Chapter 10

멀티프로세스 기반의 서버 구현

다중 접속 서버의 구현 방법들

다중 접속 서버의 정의

- 다중 접속 서버
  - ✓ 둘 이상의 클라이언트에게 **동시에 접속을 허용**
  - ✓ 둘 이상의 클라이언트에게 **동시에 서비스를 제공**하는 서버를 의미함

다중 접속 서버 구현 방법

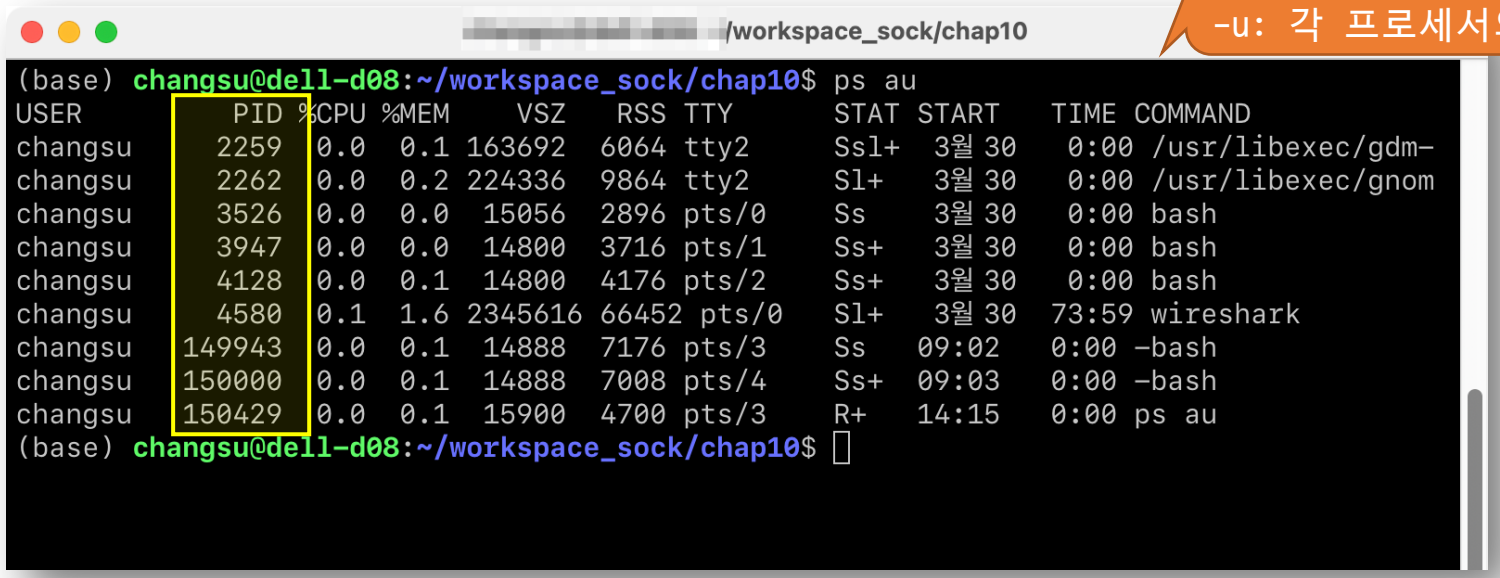
- 멀티프로세스 기반 서버: 다수의 프로세스를 생성하는 방식
- 멀티플렉싱 기반 서버: 입출력 대상을 묶어서 관리하는 방식
- 멀티쓰레딩 기반 서버: 클라이언트의 수만큼 쓰레드를 생성하는 방식

프로세스와 프로세스 ID

- 프로세스란?
  - 간단하게는 실행 중인 프로그램을 의미
  - 실행 중인 프로그램에 관련된 memory, resource 등을 총칭하는 의미
  - 멀티프로세스 운영체제는 둘 이상의 프로세스를 동시에 생성 가능

- 프로세스 ID (PID)
  - 운영체제는 생성되는 모든 프로세스에 ID를 할당함

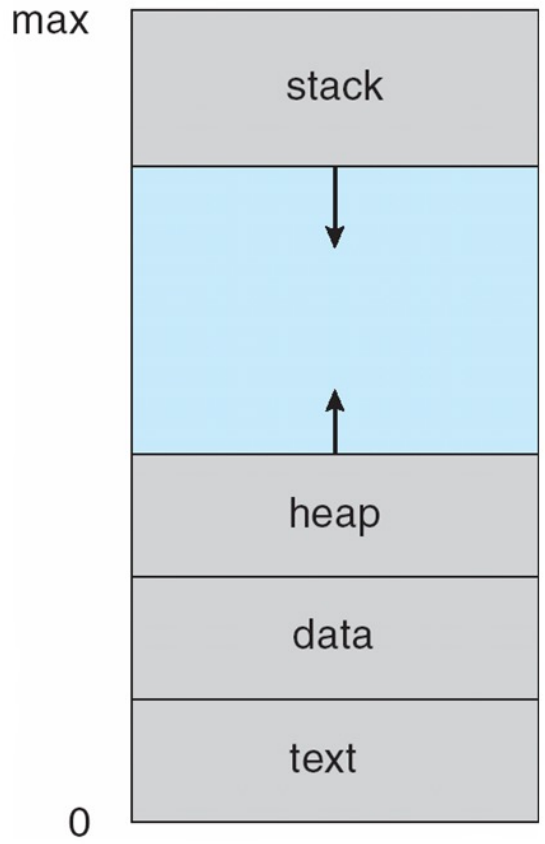
ps: process status
-a: 다른 사용자들의 프로세스 보여줌
-u: 각 프로세서의 사용자 이름과 시작 시간



프로세스 개념

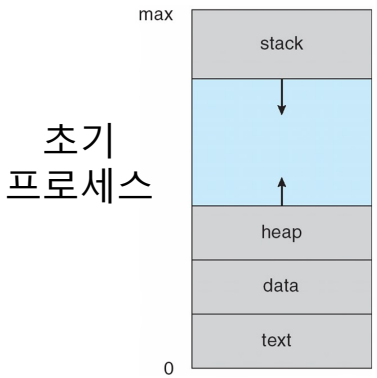
- 프로세스
  - 실행 중인 프로그램

- 프로세스 구성 요소
  - **Text section (Code section)**
    - 프로그램 소스 코드
  - **Data section**
    - 전역 변수나 정적 변수를 저장
  - **Heap section**
    - 동적 메모리 할당 영역
    - malloc()
  - **Stack section**
    - 임시 데이터 저장
    - 지역 변수들, 함수 파라미터, 리턴 주소

