유닉스

**유닉스 기본 설명**

UNIX는 미국 벨(Bell) 연구소에서 개발된 운영체제로, 프로그램 대부분이 C언어로 수정되면서 **이식성이 높아지고 동시 다중 사용자 및 다중작업의 실행을 지원**할 수 있는 **대화형 소프트웨어**이다.

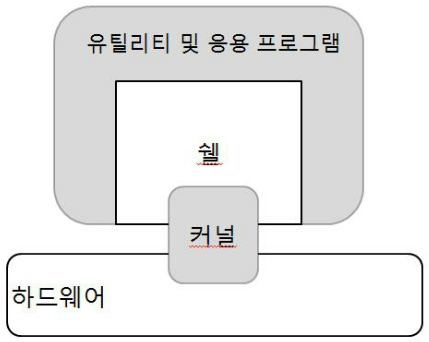
주로 **서버용** 컴퓨터에서 사용되는 운영체제

* 포털이나 대기업의 서버에 사용되며 보안성이 매우 뛰어남

**특징**

1. 시분할 시스템(Time Sharing System)을 위해 설계된 **대화식 운영체제** 🡪 shell 이용
2. 대부분 C언어로 작성되어 있어 **이식성이 높으며 장치, 프로세스 간의 호환성이 높다**.
3. **다중 사용자(Multi-user), 다중 작업(Multi-Tasking)을 지원**
4. 많은 네트워킹 기능을 제공하므로 통신망 관리용 운영체제로 적합
5. **트리구조의 파일 시스템**을 가진다

**UNIX 시스템의 구성 – 커널, 쉘, 유틸리티 및 응용 프로그램**

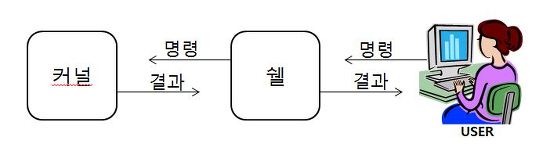


1. **커널(Kernel)**: 유닉스 시스템이 부팅될 때 가장 먼저 읽히는 운영체제 핵심 부분

하드웨어와 관련된 작업을 수행하는 것으로 **사용자들은 커널에 접근 불가능**

+) **시스템 호출(System call)**: 커널과 사용자 사이의 인터페이스 역할을 한다. (사용자가 커널의 기능을 사용할 수 있도록 하는 역할) 🡪 보통 시스템 콜을 직접 사용하기 보다 해당 시스템 콜을 사용해서 만든 API를 사용

1. **쉘(Shell)**: 사용자가 운영체제 기능과 서비스를 조작할 수 있도록 인터페이스를 제공하는 프로그램. 유닉스 터미널



1. **유틸리티 및 응용 프로그램**: 유닉스는 많은 프로그래밍 언어를 지원하고, 파일작성, 파일출력, 파일 내용을 선택적으로 변경시키는 기능 등과 같은 여러 가지 유틸리티와 다른 컴퓨터와 통신을 가능하게 해주는 여러 가지 응용 프로그램으로 구성

**Unix File System**

**File**

유닉스는 일반적인 파일들 뿐 아니라 **터미널, 프린터, 디스크 등 모든 주변 장치들도 하나의 파일로 취급** 🡺 장점: 프로그램을 작성하는 것 없이 옮기기, 쓰기, 읽기를 간단하게 할 수 있다.

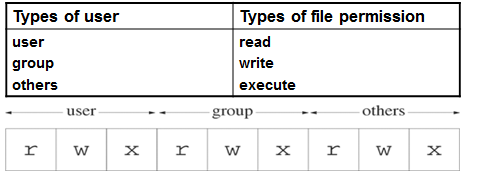
1. **파일 종류 – 정규파일, 디렉토리 파일, 특수 파일, FIFO, Socket**
2. 일반파일, 정규파일: 각종 텍스트파일, 시스템 파일, 이미지 파일 등 유닉스에서 사용되는 대부분의 파일
3. 디렉토리 파일: 다른 파일들의 목록과 그 파일들의 정보(주소)를 가리키는 포인터들을 가지는 파일 (우리가 아는 폴더)
4. Block Special file: 버퍼를 이용해 블록단위로 데이터를 쓰고 읽는다. (디스크 파일)
5. Character Special file: 버퍼를 쓰지 않고 데이터를 쓰거나 읽는 것이 즉시 일어난다. (키보드, 마우스 입출력)
6. FIFO (named PIPE): 프로세스간 통신에 쓰이는 이름이 있는 파이프 파일
7. Socket(소켓): 네트워크의 입출력을 담당하는 API로 두 호스트 컴퓨터 사이의 정보를 전달
8. Symbolic link: 다른 파일을 가리키는 타입의 파일로 윈도우의 바로가기 파일과 비슷하다.
9. **파일의 속성** (**‘ls -l’ 명령어**를 통해 해당 디렉토리와 속한 파일 정보 알 수 있다.)



1. 파일 종류



1. Permission: 여러 유형의 유저들(user, group, others)이 파일에 접근할 수 있는 방법(read, write, execute)을 결정 🡪 chmod()를 통해 소유자가 변경 가능



User: 파일을 만든 소유주

Group: 파일을 만든 소유주가 속한 그룹의 사용자

Other: 기타 사용자

1. 링크 카운터(link counter): 동일한 inode를 참조하는 파일 수
2. Owner: 해당 디렉토리 및 파일의 소유주 이름 🡪 chown명령 사용해 변경 가능
3. Group: 파일이 속한 그룹의 이름 🡪 chgrp명령 이용해 변경 가능

**Directory**

유닉스 파일 시스템은 계층 구조로 시작 디렉토리를 root(/.)라고 한다.

