<프로세스>

-프로세스가 생성하면 시스템에서 유일한 ID를 부여한다. (양의 숫자)
Process id : 0 (Swapper) – 스케쥴러 총괄

Process ID 1 : init process
-부트스트랩 과정의 마지막에 커널에 의해 실행된다.
-/etc/init 아니면 /sbin/init

Process id 2 : pagedaemon
-메인 메모리와 하드디스크 간의 페이징 업무를 담당하는 프로세스

-pid_t fork()
->리눅스 커널에서 프로세스를 만드는 유일한 방법이다.
->return 이 두개인데 0이면 자식 프로세스고, 0이 아닌 숫자는 parent 프로세스를 가리킨다.
(returns the child process id in parent process)

Child process 가 getppid()를 하면 parent process id를 알 수 있다

-부모와 자식은 둘 다 fock() call 다음의 명령어를 실행한다.
-fork 이후에 exec을 통해 child process가 실행 해야 하는 프로그램을 로딩 되도록 한다.
-fork()를 하면 프로그램이 두 개의 독립적인 프로세스로 나뉜다.
-child는 parent가 만든 같은 파일 디스크립터를 공유한다 (child에게 그대로 상속이 된다)
(파일 offset도 포함) 그러나, parent와 child 간의 누가 먼저 실행하는 지 스케쥴링은 명확하지 않다.

-fork() 이전에 열리진 파일을 parent와 child가 같이 쓴다고 하면 문제가 발생한다 (뒤죽박죽)

<Child에게 상속 되는 것>
-real user and group id, effective user and group id
-supplementary group id : 모든 유저는 적어도 한 개의 그룹에 속해야 하는데, /etc/passwd 의 GID에 명시되어 있다. 이것이 primary group id이다. 그리고, 유저는 한 개 이상의 그룹에 더 속할 수가 있는데 이것은 /etc/group에 있다. (supplementary group id)

-session id : 프로세스와 프로세스 간의 논리적인 연결(세션) 이 있다. 그리고 세션은 세션 ID가 있다. (메시지를 주고 받을 때 처럼 서로 통신을 할 때 무조건 한 개의 세션이 형성된다.)

<child에게 상속이 안 되는 것>
-file lock
-frok()이후의 return 값
-the process id's are different

Vfork()
-fork() 시스템 콜은 child process에게 별도의 메모리 공간을 할당 해야 한다
(유지 보수, 관리에 관한 일을 추가적으로 해야하고, 시스템 자원을 효율적으로 쓰지 못하게 되고 context switching도 해야 함) ->오버 헤드, 시간 많이 걸림
->vfork()는 새로운 프로세스만 만들고 새로운 일을 시키는데 ,부모의 메모리 공간을 그대로 이용한다
->parent 프로세스가 기존에 가지고 있던 코드가 완전히 무시 된다.(exec 이후에)
->child를 먼저 실행시킨다.
->경우에 따라서는 deadlock, 혼란을 가져올 수가 있어서 조심해야 함.

Exit()
-parent는 child process의 종료상태를 wait(), waitpid()로 받을 수 있다.
-부모 프로세스가 먼저 종료되면 자식 프로세스의 부모 프로세스는 init process가 된다.
(통상적으로 , child process가 먼저 종료 됨)
-부모 프로세스가 wait() 이나 waitpid()를 하지 않은 상태에서 자식 프로세스가 종료되면, 죽은 자식 프로세스의 정보가 커널에 저장되서 부모 프로세스가 wait(), waitpid()를 하면 넘겨준다
-부모 프로세스가 wait(), waitpid()를 계속 안하고 있으면 자식 프로세스는 종료됐지만 부모 프로세스가 거두어 들이지 않았기에 zombie process가 된다
(커널에 계속 저장은 된다) -> 많아 질수록 커널의 오버헤드가 커진다.

Wait()
-child process가 종료되면 커널은 parent에게 SIGCHLD signal을 보낸다
(wait이나 waitpid 시스템 콜을 등록하면, 커널이 SIGCHLD signal을 보낼 시 wait, waitpid system call 이 activate 된다)

#include <sys/wait.h>
Pid_t wait(int *statloc);
Pid_t waitpid(pid_t pid, int *statloc, int options);
->statloc : termination status(종료 상태)를 저장할 주소에 대한 포인터

-> return process id (성공) , -1 : 에러 , waitpid는 리턴값으로 0을 받을 수 있는데 WNOHANG option 에서 child 프로세스가 종료된 상태가 아니고 아직 실행 중인 것이 있을 때)

->waitpid() 에서는 명시한 특정한 pid 에 대해서만 기다린다
->반면에 wait()는 어느 하나 자식 프로세스가 종료되면 빠져나온다

종료 상태
1) WIFEXITED(statloc) : 정상 종료 (true)
WEXITSTATUS(stat_val) 로 종료 상태(신호)번호를 받을 수 있다.