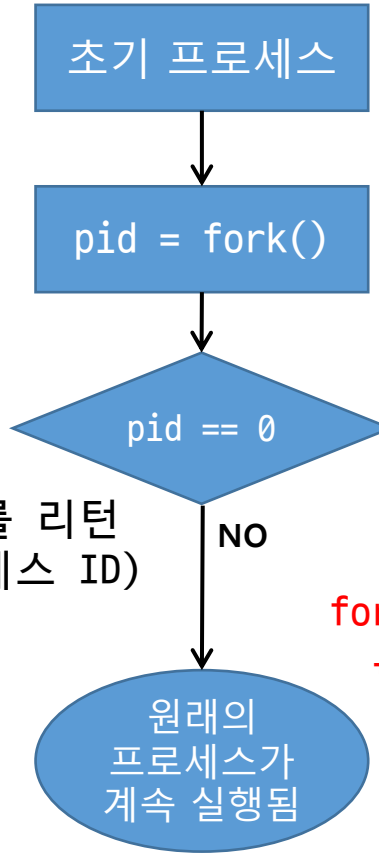


프로세스 구조

프로세스 생성: 복사본 생성 fork()



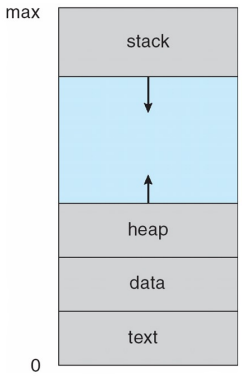
초기
프로세스




```
include <unistd.h>

pid_t fork(void);
-  실패: -1 리턴
-  성공: 새로운 PID를 리턴
-  이미 실행중인 프로세스를 복사함
-  메모리 영역까지 동일하게 복사(코드 포함)
```

- pid=0을 리턴(자식프로세스)
- 초기 프로세스의 메모리 영역을 복사
- 별도의 메모리 공간을 가짐

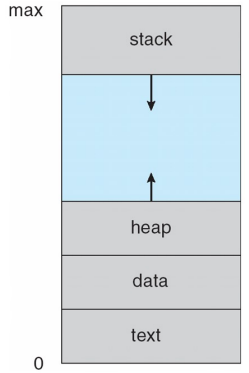
fork()함수 이후부터
독립적으로 실행



부모 프로세스

fork()함수를 호출한 프로세스

새
프로세스



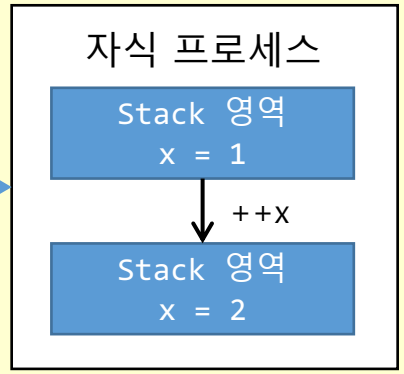
자식 프로세스

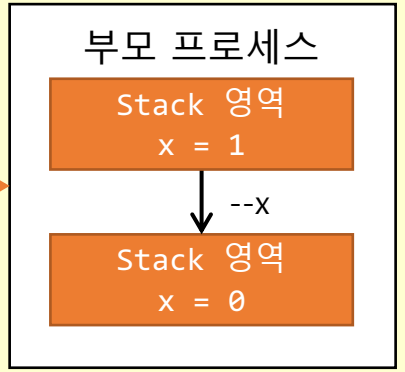
fork()함수 호출을 통해 생성(복제)된 프로세스

fork 예제 (forktest1.c)

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

void forkexample()
{
    int x = 1;
    pid_t pid;

    pid = fork();
    if(pid == 0)
        printf("Child has x= %d\n", ++x);
    else
        printf("Parent has x= %d, Child pid= %d\n", --x, pid);
}

int main()
{
    forkexample();
    return 0;
}
```

fork() 함수 호출
- 메모리 영역까지 동일하게 복사된 프로세스가 생성
- Text section (코드영역)도 동일하게 복사



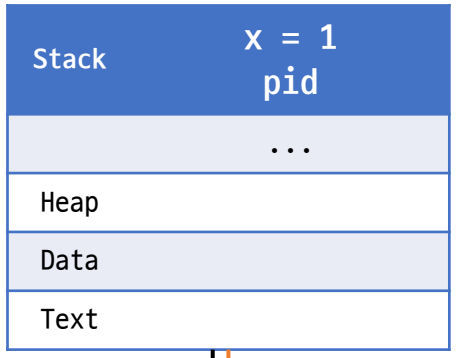



```
Parent has x = 0, Child pid=150472
Child has x = 2
```

fork() 함수 호출 이후,
부모 프로세스와 자식 프로세스는
독립된 메모리 공간을 가짐

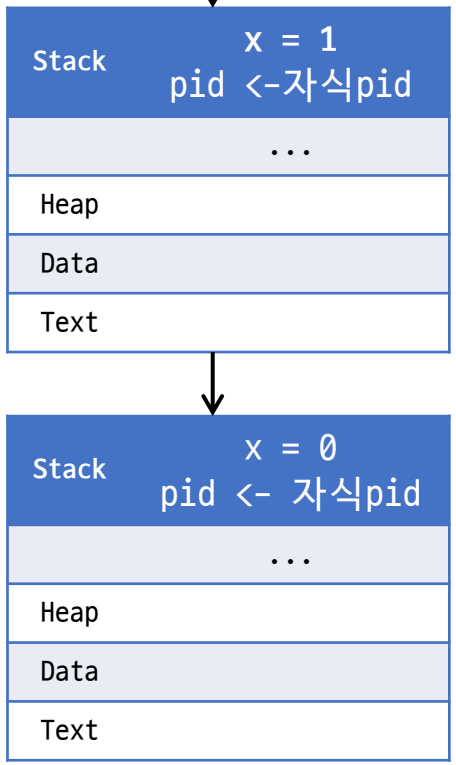
프로세스 복제 과정

초기 프로세스

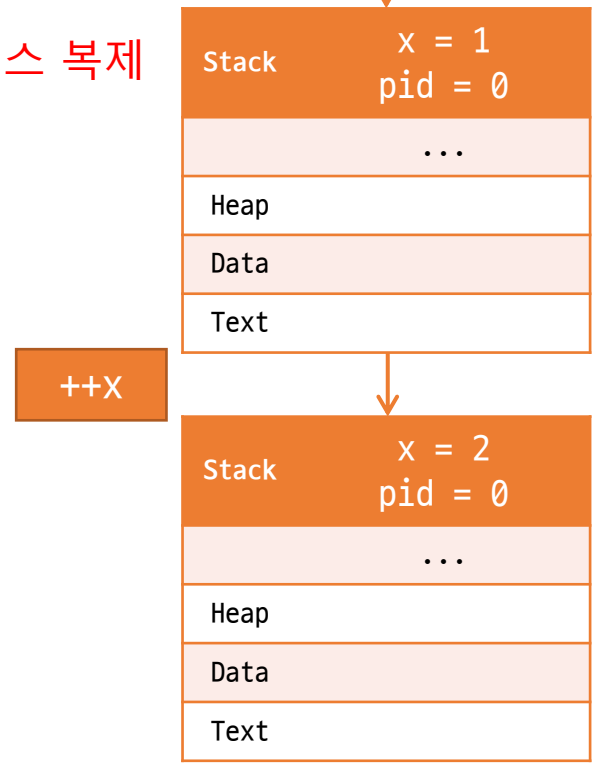