1. **디렉토리의 permission 의미**
   1. Read: 하위 디렉토리와 파일들의 이름을 나열
   2. Write: 파일 생성, 제거
   3. Execute: 특정 filename을 찾기 위한 탐색권한

Ex) /usr/include/stdio.h 파일을 열기 위해 실행 권한 필요한 디렉토리: /, /usr, /usr/include

1. **경로**
2. 절대 경로: 루트 디렉토리에서 시작해 파일 위치를 표현
3. 상대 경로: working directory기준으로 파일 위치를 표현
4. Working directory: 현재 수행중인 디렉토리
5. Home directory: login할 때 첫번째 directory로 초기 working directory는 home디렉토리
6. 관련 System call

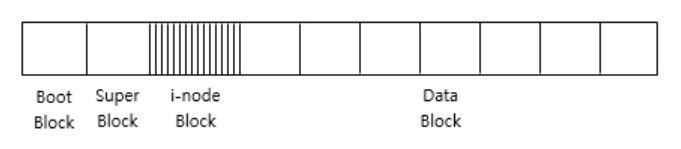
Mkdir(폴더 생성), **chdir(현재 작업 디렉토리 변경)**, opendir, closedir, readdir, rewinddir, getcwd, ftw

**File System**

디렉토리와 그 안에 저장된 파일을 찾기 쉽도록 유지/관리하는 시스템

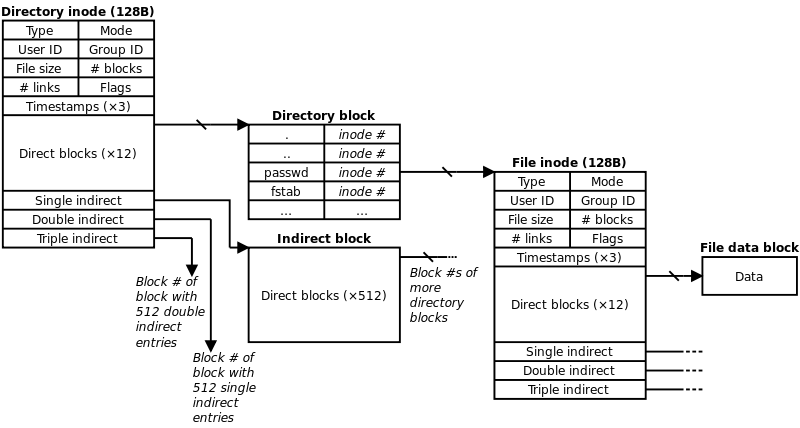
1. 파일 시스템의 구조 – 시스템의 각 디스크 블록은 4개 영역을 가진다.

**Boot Block, Super Block, i-node Block, Data Block**



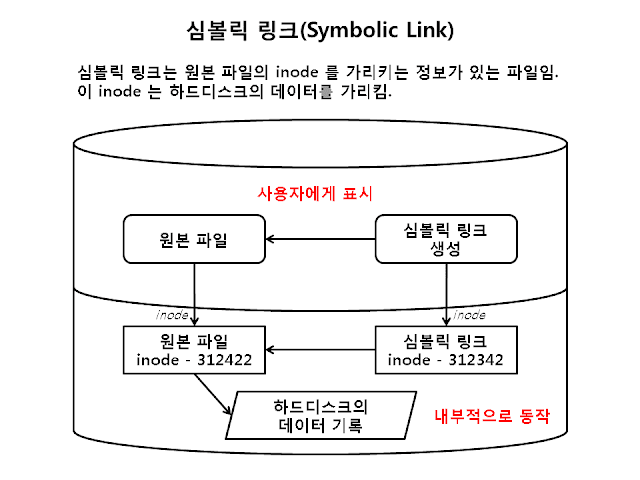
1. Boot block: 파일 시스템으로부터 UNIX커널을 적재시키기위한 프로그램이 저장됨
2. Super block: 전체 파일 시스템에 관한 정보(파일 시스템에 있는 총 블록의 개수, 블록 크기 등)
3. i-node: 파일이나 디렉터리에 대한 모든 정보를 가지고 있는 구조체
4. data block: 실제 데이터가 파일의 형태로 저장되는 공간 (자유 블록을 포함하며 연결리스트로 유지)
5. **i-node**

* 약 120byte 고정 크기의 구조체로 외부적으로는 번호로 표현된다.
* 파일 시스템 내에서 파일이나 디렉토리는 고유한 i-node를 가지고 있음 i-node번호를 통해 식별 가능하다 (key역할)
  + 사용자가 파일 또는 파일과 관련된 정보에 접근할 때 파일이름을 사용하지만, 내부적으로는 파일 이름은 먼저 디렉토리 테이블에 저장된 i-node번호로 mapping되며 그 후 i-node번호를 통해 해당 i-node에 액세스 됨
* 포함 정보 - 소유권, 권한, 실제 데이터가 있는 물리적 주소, 파일의 형태, 크기, 링크 수, timestamp 등 (파일 이름 제외)
* 파일 이름은 inode의 번호와 함께 directory block안에 저장된다.
* I-node번호는 0,1을 빼고 사용. 2번은 root directory에 예약되어 있다.