2)WIFSIGNALED(statloc) : 비정상 종료 (by receipt of a signal)
WTERMSIG(status) 로 비정상 종료 신호 번호를 받을 수 있다.
WCOREDUMP(status) : true if a core file was generated

3)WIFSTOPPED: 비정상 종료 (자발적으로 잠시 중단)
WSTOPSIG(stat_val) 로 비정상 종료 신호 번호를 받을 수 있다.

Pid_t waitpid(pid_t pid, int *statloc, int options);
-특정 프로세스에 대해 기다린다.
->성공 시 return process id
-pid 첫번째 argument에서
1) pid = -1 이면, wait()시스템 콜이랑 똑같다 (임의의 자식 프로세스를 기다림)
2) pid > 0 이면, pid에 해당하는 프로세스만 기다린다.
3) pid = 0 이면 , parent process와 process group id가 같은 모든 자식 프로세스
4) pid < -1 이면, 특정 process 그룹 id가 pid의 절대값과 같은 모든 자식 프로세스

-options
1)WNOHANG : 기다리는 pid가 종료되지 않아서 즉시 종료 상태를 회수 할 수 없는 상황에서 호출자는 차단되지 않고 반환값으로 0을 받음

2)WUNTRACED : 중단된 자식 프로세스의 상태를 받음

Waitpid(pid, &status, 0) : 이면 따로 option을 준게 없음
Ex) waitpid(-1, &status, 0) : child

<Race conditions> : 여러 개의 프로세스가 공통의 자원이 공유되는 상황. 프로세스의 실행 순서에 따라 결과가 달라진다면 그것을 Race condition이라고 한다.
Ex) 프로세스의 실행 순서는 시스템이 결정한다 (자식과 부모 간의 실행도 예측할 수 없음)

-TELL : 프로세스 간의 통신 방법 중의 한 함수
Tellwait.h , apue.h 라이브러리 첨부해야 함

<exec() > : 프로그램 실행
: 새로운 프로그램을 로드하지만, 같은 PID 이다
: execvp(), execle(), execl(), execlp(), execv(), execve()를 합쳐서 exec() 시스템 콜이라 한다.

1) **int execl(const char *path, const char *arg0, ... Const char *argn, (char *) 0)**
->path에 지정한 경로명의 파일을 실행하며 arg0 ~ argn 을 인자로 전달한다

-> path : 실행 가능한 파일 경로
-> argv0 ~ argn : 파일에 전달한 argv 정보 (명령 인수), (char *)0 : NULL 포인터
-> 실패시 -1

Ex) execl("/bin/ls", "ls", argv[1], (char *)0);

2) **int execv(const char *path, const char *argv[]);**
->array처럼 넘긴다.
Ex)

```
Av[0] = "ls";
Av[1] = "-l";
Av[2] = "(char *)0"
Execv("/bin/ls", av);
```

3) int execle (const char *path, const char *arg0, argn.. , (char *)0, char *const envp[]);
Execl()에 환경 변수 정보를 전달하는 기능을 추가한 것

4) int execve (const char *path, char *const argv[], char *const envp[])

4) execlp, execvp
Int execlp (const char *file, const char *arg0, … *argn, (char *)0);
->file : 실행하는 파일의 파일이름
->arg0 ~ argn : 파일에 전달한 argv 정보 (명령 인수)
-> execlp()는 첫 번째 인자 file에서 지정한 파일 이름을 환경 변수 PATH에서 지정하고 있는 디렉터리 안에서 찾아 실행한다.