```
int x = 1;
pid_t pid;

pid = fork();
if(pid == 0)
    printf("Child has x= %d\n", ++x);
else
    printf("Parent has x= %d, Child pid= %d\n", --x, pid);
```

fork()호출



프로세스 복제



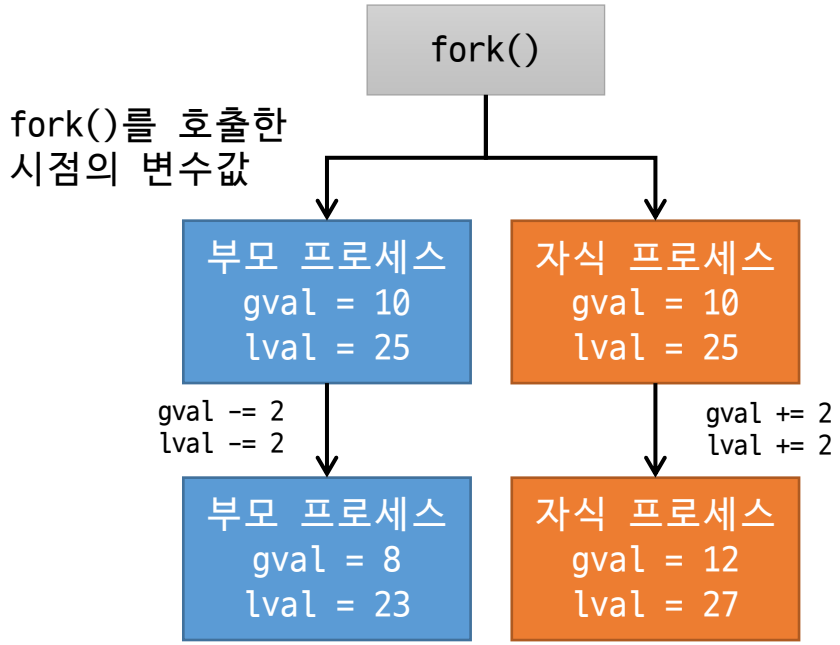
자식 프로세스

부모 프로세스

- fork()함수를 호출한 프로세스

fork.c

```c
#include <stdio.h>
#include <unistd.h>
int gval=10;

int main(int argc, char *argv[])
{
    pid_t pid;
    int lval=20;
    lval+=5;

    pid = fork();
    if(pid==0) // if Child Process
        gval += 2, lval += 2;
    else       // if Parent Process
        gval -= 2, lval -= 2;

    if(pid==0)
        printf("Child Proc: [%d, %d] \n", gval, lval);
    else
        printf("Parent Proc: [%d, %d] \n", gval, lval);

    return 0;
}
```



부모, 자식 프로세스의
전역 변수, 지역 변수 모두
각각 다른 값을 가짐

```
Parent Proc: [8, 23]
Child Proc: [12, 27]
```

fork.c 실행 과정

```c
#include <stdio.h>
#include <unistd.h>
int gval=10;

int main(int argc, char *argv[])
{
    pid_t pid;
    int lval=20;
    lval+=5;

    pid=fork();
```

pid = 자식프로세스

부모 프로세스

```c
  if(pid==0) // if Child Process
    gval+=2, lval+=2;
  else           // if Parent Process
    gval-=2, lval-=2;

  if(pid==0)
    printf("Child Proc: [%d, %d]\n", gval, lval);
  else
    printf("Parent Proc: [%d, %d]\n", gval, lval);
  return 0;
}
```

pid = 0

자식 프로세스

```c
  if(pid==0) // if Child Process
    gval+=2, lval+=2;
  else           // if Parent Process
    gval-=2, lval-=2;

  if(pid==0)
    printf("Child Proc: [%d, %d] \n", gval, lval);
  else
    printf("Parent Proc: [%d, %d] \n", gval, lval);
  return 0;
}
```

좀비 프로세스의 이해

- 좀비 프로세스란?
  - 실행이 완료되었음에도 소멸되지 않은 프로세스
  - 프로세스도 main 함수가 종료(반환)되면 소멸되어야 함
  - 소멸되지 않았다는 것은 프로세스가 사용한 리소스가 메모리 공간에 여전히 존재함을 의미

- 자식 프로세스가 종료되는 상황 2가지

  - 인자를 전달하면서 exit()를 호출하는 경우: exit(1)
  - main 함수에서 return 문을 실행하면서 값을 반환하는 경우: return 0

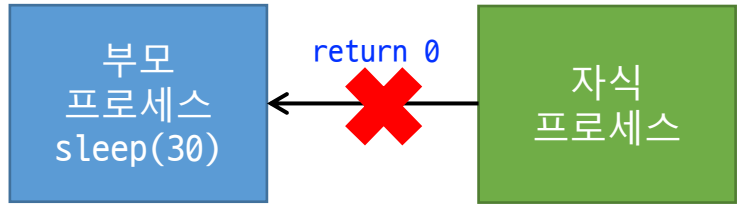
- 좀비 프로세스의 생성 원인
  - 자식 프로세스가 종료되면서 반환하는 상태 값이 부모 프로세스에게 전달되지 않으면
    - 해당 프로세스는 소멸되지 않고 좀비가 됨

zombie.c

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

int main(int argc, char *argv[])
{
    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;
}
```

