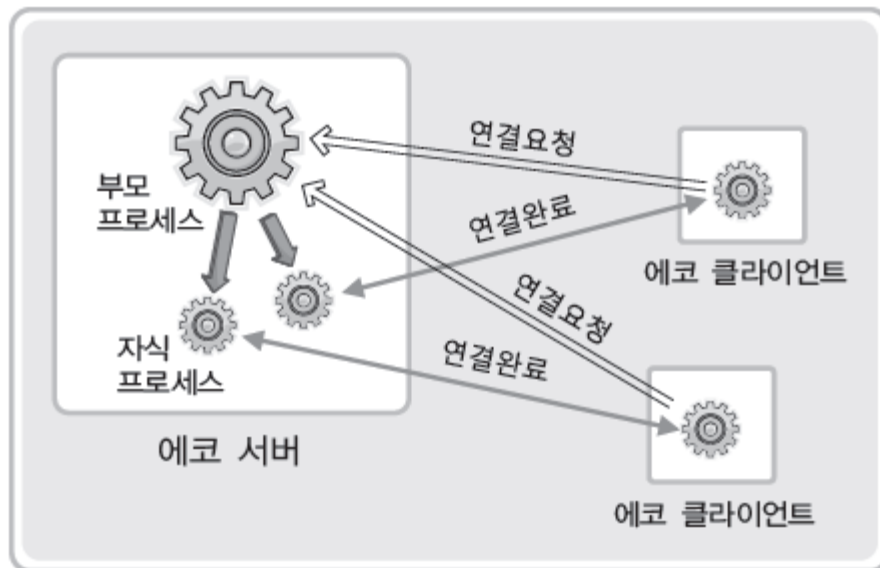


Chapter 10-4. 멀티태스킹 기반의 다중접속 서버

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

프로세스 기반 다중접속 서버 모델

프로세스 기반 다중접속 서버의 전형적인 모델



핵심은 연결이 하나 생성될 때마다 프로세스를 생성해서 해당 클라이언트에 대해 서비스를 제공하는 것이다.

- 1단계 에코 서버(부모 프로세스)는 accept 함수호출을 통해서 연결요청을 수락한다.
- 2단계 이때 얻게 되는 소켓의 파일 디스크립터를 자식 프로세스를 생성해서 넘겨준다.
- 3단계 자식 프로세스는 전달받은 파일 디스크립터를 바탕으로 서비스를 제공한다.