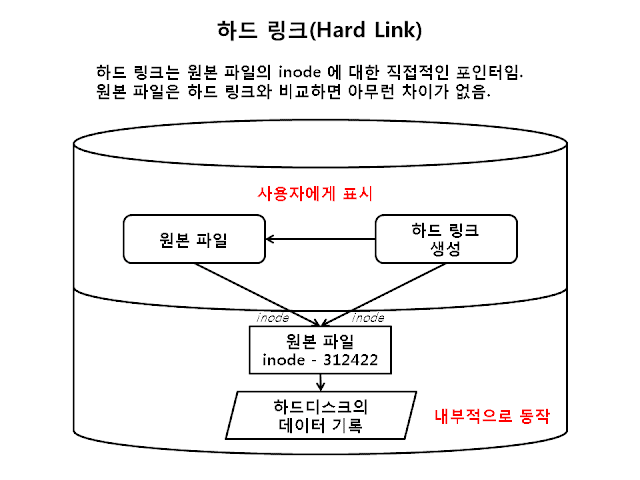
* 링크 – 심볼릭 링크, 하드 링크

1. Symbolic link – symlink() system call로 생성



* + 윈도우 시스템에서 제공하는 **바로가기 기능**과 매우 유사
  + **파일 내용에** 원본 파일에 대한 정보가 아닌 **원본 파일 위치에 대한 포인터가 포함**되므로 **새로운 inode를 가진 링크파일이 생성됨**
  + Read()사용할 경우 원본 파일의 정보가 읽힌다.
  + Readlink(): 심볼릭 링크 파일에 저장된 정보(경로)를 읽을 때 사용

1. Hard link – link() system call로 생성



* + 원본 파일의 inode의 위치를 가리키므로 **원본 파일과 inode번호가 동일**
  + 심볼릭 링크와 가장 큰 차이는 새로운 inode가 생성되지 않는다는 것! 🡪 이에 따라 원본 파일이 삭제될 경우 접근 가능 여부가 달라진다.

|  |  |  |
| --- | --- | --- |
| 구분 | **심볼릭 링크** | **하드 링크** |
| **생성 명령어** | ln –s [원본 파일명] [링크 파일명] | ln [원본 파일명] [링크 파일명] |
| **생성 종류** | 파일과 디렉토리 모두 생성 | 파일만 생성 |
| **링크 기능** | 파일 또는 디렉토리 이름에 대한 링크를 가리킴 | 원본 파일에 대한 참조 또는 포인터 |
| **원본 파일 삭제할 경우** | 액세스 불가능 | 액세스 가능 |
| **inode 번호** | 다른 inode 번호 | 동일한 inode 번호 |
| **다른 파티션 링크 여부** | 다른 파티션에 링크 가능  (다른 파일 시스템에도 cross가능하다는 뜻 – 가리키는 file의 절대 경로 있으면 되므로) | 다른 파티션에 링크 불가능  (같은 파일 시스템에서만 있다. – 다른 파일 시스템에서 inode 같은 건 다른 file, 같은 파일 시스템에서 inode같은 것은 같은 file) |

1. **System call**

**기본적인 파일접근 시스템 콜**

1. Open: 읽거나 쓰기위해 파일을 열거나 새로 생성
2. Create: 파일을 생성 (open과 달리 항상 파일이 존재하면 내용 삭제, 항상 쓰기모드)

* open함수로 구현 가능 open(path, O\_WRONLY | O\_CREATE | O\_TRUNC, mode);

1. Close: 파일을 닫음
2. Read: 파일에서 정보 추출
3. Write: 기록
4. Lseek: 파일안의 지정된 바이트로 이동
5. Unlink: 파일제거(remove도 쓰인다)
6. Fcntl: 이미 열려 있는 file의 속성을 바꿈

**권한 관리 시스템 콜**

1. Chmod: 파일의 permission을 수정, 파일의 owner나 슈퍼유저만 가능
2. Umask: 파일 생성 마스크 값을 변경

**파일 생성 마스크?** 파일이 생성될 때 자동으로 특정 권한비트들을 0으로 바꿈. Permission이 실수로 켜지는 것을 방지해 파일 보호 (default로 작동됨)

**소유주 변경 시스템 콜**

1. Chown: 파일의 uid(소유주)와 gid(그룹)를 변경. 슈퍼사용자나 owner만 가능

Chown을 통해 파일 소유권이 바뀌면 set-user-id와 set-group-id권한이 꺼진다. (특수권한, 뒷부분에 설명)

* 권한이 꺼진다는 말은 파일 소유주의 id를 effective uid로 부여하지 않는다는 의미
* 즉, 파일에 대한 이전 소유주는 소유주 바뀐 해당 파일에 접근할 수 없다.

\*\* 참고 \*\*

ssized\_t ? UNIX버전마다 각 시스템콜에서 사용하는 자료형이 다를 수 있다. 버전 사이에 호환성을 유지하기 위해 사용되는 자료형.

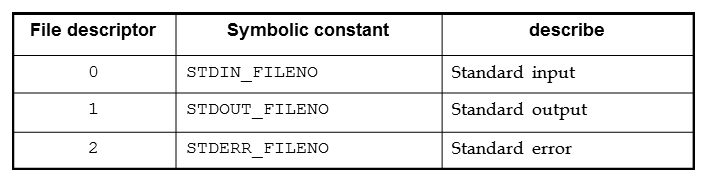
\_t로 끝나는 자료형들을 primitive system dataype이라 한다.

1. **File descriptor (파일 기술자)**

프로세스에서 특정파일에 접근 시 사용하는 **file을 가리키는 번호**로 **모든 오픈파일은 file descriptor를 갖는다**. (시스템에 의해 결정되는 음이 아닌 정수 값)

* Open(), create() 함수 수행 시 커널에서 파일 디스크립터 값을 반환
* Read(), write() 함수 인수로서 전달된다.