*exec() 이후의 문자
- l : argv 정보를 개개의 문자열 데이터를 가르키는 포인터 arg0, arg1… 으로 전달
- v : argv 정보를 개개의 문자열 데이터를 가리키는 포인터 배열의 선두주소 argv로 전달
- e : envp 정보를 전달 (environment 정보를 envp[] array로 전달)
- p : 파일 이름을 환경 변수 PATH로 지정한 디렉터리 안에서 찾아내어 실행

5) int system(const char *cmdstring);

Ex) system("date > file");
->return -1 : fork or waitpid error
Return 127 : exec error
Return : termination status of shell (waitpid로 지정된 값): fork, exec, waitpid가 성공하면

<프로세스간 통신>

IPC : 프로세스들끼리 데이터를 주고 받는 기능 수행
Ex) client/server 응용 프로그램, 여러 명이 공동 작업, 서로 다른 프로그램 모듈 간의 데이터 전달

1) PIPE : 주로 부모 자식 간에 쓴다
-메모리 버퍼 영역을 가리킴
-읽기용 입구 하나, 쓰기용 입구 하나를 가짐 (한 방향으로만 감)
-파일 디스크립터로 접근
-읽기용 입구와 쓰기용 입구에 각각 다른 파일 디스크립터가 할당됨
-부모 자식 간에는 fork를 통해 자연스럽게 사용가능
-pipe를 통하여 두 개의 프로세스 들이 통신을 할 경우, producer가 write한 순서대로 consumer가 read하게 된다 (FIFO) – 뒤 바뀌는 일은 없음

- 파이프를 생성한 프로세스가 종료하면 파이프이 사라진다 (파이프를 건네줄 수 없음)
(부모와 자식간에 상속은 가능하지만 서로 모르는 프로세스들끼리는 건네줄 수 없다는 말)
-**파이프의 기호는 ' | ' 이다.**
EX) ls -alg | more -> more이 ls -alg의 **표준출력을 입력으로 받아서 다시 출력**으로 내보낸다.

#include <unistd.h>
Int pipe(int fd[2])
->return 0 : 정상 종료, -1 : 에러
->pipe() system call은 인자로 넘겨 받은 fd[0], fd[1]에 파일 디스크립터를 저장해서 반환
->**fd[0]** : 읽기 모드에서 연 파일 디스크립터(**읽기용 입구**)
Fd[1] : 쓰기 모드에서 연 파일 디스크립터 (**쓰기용 입구**)

-pipe, fork 시스템 호출에 의해 부모, 자식 프로세스가 fd[0], fd[1]을 공유한다고 가정

FILE *popen(const char *cmdstring, const char *type); //
->return : file pointer if ok, Null on error
->cmdstring : 시스템에게 해당 명령어를 실행하라고, type : 어떤 용도로 쓸 것 이냐?
Type (돌려받는 파일 디스크립터 fp가 **parent의 입장**) // **cmdstring은 child가 실행한 것**
Cmdstring은 내부적으로 fork(), exec() 으로 만들어진 child가 실행하고 FILE *fp가 가리키는 곳에 갖다놓았다.

1) **'r'** : 명령어의 stdout 쪽과 연결이 된다. **즉, parent는 cmdstring의 결과를 읽는다**
2) **'w'** : 명령어의 stdin 쪽과 연결이 된다. **즉, parent는 쓰는데 child가 입력으로 받아서 실행**

Ex) fp = popen(cmdstring, "r") ; // parent가 명령어를 읽기용으로 사용하겠다.

Int pclose(FILE *fp); //

->I/O stream닫는다
Returns : termination status of cmdstring, or -1 on error

참고) fread : 파일로부터 지정한 개수만큼 자료 읽기
Size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
ex) int chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
파일 포인터로부터 읽고 buffer에 저장, chars_read : 실제로 읽은 바이트 수

-파일 내용을 읽어 들일 메모리 포인터, 데이터 하나의 크기, 읽어들일 데이터의 개수, FILE *stream 대상 파일 스트림
반환 : 읽어들인 데이터 개수, 오류 : -1

*단방향 파이프 :
-Producer 프로세스는 pipe에 대한 write용 입구만 사용
-Consumer 프로세스는 pipe에 대한 read용 입구만 사용

*양방향 파이프 : 파이프를 2개 만들어야 함

2) FIFO : named pipe
-파이프의 한계 (parent, children)를 극복하여 전혀 관련 없는 프로세스들 끼리도 통신을 할 수 ㅇ
-pipe에 이름을 붙여 파일처럼 사용하여 프로세스 간 통신

#include <sys/types.h>
#include <sys/stat.h>
Int mkfifo(const char *pathname, mode_t mode)
->**FIFO 파일 생성**
->pathname : 생성하고자 하는 FIFO 파일 명
->Mode : 권한 지정
->return 0 : 정상, -1 : 에러