다중접속 에코 서버의 구현

예제 echo_mpserv.c의 일부

```
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==-1)
    {
        close(clnt_sock);
        continue;
    }
    if(pid==0) /* 자식 프로세스 실행영역 */
    {
        close(serv_sock);
        while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
            write(clnt_sock, buf, str_len);

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

클라이언트와 연결되면

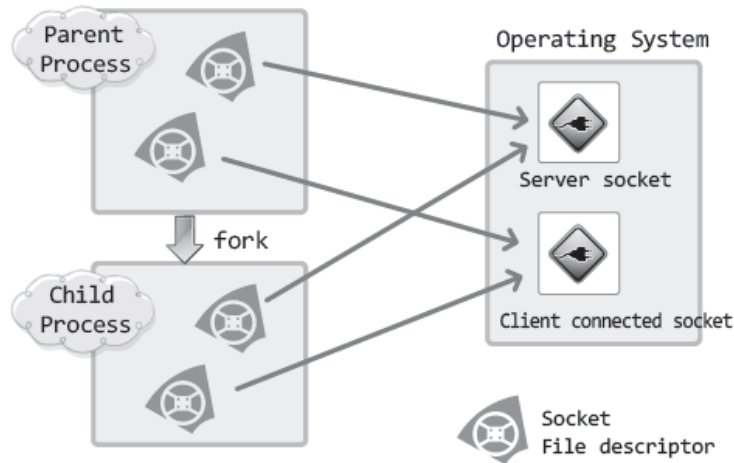
자식 프로세스를 생성해서

자식 프로세스가 서비스를 제공하게 한다.

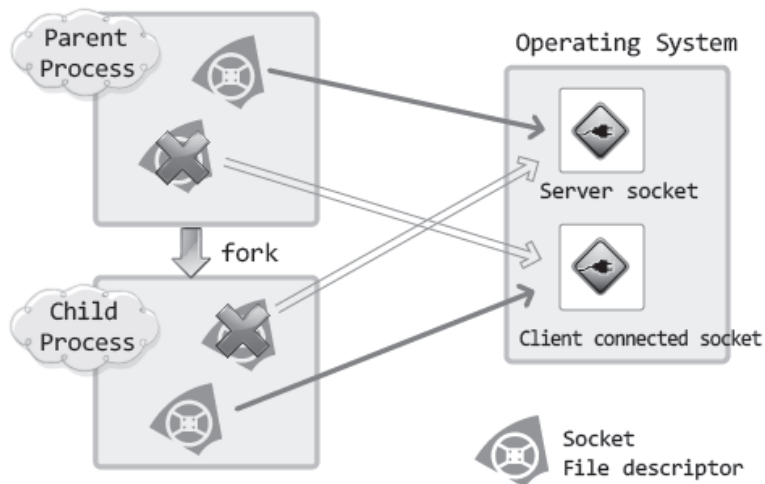
잠시 후에 설명

물론, 좀비 프로세스의 소멸을 위해서 앞서 보인, 좀비의 해결을 위한 함수의 호출과정은 거쳐야 한다. 이 코드에서는 이 내용을 보이지 않고 있다.

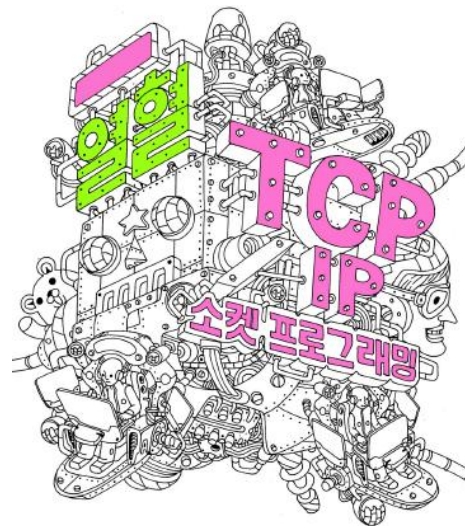
fork 함수호출을 통한 디스크립터의 복사



프로세스가 복사되는 경우 해당 프로세스에 의해 만들어진 소켓이 복사되는 게 아니고, 파일 디스크립터가 복사되어 왼쪽 그림의 형태가 된다.



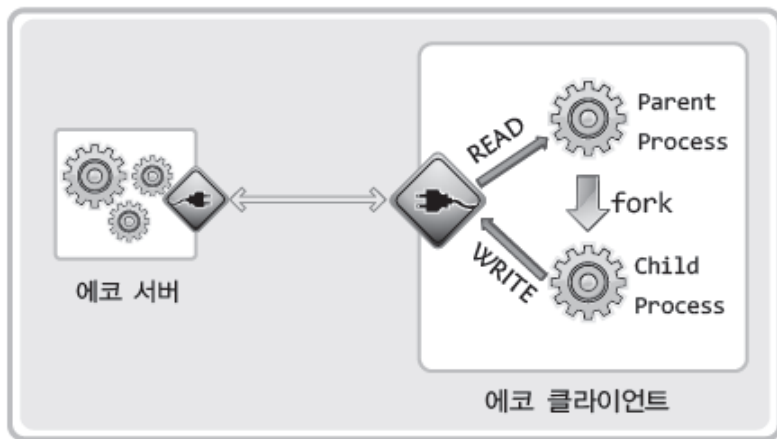
하나의 소켓에 두 개의 파일 디스크립터가 존재하는 경우, 두 파일 디스크립터 모두 종료되어야 해당 소켓 소멸 그래서 fork 함수호출 후에는 서로에게 상관 없는 파일 디스크립터를 종료한다.



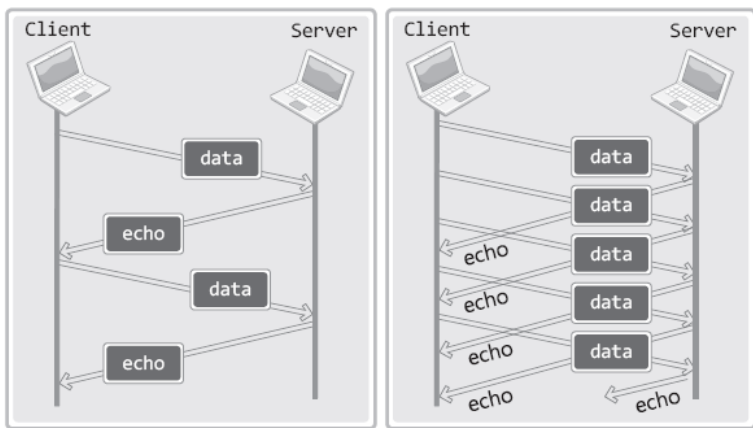
Chapter 10-5. TCP의 입출력 루틴 분할

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

입출력 루틴 분할의 이점과 의미



소켓은 양방향 통신이 가능하다. 따라서 왼쪽 그림과 같이 입력을 담당하는 프로세스와 출력을 담당하는 프로세스를 각각 생성하면, 입력과 출력을 각각 별도로 진행시킬 수 있다.



입출력 루틴을 분할하면, 보내고 받는 구조가 아니라, 이 둘이 동시에 진행 가능하다.

에코 클라이언트의 입출력 분할의 예

예제 `echo_mpclient.c`의 일부

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

pid=fork();
if(pid==0)
    write_routine(sock, buf);
else
    read_routine(sock, buf);
```