프로그램 실행될 대 기본적으로 할당되는 디스크립터 (표준 입출력)

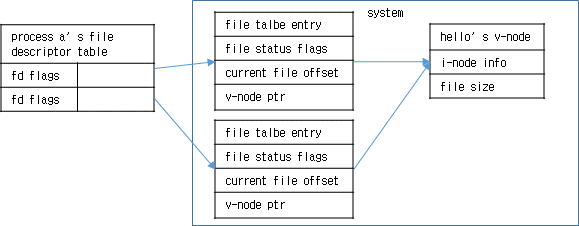


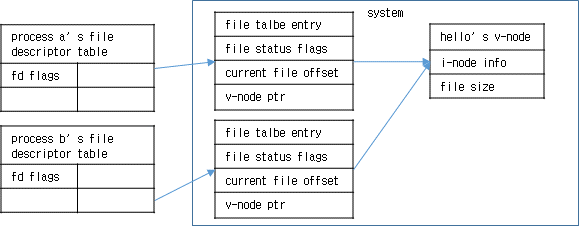
1. **프로세스의 파일 공유**

유닉스에서 프로세스간 파일을 공유하기 위해 3가지 테이블이 필요

**File descriptor table, file table, v-node table**

1. File descriptor table(=process table entry): 프로세스마다 한 개 생성되며 다음과 같이 구성된다. (파일 디스크립터가 index역할을 한다)
   * fd flag (open에서 사용한 상태 flag)
   * file pointer (파일 테이블 요소의 위치 정보)
2. File table: 시스템의 모든 열려진 파일들을 관리하는 테이블로 각 entry의 구성은 다음과 같다.
   * File state flags
   * Current file offset
   * v-node pointer (v-node 테이블의 위치 정보)
   * 한 프로세스에서 A파일이 두 번 열린 경우: 서로 다른 fd부여 및 다른 entry를 구성 🡪 v-node pointer는 동일
   * 서로 다른 프로세스에서 A파일이 열린 경우: 각 프로세스의 fd부여 및 다른 entry 구성 🡪 v-node pointer는 동일
3. V-node table: 각 entry는 open된 파일의 i-node정보와 파일의 현재 크기를 포함한다. (같은 파일이 여러 번 open된 경우 하나의 v-node entry존재. 때문에 **같은 파일을 사용하는 프로세스들은 v-node를 공유)**

 🡨한 개의 process

 🡨 여러 process

1. **File in context - 다중 사용자 환경에서 파일**

프로세스에 부여되는 번호들: PID, RUID, EUID, RGID, EGID

1. PID: 프로세스 식별자
2. Owner: 모든 파일은 시스템의 한 유저에게 소유되고 user-id(UID)로 구분
3. Group: 모든 유저는 하나 이상의 그룹에 속한다. Group-id(GID)

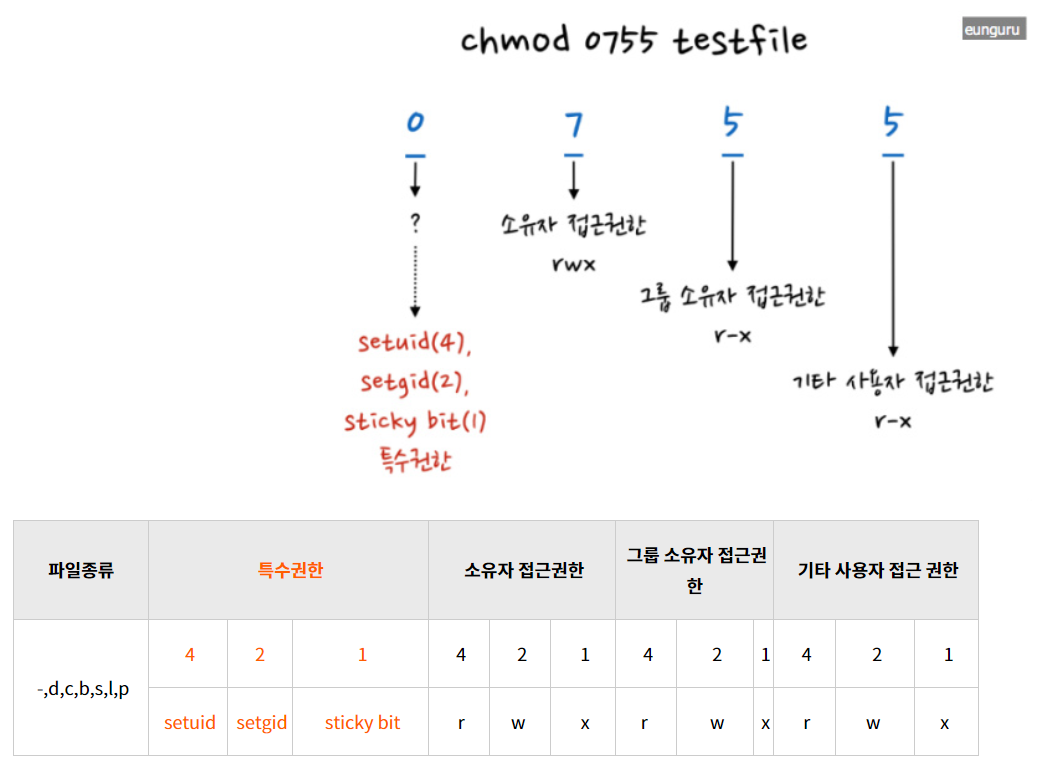
(파일이나 디렉토리는 만든 사용자의 소유가 됨)

1. RUID (real user-id): 프로세스를 시작한 유저의 UID (계정 관리에 사용)
2. EUID (effect user-id): 접근 권한 결정에서 사용(파일 접근 가능 여부를 결정한다.)

(보통 RUID == EUID이지만, special case(RUID != EUID)가 존재.)

open()함수 파라미터로 파일의 접근 모드가 전달되고 커널이 파일 접근 테스트 수행

1. EUID == 0(superuser) 🡪 접근 허용
2. user접근 허용 bit가 set, EUID == owner 🡪 접근 허용
3. group접근 허용 bit가 set, EGID == 파일의 gid 🡪 접근 허용
4. Other 접근 허용 비트가 셋 🡪 접근 허용
5. **Extra permission for executable file**



특수권한 종류: **set user-id(SUID),** set group-id(SGID), sticky bit

1. **SUID**: 어떤 사용자가 프로그램을 실행해도 파일 소유자의 권한으로 실행된다.