부모 프로세스의 종료를 일부러 늦춤

자식 프로세스의 종료(리턴) 값을 받을 부모 프로세스가 소멸되면, 좀비 상태의 자식 프로세스도 소멸됨
- 부모 프로세스가 소멸되기 전에 좀비 상태 확인




```
$ ./zombie
Child Process ID: 150526
Hi I am a child process
End child process
End parent process
```

Zombie 프로세스 확인

- 부모 프로세스가 종료되지 않은 시점
  - 자식 프로세스(pid: 150526)가 zombie가 됨

```
$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
changsu     2259  0.0  0.1 163692  6064 tty2     Ssl+  3월30   0:00 /usr/libexec/gdm-way
changsu     4128  0.0  0.1  14800  4176 pts/2    Ss+   3월30   0:00 bash
changsu   150000  0.0  0.1  14888  7008 pts/4    Ss    09:03   0:00 -bash
changsu   150525  0.0  0.0   2772  1024 pts/3    S+    15:36   0:00 ./zombie
changsu   150526  0.0  0.0      0     0 pts/3    Z+    15:36   0:00 [zombie] <defunct>
changsu   150527  0.0  0.1  15900  4556 pts/4    R+    15:36   0:00 ps au
```

  - ➢ defunct 프로세스: 실행은 완료했지만, 부모 프로세스에게 완료 상태를 전달하지 못한 프로세스

- 부모 프로세스가 종료된 시점: [zombie] <defunct> 없어짐

```
$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
changsu     2259  0.0  0.1 163692  6064 tty2     Ssl+  3월30   0:00 /usr/libexec/gdm-way
changsu     4128  0.0  0.1  14800  4176 pts/2    Ss+   3월30   0:00 bash
changsu   150000  0.0  0.1  14888  7008 pts/4    Ss    09:03   0:00 -bash
changsu   150530  0.0  0.1  15900  4556 pts/4    R+    15:45   0:00 ps au
```

좀비 프로세스의 소멸 방법 #1: wait()함수

- wait() 함수
  - 자식 프로세스가 종료될 때까지 대기
  - 자식 프로세스가 종료한 상태를 가져옴

  - wait() 함수 호출 시, 이미 종료된 프로세스가 있는 경우
    - 자식 프로세스가 종료되면서 전달한 값이 status 변수에 저장
    - status 변수
      - exit함수의 인자값
      - main함수의 return값 저장

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

pid_t wait(int *status);
-> 성공 시 종료된 자식 프로세스의 ID, 실패 시 -1 반환
```

좀비 프로세스의 소멸 방법 #1: wait()함수

- 자식 프로세스 종료 상태 분석 매크로 함수
  - **int WIFEXITED (int status)**
    - 자식 프로세스의 종료 여부 확인 (wait if exited)
    - 주로 조건문에 사용

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

int WIFEXITED(int status);
-> 자식 프로세스가 정상 종료한 경우 non-zero 반환
```

  - **int WEXITSTATUS (int status)**
    - 자식 프로세스의 전달 값을 반환: 종료 시 리턴값 확인
    - wait exit status

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

int WEXITSTATUS(int status);
-> exit() 함수의 인자값, return 값이 반환
```

/usr/include/x86_64-linux-gnu/bits/waitstatus.h

```
/* If WIFEXITED(STATUS), the low-order 8 bits of the status. */
#define __WEXITSTATUS(status) (((status) & 0xff00) >> 8)

/* If WIFSIGNALED(STATUS), the terminating signal. */
#define __WTERMSIG(status) ((status) & 0x7f)

/* If WIFSTOPPED(STATUS), the signal that stopped the child. */
#define __WSTOPSIG(status) __WEXITSTATUS(status)

/* Nonzero if STATUS indicates normal termination. */
#define __WIFEXITED(status) (__WTERMSIG(status) == 0)
```

```
$ grep -rn WEXITSTATUS /usr/include/

/usr/include/x86_64-linux-gnu/bits/waitstatus.h:28:#define
            __WEXITSTATUS(status)   (((status) &
0xff00) >> 8)
 . . .
/usr/include/stdlib.h:44:# define WEXITSTATUS(status)
            __WEXITSTATUS (status)
```

좀비 프로세스의 소멸 방법 #1: wait.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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

    if(pid==0)
    {
        printf("Child process #1 is terminated(3)\n");
        return 3;
    }
    else
    {
        printf("Child PID: %d \n", pid);
        pid = fork();
        if(pid==0)
        {
            printf("Child process #2 is terminated(7)\n");
            exit(7);
        }
        else
        {
            printf("Child PID: %d \n", pid);
            wait(&status);
            if(WIFEXITED(status))
                printf("Child #1 sent: %d \n", WEXITSTATUS(status));

            wait(&status);
            if(WIFEXITED(status))
                printf("Child #2 sent: %d \n", WEXITSTATUS(status));
            sleep(30); // Sleep 30 sec.
        }
    }
    return 0;
}
```

첫 번째 자식 프로세스 생성

자식 프로세스 #1 생성 및 종료

두 번째 자식 프로세스 생성

자식 프로세스 #2 생성 및 종료

부모 프로세스 실행 영역

2개의 자식 프로세스 생성 -> 2번의 wait() 함수 호출

wait 함수의 경우, 자식 프로세스가 종료되지 않은 상황에서는 반환하지 않고 **블로킹 상태**에 놓임

```
$ ./wait
Child PID: 150694
Child process #1 is terminated(3)
Child PID: 150695
Child process #2 is terminated(7)
Child #1 sent: 3
Child #2 sent: 7
```

좀비 프로세스의 소멸 방법 #2: waitpid()함수

- waitpid() 함수

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

pid_t waitpid(pid_t pid, int *statloc, int options);
-> 성공 시 종료된 자식 프로세스의 ID, 실패 시 -1 반환
```

  - pid: 종료를 기다릴 자식 프로세스의 ID
    - -1을 전달하면, wait() 함수와 마찬가지로 임의의 자식 프로세스가 종료되기를 기다림
  - statloc: wait() 함수의 매개변수(status)와 동일한 의미
  - options
    - WNOHANG: Non-blocking (Wait No Hang의 의미), 종료된 자식 프로세스가 존재하지 않아도 블록킹 상태에 있지 않음 (0을 반환하면서 waitpid()함수를 빠져 나옴)

- **wait() 함수**
  - ✓ 호출된 시점에 종료된 프로세스가 없는 경우, 임의의 자식 프로세스가 종료될 때까지 blocking 상태에 놓임

- **waitpid() 함수**
  - ✓ Blocking 상태에 놓이지 않게 할 수 있는 장점이 있음
  - ✓ WNOHANG 옵션 사용 (wait no hang)

좀비 프로세스의 소멸 방법 #2: waitpid.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    int status;
    pid_t pid=fork();
    int cid = 0;

    if(pid==0)
    {
        sleep(15);
        return 24;
    }
    else
    {
        printf("Child pid= %d\n", pid);
        while((cid = waitpid(-1, &status, WNOHANG))==0)
        {
            sleep(3);
            puts("sleep 3sec.");
        }
        printf("exit while: cid= %d\n", cid);
        if(WIFEXITED(status))
            printf("Child sent %d \n", WEXITSTATUS(status));
    }
    return 0;
}
```