FIFO
-**SIGPIPE** : FIFO에 write하러 갔는데 어떠한 프로세스도 읽기 위해 오픈하지 않으면 발생
(pipe가 끊겼을 때 발생하는 signal -> end쪽에 reading 하려고 연결되어 있는 프로세스가 없을 때)
-PIPE_BUF (kernel's pipe buffer size) : 한번 FIFO에 쓸 때 최대 크기 매크로
-Client / server application에서 사용할 수 있다.
Server가 well-known FIFO 파일을 만들고 Clients와 통신한다
Client는 PIPE_BUF 범위를 벗어 나지 않도록 write 해야 한다.
문제 : 서버가 하나의 "well-know" FIFO 파일로만 사용해서는 응답할 수 어렵다.
(특정한 클라이언트에게 응답을 해줘야 하는데, request한 client가 누구인지 모른다)
Client가 중간에 crashed 되면 커널에서 SIGPIPE 신호를 보낸다
해결 : 서버가 well-known FIFO 파일로부터 읽어서 각각의 클라이언트에 Client-specific FIFO 파일을 만들어서 개별적으로 Reply 한다.

Tee : 자기가 받은 입력을 두 군데로 나누어 줄 수 있다 (표준 출력, 내가 지정하는 특정 파일)

3) Socket : 부모 자식 이외에 로그인된 프로세스들이 다 사용할 수 있음
-UNIX 시스템에 속한 프로세스들 뿐만 아니라, 서로 다른 UNIX 시스템 들에 속한 프로세스들 간의 통신을 가능하게 함
-socket() 시스템 호출에 의해 만들어 짐
-pipe와 마찬가지로 파일 디스크립터로 접근
-전혀 관계없는 프로세스 간에 통신을 위해 socket을 식별할 목적으로 이름을 붙임
-internet 도메인에서는 socket 이름으로 포트번호를 사용한다.
*소켓 주소 : IP address + 포트 번호 (unique 함)

1. Virtual Circuit : circuit switching 방식 ->TCP
-통신 대상에 연결(논리적 접속)을 설정
-그 연결을 통하여 데이터를 송수신, 송수신이 끝나면 연결 해제
-양 쪽 소켓을 미리 연결을 해놓고 데이터를 주고 받는다. (TCP 프로토콜 기반)

Virtual circuit 기반 Stream socket

2.Datagram :
-통신 대상에 연결(논리적 접속)을 설정할 필요가 없음
-비 연결형 통신

Datagram 기반 Datagram socket ->UDP
-Datagram 기능을 이용하여 구현한 socket
-프로세스간에 작은 데이터를 계속해서 보내는 통신
-전송 데이터의 신뢰성이 보증되지 않음 (시간, 순서 등)

char *fgets(char *str, int num, FILE *stream)
str : 읽어들인 문자열을 저장, 최대 문자열 개수, stream에서 읽어옴

char *fputs(char *buffer, FILE *stream)
buffer에 있는 내용을 stream에 넣음

<소켓 프로그래밍>

소켓 : 같은 UNIX 시스템 혹은 서로 다른 UNIX 시스템 들에 속한 프로세스들 간에 통신 가능함

-socket() 시스템 호출에 의해 만들어 짐
-pipe와 마찬가지로 파일 디스크립터로 접근 가능
-bind() : 소켓에 특정 번호를 엮어 놓는 작업으로서 어느 프로세스든지 번호를 알면 통신이 가능하게 하도록 하기 위함

virtual circuit : 통신 대상에 연결(논리적 접속)을 설정 (**연결형 통신**)
-그 연결을 통하여 데이터를 송수신
-송수신이 끝나면 연결해제

stream socket : Virtual circuit 기능을 이용하여 구현한 socket
-프로세스 간에 대량의 데이터를 송수신하는 경우
-통신의 신뢰성 보증

////////////////////////////// Stream socket //////////////////////////////
client
-socket 작성 : socket() 시스템 호출
-socket()에 이름 붙임 : bind() //생략 가능
-server에 접속 요구 : connect()
-데이터 송수신 : read() / write() - > socket()이 파일 디스크립터로 접근이 가능하기 때문
-socket() 제거 : close()
server
-socket 작성 : socket() 시스템 호출
-socket에 이름 붙임 : bind()
-client의 접속 요구 받을 준비 : listen()