즉, **EUID = 파일 소유자 UID가 되어 파일 접근이 허용**된다. 🡨 special case(위 내용)

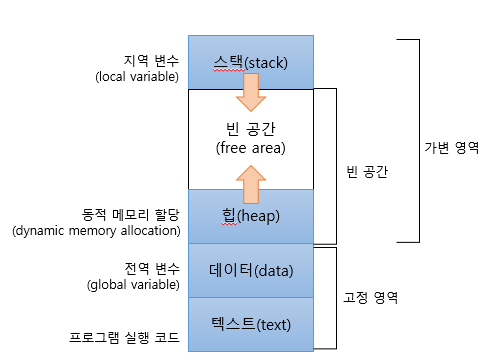
1. SGID: 누가 파일을 생성해도 파일의 소유그룹은 원래 파일의 소유 그룹으로 적용됨 (여러 사용자가 공유하는 디렉토리에 설정하면 편리)
2. Sticky bit: 디렉토리에 대해서만 설정. Sticky bit가 set된 디렉터리는 자신에게 쓰기 권한이 있어도 자신이 소유한 파일 외에는 삭제 불가능 (생성만 가능하다)

(write권한을 가지면서 파일의 소유주, 디렉토리 소유주, 슈퍼유저 경우만 삭제 가능)

1. SUID는 owner의 실행권한에 **s**로 표시, SGID는 group의 실행권한에 **s**표시, Sticky bit는 other의 실행권한에 **t**로 표시 (**rwsrwsrwt**)

**Process**

1. 프로세스 구조

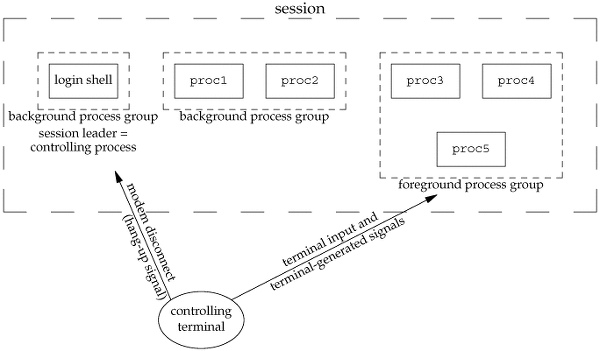


1. 텍스트 영역: 실행 코드를 저장한다. 텍스트 영역은 프로세스 실행 중 크기가 변하지 않는 고정 영역에 속한다.
2. 데이터 영역: 프로그램에서 정의한 전역 변수를 저장한다. 전역 변수는 프로그램을 작성할 때 크기가 고정되므로 고정영역에 할당된다. (컴파일 때 배열 크기가 정해진다.)
3. 힙(heap): 프로그램 실행 중에 동적으로 메모리를 요청하는 경우에 할당되는 영역으로, 빈 영역→할당→할당 해제처럼 상태가 변하는 가변 영역이다. (runtime시 배열 크기 유연)
4. 스택(stack): 프로그램에서 정의한 지역 변수를 저장하는 메모리 영역으로, 지역 변수를 정의한 부분에서 할당해 사용한다. 서브 함수를 부를 때 stack에 쌓인다.
5. 빈 공간: 스택이나 힙과 같이 가변적인 메모리 할당을 위해 유지하고 있는 빈 메모리 영역이다. 프로세스에 할당된 빈 메모리 영역이 모두 소진되면 메모리 부족으로 프로그램의 실행이 중단될 수도 있다.
6. PID(process ID): 시스템에서 여러 프로세스를 식별하기 위해 사용

-> system call: getpid(), getppid()

1. PGID(process group ID): 관련 프로세스들이 모여 프로세스 그룹을 구성, 이를 구성하는 프로세스 중 하나가 그룹의 리더가 되고 리더의 PID가 PGID가 된다.
2. Session(세션): POSIX표준에서 제안한 개념으로, 사용자가 로그인해 작업하는 터미널(terminal)단위로 프로세스 그룹을 묶은 것 (즉, 프로세스들을 묶으면 프로세스그룹, 프로세스그룹들을 묶으면 세션)

프로세스가 새로운 세션을 생성하면 해당 프로세스가 세션의 리더가 된다



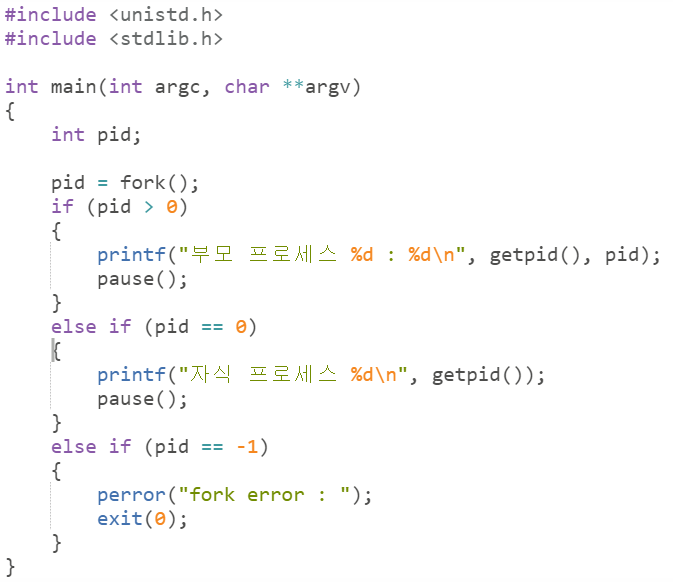
Controlling terminal: 표준 입출력에 연결된 terminal