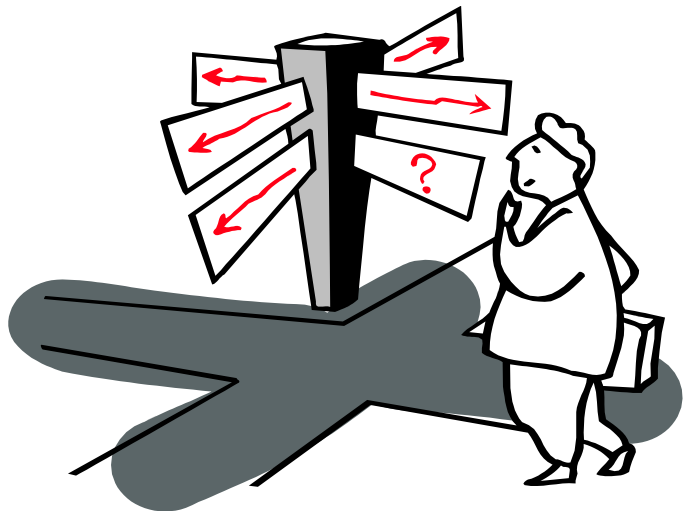
입력을 담당하는 함수와 출력을 담당하는 함수를 구분 지어 정의했기 때문에, 구현의 용이성에도 좋은 점수를 줄 수 있다.

물론, 인터랙티브 방식의 데이터 송수신을 진행하는 경우에는 이러한 분할이 큰 의미를 부여하지 못한다. 즉, 이러한 형태의 구현이 어울리는 상황이 있고, 또 어울리지 않는 상황도 있다.

```
void read_routine(int sock, char *buf)
{
    while(1)
    {
        int str_len=read(sock, buf, BUF_SIZE);
        if(str_len==0)
            return;

        buf[str_len]=0;
        printf("Message from server: %s", buf);
    }
}

void write_routine(int sock, char *buf)
{
    while(1)
    {
        fgets(buf, BUF_SIZE, stdin);
        if(!strcmp(buf, "q\n") || !strcmp(buf, "Q\n"))
        {
            shutdown(sock, SHUT_WR);
            return;
        }
        write(sock, buf, strlen(buf));
    }
}
```



Chapter 10이 끝났습니다. 질문 있으신지요?