자식 프로세스의 종료를 지연 시킴

종료된 자식 프로세스가 없는 경우에 수행됨

```
$ ./waitpid
Child pid= 150719
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
exit while: cid= 150719
Child sent 24
```

5회 반복

waitpid(-1, &status, WNOHANG):
- 첫 번째 인자: -1
  . 임의의 프로세스가 종료 되는 것을 기다림
- 세 번째 인자(WNOHANG):
  . 종료된 자식 프로세스가 없으면 0을 반환하고 함수를 빠져나옴

시그널 핸들링

- 자식 프로세서의 종료 시점을 알 수 있는 방법은 없을까?
  - **시그널 핸들링**
    - 특정 시그널(메시지)과 연관된, 미리 정의된 작업을 수행하는 과정
- 시그널(signal)이란?
  - 운영체제가 프로세스에게 특정한 상황이 발생했음을 알리는 메시지
  - 등록 가능한 시그널의 예

```
• SIGALRM: alarm() 함수 호출을 통해 등록된 시간이 지난 상황
• SIGINT: ‘CTRL’ + ‘C’ 키가 입력된 상황
• SIGCHLD: 자식 프로세스가 종료된 상황
```

- 시그널 등록이란?
  - 특정 상황에서 운영체제로부터 프로세스가 시그널을 받기 위해서는 해당 상황에 대해 등록의 과정을 거쳐야 함

signal과 signal() 함수

- signal() 함수: signal 등록에 사용

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
-> 시그널 발생시 호출되도록 이전에 등록된 함수의 포인터 반환
```

  - 함수 이름:signal
  - 매개변수 선언: int signo, void (*func)(int)
  - 반환형: 매개변수가 int형이고 반환형이 void인 함수 포인터

- 시그널 등록의 예

```
• signal(SIGCHLD, mychild); 자식 프로세스가 종료되면 mychild 함수를 호출
• signal(SIGALRM, timeout): alarm 함수호출을 통해서 등록된 시간이 지나면 timeout 함수 호출
• signal(SIGINT, keycontrol): CTRL+C가 입력되면 keycontrol 함수를 호출
```

  - signal이 등록되고 해당 signal이 발생되면, 운영체제는 등록된 함수를 호출함

signal 함수 선언 분석

- typedef을 이용한 함수 포인터 정의
  - 사용자 정의 자료형처럼 함수 포인터를 정의해서 쉽게 사용하기 위함
    - 함수 포인터를 하나의 자료형으로 정의

```
$ man signal
```

```
SIGNAL(2)                    Linux Programmer's Manual                    SIGNAL(2)

NAME
       signal - ANSI C signal handling

SYNOPSIS
       #include <signal.h>
       typedef void (*sighandler_t)(int);
       sighandler_t signal(int signum, sighandler_t handler);
```

```
typedef void (*sighandler_t)(int)
```

- 함수 포인터 sighandler_t를 정의
- 매개 변수는 정수형 1개, 리턴 타입은 void인 함수 포인터를 sighandler_t로 정의

```
sighandler_t signal(int signum, sighandler_t handler);
```

- signal 함수의 매개 변수: 정수형 1개, sighandler_t 형태의 함수 포인터
- 리턴값: sighandler_t 타입 (함수 포인터를 리턴)

typedef을 이용한 함수 포인터 정의 예제

```c
#include <stdio.h>

// 함수 포인터 ptrfunc 정의
typedef void (*ptrfunc)(int, int);

void add(int a, int b)
{
    printf("a + b = %d\n", a + b);
}

void sub(int a, int b)
{
    printf("a - b = %d\n", a - b);
}

int main()
{
    ptrfunc handler;

    handler = add;
    handler(2, 3);

    handler = sub;
    handler(10, 4);
    return 0;
}
```

int형 변수 2개를 인자로 가지는 함수 포인터 ptrfunc 정의

포인터 ptrfunc에 add 함수 연결

포인터 ptrfunc에 sub 함수 연결

실행 결과

```
% ./"typedef_funptr"
a + b = 5
a - b = 6
```

alarm(), sleep() 함수 내용

- alarm(seconds) 함수
  - seconds 초 후에 프로세스에 SIGALRM을 전달
  - 반환값: SIGALRM 시그널이 발생하기까지 남아 있는 초 단위 시간

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

- sleep(seconds) 함수
  - 지정한 초 단위 시간만큼 대기

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

  - 반환값: 시간이 경과하면 0, 그렇지 않은 경우 남아 있는 시간을 리턴

signal.c

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

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

    alarm(2);
}

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

반복적으로 SIGALRM을 발생
- timeout() 함수 내부에서 다시 alarm() 함수를 호출해야 됨

```c
int main(int argc, char *argv[])
{
    int i;
    signal(SIGALRM, timeout);
    signal(SIGINT, keycontrol);

    alarm(2);

    for(i=0; i<3; i++)
    {
        printf("wait: %d\n", i);
        sleep(100);
    }

    printf("i=%d\n", i);
    return 0;
}
```

alarm(2) 호출로 sleep(100)이 동작하지 않음
- 시그널이 발생하면 sleep() 호출로 블로킹 상태에 있던 프로세스가 깨어남

```
$ ./signal
wait: 0
Time out!
wait: 1
Time out!
wait: 2
Time out!
i=3
```

sigaction 함수

- sigaction 함수

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigactgion *oldact);
-> 성공 시 0, 실패 시 -1 반환
```

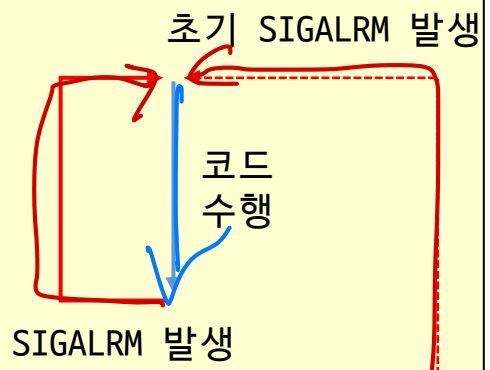
  - signo: signal 함수와 동일(시그널 정보)
  - act: 시그널 발생시 호출될 함수(시그널 핸들러)
  - oldact: 이전에 등록되었던 시그널 핸들러의 함수 포인터를 얻는데 사용 (필요 없으면 0 전달)