1. 프로세스 종류

시스템 프로세스: UNIX운영에 필요한 기능 수행

사용자 프로세스: 사용자들이 실행시킨 프로세스

1. 데몬(daemon): 특정 서비스 제공하기 위해 존재, 리눅스 커널에 의해 실행되고 평소에는 대기상태로 요청을 기다림
2. 부모: 자식 프로세스를 생성하는 프로세스, 모든 프로세스는 부모로부터 기본 환경을 물려받음 (프로세스가 프로세스를 만들기 때문에 tree구조를 가짐, shell도 프로세스)
3. 자식: 부모에 의해 생성, 부모에게 결과를 return하고 종료
4. 고아: 자식 프로세스가 실행 중인데 부모가 먼저 종료된 경우, 1번(init)프로세스가 새로운 부모가 되어 종료를 지원
5. 좀비: 이미 사용중인 자원을 반납했지만 부모 프로세스로부터 작업종료에 대한 승인을 받지 못한 상태, 좀비 프로세스들은 실행되고 있지 않지만 프로세스 테이블 목록을 차지하고 있어 자원의 낭비요소가 된다. (목록에 defunct 프로세스라고 나옴) Kill명령으로 제거 불가능하며 부모 프로세스에게 SIGCHLD를 통해 자식을 없애거나 부모 자체를 종료해야 한다.
6. 프로세스 생성 (fork, exec)
   1. Fork: 부모가 fork()함수 호출하면 새로운 메모리공간을 할당받은 자식프로세스가 생성된다.
   * 부모의 stack, heap, data 공간을 copy (다른 메모리에 저장)
   * 부모와 자식은 코드만 공유
   * Fork()함수의 반환 값은 자식과 부모에서 다르다 (자식: 0반환, 부모: child의 pid반환)
   * Fork전 open file 있을 경우: 부모와 자식은 같은 file table을 공유하고 이는 exec호출되어도 공유됨 (공유하기 싫은 경우 부모가 file lock 걸면 됨 -> close-on-exec flag를 on으로 설정)



* 1. Exec: 원래 프로세스를 새로운 프로세스로 대체한다.
  + 메모리에 새로운 프로세스의 코드를 덮어 씌움
  + 코드의 첫 줄부터 수행된다.

1. 프로세스 종료

프로세스가 종료되는 두가지 경우 – 정상종료, 비정상 종료

* 1. 정상 종료: main()함수가 끝나면 자동으로 **exit()함수**가 실행되고 종료된다.
  2. 비정상 종료: exit()실행 없이 종료된 것
     1. Abort system call (잘 안 씀)
     2. **Signal 받을 경우 (kill)**

1. Synchronizing process – wait함수
2. Wait(): 부모 프로세스가 자식 프로세스의 종료 상태를 얻기 위해 사용. 🡪 좀비 프로세스, 고아 프로세스가 발생하지 않는다.
3. Waitpid(pid, &status, options): 여러 자식들 중 특정 pid에 대해서만 기다린다.
   * Options: [WCONTINUED, WNOHANG, WUNTRACED]
   * WNOHANG옵션을 쓰기 위해 이 시스템 콜을 사용 🡪 부모 놀지말고 계속 실행해라. 자식이 종료 안되더라도 기다리지 않고 진행
4. 포그라운드 & 백그라운드 프로세스
   1. 포그라운드 처리: 사용자가 입력한 명령이 실행되어 결과가 처리될 때까지 프롬프트가 출력되지 않아서 다른 명령을 입력할 수 없는 처리방법
   2. 백그라운드 처리: 명령의 입력이 끝나면 결과에 상관없이 바로 다음 프롬프트가 출력되어 다른 명령을 입력할 수 있는 처리 방법

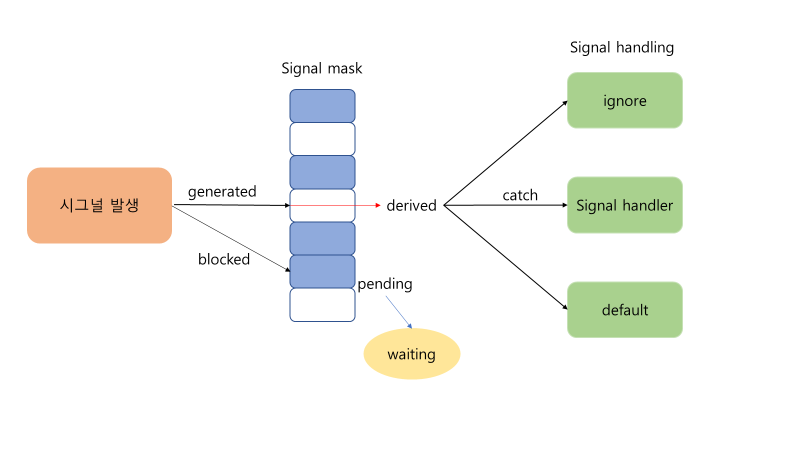
(명령의 처리시간이 오래 걸리는 경우 유용, 데몬 프로세스처럼 서비스 요청이 올 때까지 기다려야 하는 프로세스들 유용)

**Signal (시그널)**

특정 이벤트가 발생했을 때 프로세스에게 전달하는 **소프트웨어 인터럽트**이다.

모든 시그널의 이름은 SIG-로 시작하고 <signal.h>헤더에 양수인 상수로 정의된다.