-접속 요구 허가 : accept()
-데이터 송수신 : read() / write()
-socket 제거 : close()

//////////////이렇게 사용하는 것이 Stream socket이다 ////////////////////

Datagram socket : Datagram 기능을 이용하여 구현한 socket (**비연결형 통신**)
-통신 대상에 연결(논리적 접속)을 설정할 필요 없음
-프로세스 간에 작은 데이터를 계속해서 보내는 통신
-매번 데이터를 보낼 때마다 상대방을 지정 (개개의 데이터를 그때 그때 상대방에게 보냄)
-전송 데이터의 신뢰성이 보증되지 않음

Client
-socket 작성 : socket() 시스템 호출
-socket에 이름 붙임 : bind() // 생략 가능
-데이터 송신 : sendto()
-데이터 수신 : recvfrom()
-socket 제거 : close()

server
-socket 작성 : socket() 시스템 호출
-socket에 이름 붙임 : bind()
-데이터 수신 : recvfrom()
-데이터 송신 : sendto()
-socket 제거 : close()

#include <sys/types.h>
#include <sys/socket.h>
1) int socket(int domain, int type, int protocol)
-소켓을 생성
-정상 종료시 생성된 소켓에 대한 파일 디스크립터 반환, 에러시 -1
-사용하는 도메인 종류 : 단일 유닉스 운영체제 ? 네트워크를 통해 나가는 인터넷 도메인?
(**PF_UNIX** : unix domain, **PF_INET** : 인터넷 도메인)

사용하는 소켓 타입 : Stream형 아니면 Datagram형?
(**SOCK_STREAM** : stream socket, **SOCK_DGRAM** : datagram socket, SOCK_RAW : 사용자가 직접 헤더

를 만들어 보냄 - IP주소를 부여X 특정 네트워크 내부에서만 통신하는 프로그램)

사용하는 프로토콜의 종류 :
->0 : 소켓 타입이 Stream이면 TCP로 , Datagram형이면 UDP로 시스템이 자동으로 해줌

2) **int bind(int sockfd, struct sockaddr *my_addr, int len)**
-socket() 시스템 호출에서 돌려 받은 socket 파일 디스크립터를 지정
-바인딩할 주소 구조체
-주소 구조체(*my_addr)의 길이
-struct sockaddr *my_addr 주소에는 다음을 지정
(UNIX DOMAIN일 경우 : Socket path (파일) 이름, Inet domain일 경우 : Inet 주소지(ip주소+포트번호))

-unix domain에서 socket의 주소 정보는 socketaddr_un 구조체에서 정의
-Inet domain에서 socket의 주소 정보는 socketaddr_in 구조체에서 정의
(#include <netinet/in.h>

```
struct sockaddr_in {
        sa_family_t    sin_family; //PF_INET 설정
        u_int16_t      sin_port; //포트 번호
        struct in_addr    sin_addr; //ip주소
        ....
}
```

#include <netinet/in.h>, <sys/socket.h> ..
3) int connect(int sockfd, const struct socketaddr *serv_addr, socklen_t , addr_len);
-정상 종료시 0, 에러 : -1
-int sockfd : socket 파일 디스크립터를 지정
-접속하고 싶은 서버의 소켓 주소 정보, 서버의 소켓 주소 정보 구조체의 길이

4) int listen(int sockfd, int backlog)
-정상 0 , 에러 : -1
-sockfd : socket 파일 디스크립터를 지정,
backlog : 연결 요청에 대한 대기열의 최대 길이 (대게 5)
->대기할 수 있는 연결 요청 수 (현재 서비스 중인 것은 제외)
->서버는 한번에 하나의 클라이언트 밖에 처리를 못함

5) int accept(int sockfd, struct sockaddr *client_addr, socklen_t *addrlen)

->**정상 종료시 소켓에 대한 새로운 파일 디스크립터를 반환**, 에러시 -1
-socket 파일 디스크립터, **접속한 client의 socket 주소 정보 구조체를 가리키는 포인터**, 길이

6) int send(int sockfd, const void *buf, size_t len, int flags)
-정상 종료시 실제 보낸 바이트 수를 반환, 에러시 -1
-accpet로 반환된 파일 디스크립터를 지정, 송신 버퍼의 시작 주소, 길이, 송신옵션