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

- sigaction() 함수
  - ✓ sigaction 구조체 변수를 선언
  - ✓ 시그널 등록시 호출될 함수 정보를 설정하여 인자로 전달
  - ✓ sa_mask의 모든 비트는 0
  - ✓ sa_flags는 0으로 초기화
- signal() 함수는 과거 프로그램과의 호환성을 위해 유지되고 있음

sigaction.c

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

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

    alarm(2);
}
```



반복적인 SIGALRM을 발생시키기 위해서 timeout() 함수 내부에서 alarm() 다시 호출

```c
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;
}
```

0으로 초기화.

시그널 번호

sigemptyset()
- sa_mask의 모든 비트를 0으로 초기화

```
$ ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

시그널 핸들링을 통한 좀비 프로세스의 소멸

```c
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);
     . . .

}
```

함수연결

시그널

- SIGCHLD 시그널
  - 자식 프로세스가 종료된 상황에서 발생
- SIGCHLD 시그널에 시그널 핸들러 등록
  - 등록된 시그널 핸들러 내부에서 좀비의 생성을 막음

```c
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));
    }
}
```

- waitpid(-1, &status, WNOHANG) 호출
  - 임의의 자식 프로세스의 종료를 기다림
  - 좀비 생성을 막음

remove_zombie.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

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 sent: %d \n", WEXITSTATUS(status));
    }
}

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);

    pid=fork();  // 자식 프로세스 생성
    if(pid==0)
    {
        puts("Hi! I'm child process #1");
        sleep(10);
        return 12;
    }
    else
    {
        printf("Child proc id: %d \n", pid);
        pid=fork(); // 자식 프로세스 생성
        if(pid==0)
        {
            puts("Hi! I'm child process #2");
            sleep(10);
            exit(24);
        }
        else
        {
            int i;
            printf("Child proc id: %d \n", pid);
            for(i=0; i<5; i++)
            {
                puts("Parent wait...");
                sleep(5);
            }
        }
    }
    return 0;
}
```

시그널 핸들러 등록

자식 프로세스 #1

자식 프로세스 #2

부모 프로세스 (25초 지연)

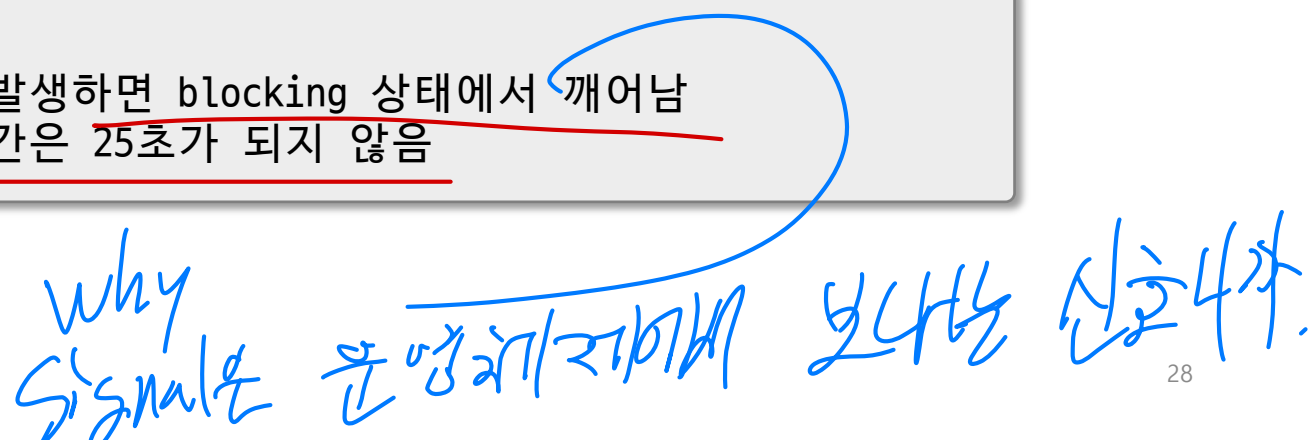
실행 결과

```
$ ./remove_zombie

Child proc id: 175089
Hi! I'm child process #1
Child proc id: 175090
Parent wait...
Hi! I'm child process #2
Parent wait...
Parent wait...
Removed proc id: 175089
Child sent: 12
Removed proc id: 175090
Child sent: 24
Parent wait...
Parent wait...
```

자식 프로세스 #1 종료

자식 프로세스 #2 종료

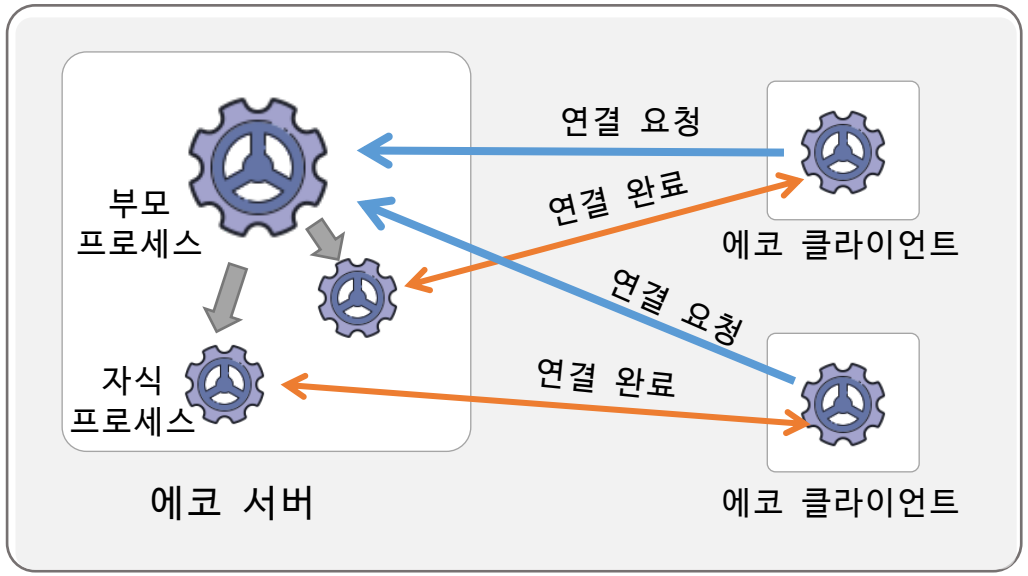
- 부모 프로세스의 지연 시간: 5초간 5회
  - 자식 프로세스의 종료 시그널(SIGCHLD)가 발생하면 blocking 상태에서 깨어남
  - 따라서 실제 부모 프로세스의 blocking 시간은 25초가 되지 않음



프로세스 기반의 다중접속 서버 모델

멀티 소켓 가는 과정.

- 프로세스 기반 다중접속 서버의 전형적인 모델



- 연결이 생성될 때마다 프로세스를 생성
  - ✓ 해당 클라이언트에게 서비스를 제공

경량화된 프로세스가 쓰레드

+ 메모리도 공유도 됨

- 에코 서버(부모 프로세스)는 accept() 함수 호출을 통해 연결 요청을 수락함
- 이때 얻게 되는 소켓 디스크립터를 자식 프로세스에게 전달
- 자식 프로세스는 전달받은 소켓 디스크립터를 바탕으로 서비스를 제공함

다중접속 에코 서버의 구현 (echo_mpserv.c 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
        printf("new client connected: %d\n", clnt_sock);
    pid = fork();
    if(pid == -1)
    {
        close(clnt_sock);
        continue;
    }
    if(pid == 0)
    {
        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 연결이 되면 자식 프로세스 생성
- 자식 프로세스에게 clnt_sock이 전달됨 (복제)

자식 프로세스가 clnt_sock을 이용하여 클라이언트와 통신

echo_mpserv.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 1024 //30
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;
    pid_t pid;
    struct sigaction act;
    socklen_t adr_sz;
    int str_len=0, state=0;
    char buf[BUF_SIZE];
```