**시그널 동작 과정**



1. 시그널 발생 경우
   1. 외부에 의해 (^C와 같은 키보드 입력)
   2. 에러에 의해 (나누기 0 등 hardware exception)
   3. 이벤트에 의해 (alarm()함수, 프로세스 종료 등)
   4. 인위적 발생 (kill 명령)
2. 시그널 처리 – 프로세스가 시그널 받으면 3가지 액션 중 한가지 수행
   1. Ignore action
   2. User-defined action – 커널에서 시그널 발생 시 호출할 함수를 정의 (signal handler)
   3. Default action – 모든 시그널은 default를 가지면 기본 처리는 종료 or 코어 덤프 생성(SIGSTOP, SIGTSTP, SIGCONT제외한 나머지)

(코어 덤프 – 비정상적인 종료 시 특정 시점에 작업 중이던 메모리 상태를 기록)

1. Signal mask에 의한 block
   1. Pending - 시그널 마스크에 의해 signal이 block되어 아직 전달되지 않은 signal
   2. Waiting – block된 signal은 process가 그 signal을 unblock할때까지 혹은 해당 signal에 대한 처리를 ignore로 변경할 때까지 대기한다.

**주요 시그널 종류**

|  |  |  |
| --- | --- | --- |
| 시그널 | 기능 | 디폴트 액션 |
| SIGABRT | Abort함수(비정상 종료) 호출함으로써 생성됨  호출 프로세스 자기자신에게 시그널 보냄 | 종료 및 코어덤프 |
| SIGALRM | alarm함수에 의해 발생.  Alarm(t)에 의해 지정된 시간 만료되면 호출 프로세스에게 시그널 보냄 | 프로세스 종료 |
| SIGCHLD | 자식 프로세스가 죽었을 때, 부모프로세스에게 이를 알리는 시그널 | 무시 |
| SIGCONT | 정지했던 프로세스가 재개되었을 때, 정지했던 프로세스에게 보내는 시그널 | 프로세스 재개 or 무시 |
| SIGFPE | 0으로 나누었을 때와 같은 산술적 예외에 관한 시그널 | 종료 및 코어덤프 생성 |
| SIGILL | 불법 하드웨어 명령 실행할 때 보내짐 | 코어 덤프 및 종료 |
| SIGINT | 인터럽트 키(^C)가 눌러졌을 때, 포그라운드 프로세스 그룹에 있는 모든 프로세스들에게 시그널 전송 | 종료 |
| SIGKILL | 프로세스를 죽임 | 종료 (디폴트 변경 불가) |
| SIGPIPE | 파이프 I/O시 발생하는 시그널 | 종료 |
| SIGSEGV | 불법적인 메모리 접근 시 발생 | 종료 및 코어 덤프 |
| SIGSTOP | 프로세스를 멈추기 위한 시그널 | 프로세스 멈춤 |
| SIGTERM | 프로세스 종료 | 종료 |
| SIGTSTP | ^C눌렀을 때 프로세스를 멈추는 시그널 | 프로세스 멈춤 |
| SIGURS1&SIGURS2 | 사용자가 정의한 시그널1,2 | 종료 |

* **SIGKILL vs SIGTERM**: 프로세스 종료. SIGKILL은 프로세스가 catch하거나 무시할 수 없다.
* **SIGSTOP vs SIGTSTP**: 프로세스 중단. SIGSTOP은 프로세스가 catch하거나 무시할 수 없다.

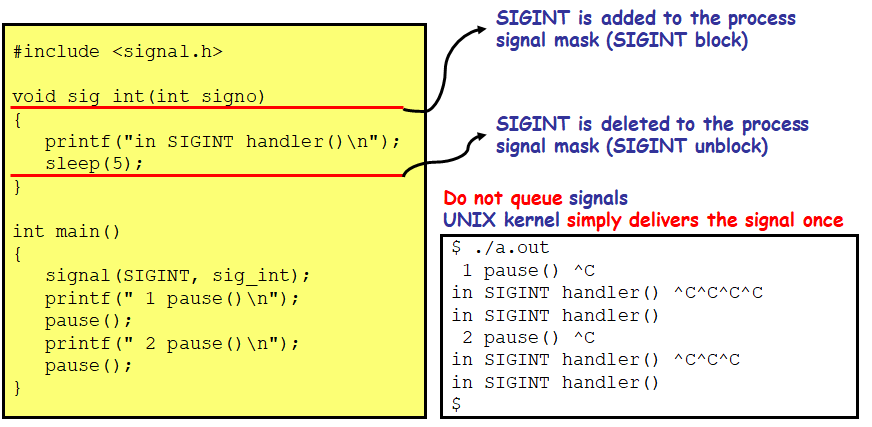
**Process Signal Mask**

시그널이 생성될 때 취해지는 조치는 해당 시그널의 signal handler와 signal mask에 따라 다르다.

Signal mask는 프로세스 속성으로서 signal blocking list이다. 즉, 목록에 있는 시그널은 block됨

1. 같은 시그널이 여러 번 들어올 경우 한번만 수행될 수 있도록 한다.

* Signal catching함수가 호출되기 전에 프로세스의 시그널 마스크를 추가하여 같은 시그널 수행 중일 경우 block처리 된다.
* 함수의 수행 완료되면 pending을 풀고 block됐던 시그널을 처리
* 같은 동작이 여러 번 들어와도 한 개만 저장해 뒀다가 시그널 처리



1. 프로그램은 sigprocmask() system call을 사용해 signal mask를 변경하여 신호를 차단한다.
2. 프로세스는 fork()와 exec()후에 시그널 마스크를 상속
3. 단, signal handler는 fork()까지만 유효
4. System call: signal(int signo, void (\*func) (int)) (해당 시그널 no에 대해 func수행)

* Signo: signal 이름
* \*func: SIG\_ING, SIG\_DFL, signal handler(signal caching function)

(더 최신 버전 sigaction(signo, \*act, \*oact) : signo(시그널 넘버), act(액션 수정), oact(이전 액션))

1. System call은 두가지 카테고리로 나뉨 (나뉜다 정도..)
   1. “slow” system call – 중요하지 않은 시스템 콜, 수행 중 시그널이 보내지면 시스템 콜 중단된다. (수행 실패로 -1반환)

* Pipes, terminal device, network device
* Pause(), wait()
* 몇몇 IPC functions
* Disk I/O와 관련되지 않는 것
* Sigaction system call은 시스템 콜이 중단되었을 경우 자동으로 재시작 가능하게 함
  1. All the other – read, write같은 시스템콜 수행 중 시그널 보내지면 시스템콜 완료 후 시그널이 수행됨

**IPC(Inter Process Communication) – 프로세스간 통신**

프로세스는 완전히 독립된 실행 객체로 존재한다.