7) int recv(int sockfd, void *buf, size_t len, int flags)
-정상 종료시 실제 받은 바이트 수를 반환, 에러시 -1
-accpet로 반환된 파일 디스크립터를 지정, 수신 버퍼의 시작 주소, 수신 버퍼의 길이, 수신 옵션

8) int sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to)
-정상 종료 시 실제 보낸 바이트 수를 반환, 에러시 -1
-int sockfd : accpet로 반환된 파일 디스크립터를 지정
-const void *buf : 송신 버퍼의 시작 주소, 길이, 송신 옵션,
-받을 곳의 소켓 주소 정보 구조체를 가리키는 포인터, 길이
(메시지 전송방식) – 실습에서는 블록킹 모드만 사용한다.
블록킹 모드 : 메시지가 전송 버퍼보다 크면 전송불허
비 블록킹 모드 : 메시지가 버퍼에 맞지 않으면 'eagain' 메시지 반환
-**Flag** : 메시지 전송에 대한 타입 정의
MSG_EOR :
MSG_OOB : out out band data(평소의 데이터보다 시급한 데이터)를 먼저 보내라 – ctrl+c 누른데이터
MSG_NOSIGNAL : 문제가 생기더라도 시스템에서 신호를 받는 것을 거부하겠다.
MSG_WAITALL : 다 기다린다 (서버입장)

9) int recvfrom (int sockfd, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)
-정상 종료 시 실제 받은 바이트 수를 반환, 에러시 -1
-int sockfd : accpet로 반환된 파일 디스크립터를 지정
-void *buf : 수신 버퍼의 시작 주소, 길이, 수신 옵션
-struct sockaddr *from : 보낸 곳의 소켓 주소 정보 구조체를 가리키는 포인터, 길이
-블록킹 모드에서는 데이터가 올때까지 기다리고, 비 블록킹 모드에서는 데이터가 오지 않으면 -1 반환한다.

<UDP 프로그래밍>
-빠르나 신뢰성은 의문, Connectionless, 패킷 손실 가능, SNMP 에서 사용, 인터넷 방송
SNMP를 지원하는 대표적인 장치에는 라우터, 스위치, 서버, 워크스테이션, 프린터, 모뎀 랙 등이 포함

struct sockaddr_in server_addr;
-소켓 관련 인터넷 주소를 구성하는 구조체
struct sockaddr_in {
    sa_family_t sin_family; // 주소 패밀리
    unsigned short int sin_port; // 포트번호
    struct in_addr sin_addr; // IP주소
}
inet_addr('127.0.0.1') -> 문자 값을 2진 수로 바꾸어 주는 함수 + IP
즉, 문자 값을 네트워크 바이트오더의 2진 수, 빅엔디안 이진 수로 바꾸어 줌
-#include <sys/types.h> , <netinet/in.h>, <arpha/inet.h>

htons : 네트워크 바이트 오더가 적용된 이진 16비트 값 (host to network unsigned short int)
-이진 16비트 값을 네트워크 바이트 오더로 전환 후 리턴
(호스트 컴퓨터는 리틀 앤디언, 네트워크에 실을 적에 빅 엔디언을 사용해서 이 둘을 맞춰주기 위해서 사용)

struct sockaddr_in client_addr, server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
➔ **해당 서버로 들어오는 어떤 ip 주소든 다 받아 주겠다.**
server_addr.sin_port = htons(3317);

<다중 접속처리 서버 구현>
-여러 개의 클라이언트가 접속할 점을 고려해야 함.
pid_t fork()를 통해서 대처한다.
➔ fork()해서 자식프로세스를 만들어서 정상 종료해도 child는 좀비 프로세스로 남아 있음.
➔ 커널이 SIGCHLD 신호를 보내는데, SIGCHILD 신호를 처리하는 핸들링 루틴을 서버 프로그램에 장착을 해야 한다(WAIT 시스템 콜은 쓰지 못한다)
➔ SIGCHLD 신호는 그것을 핸들링 하는 루틴이 없으면 무시된다.

#include <signal.h>
void (*signal) (int signum, void (*handler) (int)) (int);
-시그널 핸들러 포인터 signal (시그널 번호, 시그널 핸들러)

서버 측면에서 아래와 같이 사용한다.

```
void handler(int sig) {
        pid_t pid;
        int status;
        pid = wait(&status);
        printf("%d번 자식 프로세스가 종료되었습니다₩n", pid);
}
main() {
        signal(SIGCHLD, (void *)handler);
}
```