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

  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");

  clnt_sock = 0;
```

SIGCHLD 시그널 등록
- 자식 프로세스 소멸 처리

echo_mpserv.c #2

```
    while(1)
    {
        adr_sz = sizeof(clnt_adr);
        clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
        if(clnt_sock == -1)
            continue;
        else
            printf("new client connected: %d\n", clnt_sock);
        pid = fork();
        if(pid == -1)
        {
            close(clnt_sock);
            continue;
        }
        if(pid == 0)
        {
            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);
    }
    close(serv_sock);
    return 0;
}
```

자식 프로세스 생성 및 클라이언트와 통신

serv_sock 종료
- fork() 수행 후 자식 프로세스에게 복사됨
- 자식 프로세스는 연결 설정을 위한 serv_sock이 필요 없음

clnt_sock 종료
- 부모 프로세스는 통신 용도의 clnt_sock이 필요 없음 (부모 프로세스는 accept()만 수행)

모든 자식프로세스의 종료처리

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

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

자식 프로세스 소멸 처리

다중접속 서버 및 클라이언트 실행

echo_mpserve.c 실행

```
서버
$ gcc echo_mpserv.c –o mpserv
$ ./mpserv 9190

new client connected: 4
new client connected: 4
client disconnected…
remove proc id: 110924
client disconnected…
remove proc id: 111011
```

부모 프로세스는 clnt_sock을 삭제하기 때문에 같은 번호가 출력됨

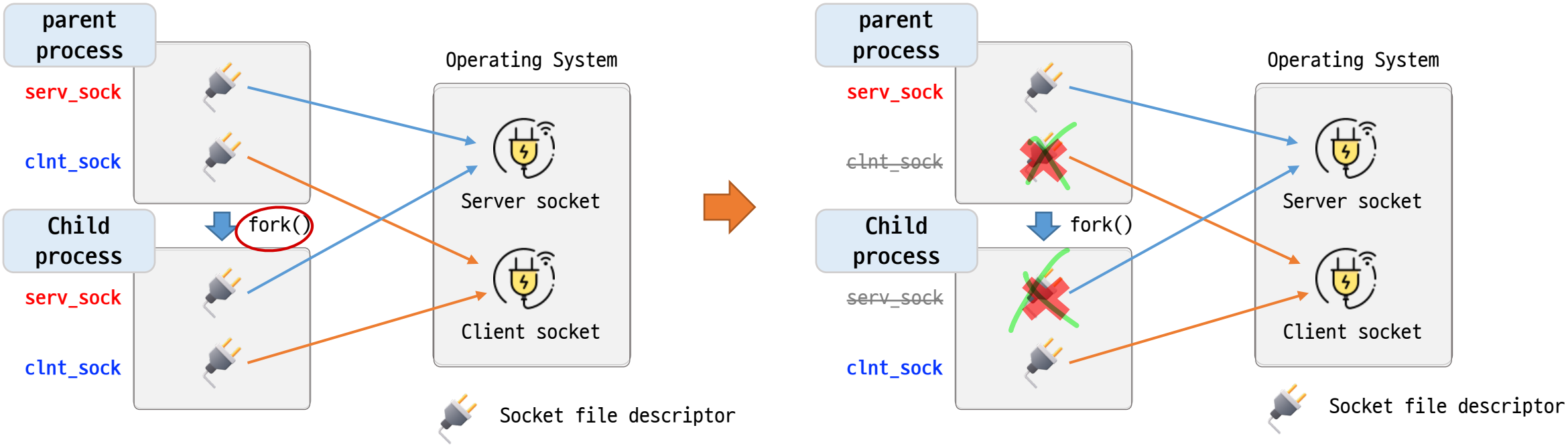
echo_client.c #1 실행

```
$ ./client 127.0.0.1 9190
Connected …….
Input message(Q to quit): Hi I’m first client
Message from server: Hi I’m first client
Input message(Q to quit): Bye
Message from server: Bye
Input message(Q to quit): Q
```

echo_client.c #2 실행

```
$ ./client 127.0.0.1 9190
Connected …….
Input message(Q to quit): Hi I’m second client
Message from server: Hi I’m second client
Input message(Q to quit): Good Bye
Message from server: Good Bye
Input message(Q to quit): Q
```

fork 함수 호출을 통한 소켓 디스크립터 복사

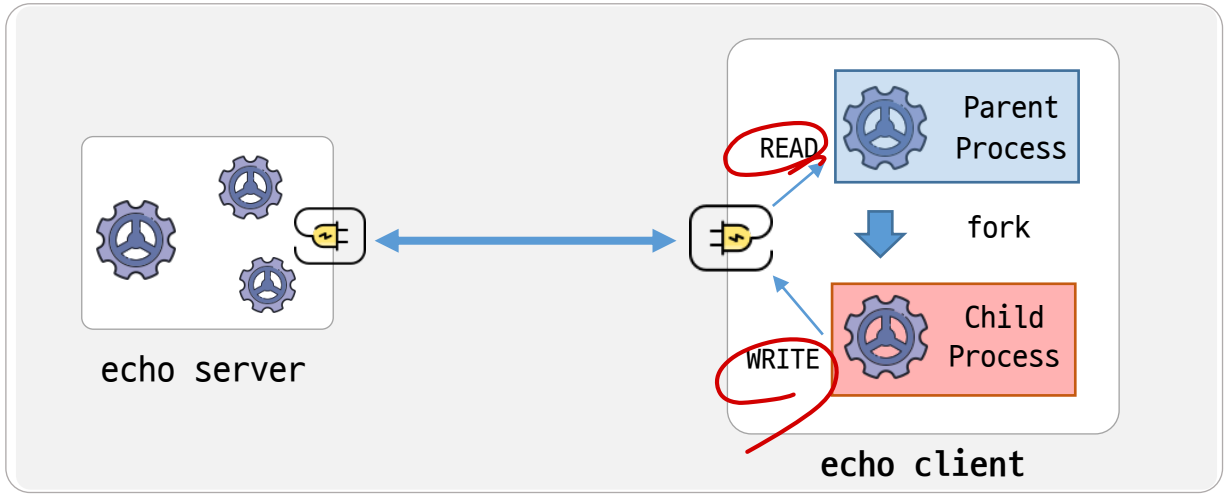


- 프로세스가 복사되는 경우(fork)
  - ✓ 소켓 자체가 복사되지 않음 (운영체제가 소켓을 소유)
  - ✓ 단순히 소켓 디스크립터가 복사됨

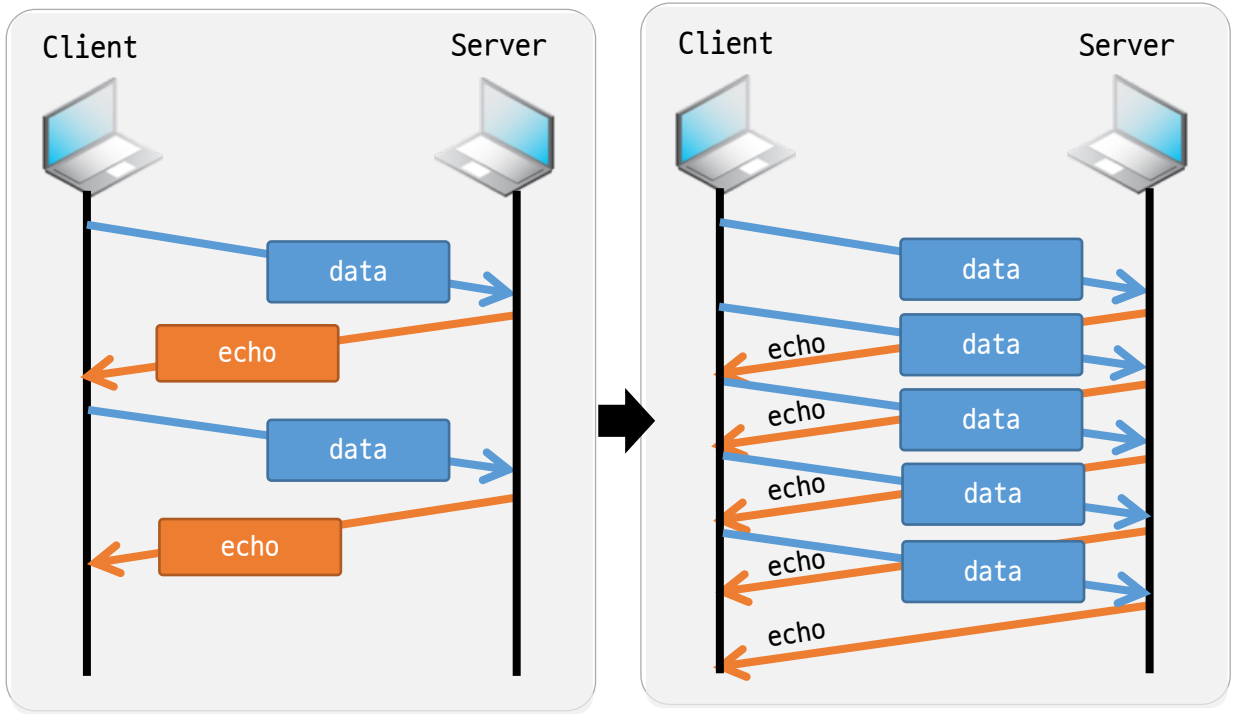
- 하나의 소켓에 두 개의 소켓 디스크립터가 존재하는 경우
  - ✓ 소켓 디스크립터 모두를 종료해야 해당 소켓이 소멸됨
  - ✓ fork 함수 호출 이후에는 서로에게 상관없는 소켓 디스크립터를 종료함

그 관찰 필요!!

입출력 루틴 분할의 이점과 의미



- 소켓은 양방향 통신이 가능함
  - ✓ 입력을 담당하는 프로세스 생성
  - ✓ 출력을 담당하는 프로세스 생성
  - ✓ 입력과 출력을 별도로 진행시킬 수 있음



- 입출력 루틴을 분할하면,
  - ✓ 입출력을 동시에 진행 할 수 있음

에코 클라이언트의 입출력 분할의 예

예제 echo_mpclient.c의 일부

```c
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);

close(sock);
```

자식 : 쓰기 (송신)

부모 : 읽기 (수신)

```c
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));
    }
}
```

```c
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);
    }
}
```

- 입력을 담당하는 함수와 출력을 담당하는 함수를 구분하여 정의
  - ✓ 구현상 편리함
- 이러한 형태의 구현이 어울리는 상황이 있고, 어울리지 않는 상황도 있음

echo_mpclient.c #1

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

#define BUF_SIZE 30
void error_handling(char *message);
void read_routine(int sock, char *buf);
void write_routine(int sock, char *buf);

int main(int argc, char *argv[])
{
    int sock;
    pid_t pid;
    char buf[BUF_SIZE];
    struct sockaddr_in serv_adr;

    if(argc!=3) {
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }
    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=inet_addr(argv[1]);
    serv_adr.sin_port=htons(atoi(argv[2]));

    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);

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

입출력 스트림 분리

echo_mpclient.c #2

```c
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));
    }
}
```

```c
void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

Questions?