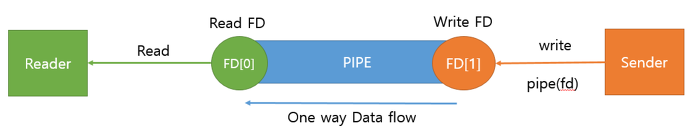
장점: 프로세스간 영향을 받지 않는다.

문제: 별도의 설비 없이 서로간 통신이 어려움 🡪 해결: 커널 영역에서 IPC제공하여 프로세스간 통신이 가능

**IPC종류: PIPE, FIFO, message queue, shared memory, socket, semaphore**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IPC 종류** | **PIPE** | **Named PIPE** | **Mesage Queue** | **Shared Memory** | **Socket** |
| **사용 시기** | 부모 자식 간  단 방향 통신 시 | 다른 프로세스와  단 방향 통신 시 | 다른 프로세스와 단 방향 통신 시 | 다른 프로세스와 양 방향 통신 시 | 다른 시스템간  양 방향 통신 시 |
| **공유 매개체** | 파일 | 파일 | 메모리 | 메모리 | 소켓 |
| **통신 단위** | Stream | Stream | 구조체 | 구조체 | Stream |
| **통신 방향** | 단 방향 | 당 방향 | 단 방향 | 양 방향 | 양 방향 |
| **통신 가능 범위** | 동일 시스템 | 동일 시스템 | 동일 시스템 | 동일 시스템 | 동일 + 외부  시스템 |

**PIPE**



두개의 프로세스를 연결하고 한쪽 방향으로만 통신이 가능하기 때문에 Half-Duplex(단방향)통신이라고도 부른다.



인자 – filedes[0]: read, filedes[1]: write

* Blocking & Non-blocking

1. 시스템 콜은 호출자를 영원히 block할 수 있다 – read할 내용 없을 때, write할 자리 없을 때, FIFO에 특정 조건이 발생할 때까지 block
2. 지정된 파일 디스크립터에 대해 nonblocking I/O를 지정하는 방법
   1. open호출 시 O\_NONBLOCK지정
   2. 이미 open된 디스크립터라면 O\_NONBLOCK파일 상태 플래그를 ON하기 위해 fcntl(옵션 변경 명령어) 호출
3. int pfd[2];
4. if(pipe (pfd) == -1)
5. fatal ("pipe call");
6. if(fcntl(pfd[0],F\_SETFL,O\_NONBLOCK)==-1)
7. fatal ("fcntl call");

이렇게 하면 read, write에서 -1반환하여 block되지 않고 내용 처리

**FIFO (named PIPE)**

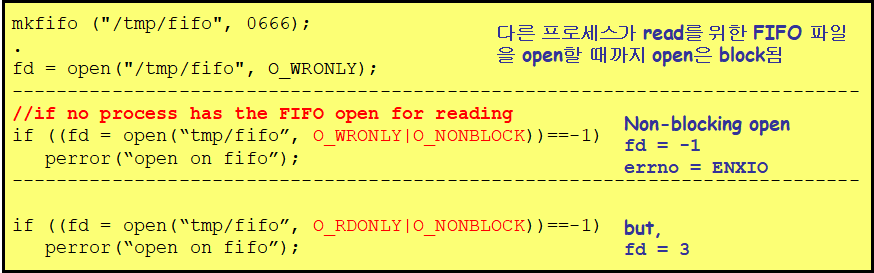
이름이 있는 파일을 사용하여 통신을 하여 부모 프로세스와 무관하게 전혀 다른 모든 프로세스들 사이에서 통신이 가능 (사용법은 파이프와 동일)

Makfifo() system call – 주어진 파일을 FIFO파일로 만들어 줌

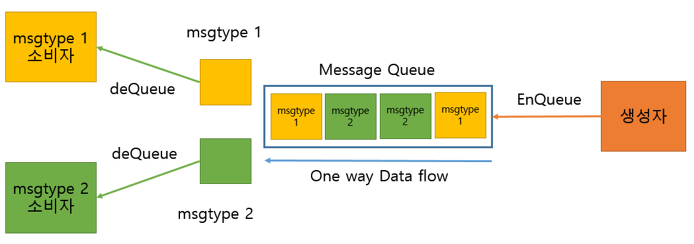
|  |  |  |
| --- | --- | --- |
|  | PIPE | FIFO |
| 파일 | 이름이 없다 | 일반 파일처럼 이름 있다 |
| 사용 범위 | 공통의 조상이 파이프를 만들었을 때만 관련 프로세스간 사용 가능 (부모-자식) | 관계없는 프로세스간 데이터 교환 가능 |
| 접근 권한 | Open시 자동으로 r, w권한 주어진다. | r,w 둘 중 하나로 권한 부여해야 함 |

Open()함수에서 O\_NONBLOCK flag

* 지정이 안된 경우
  + Read\_only로 open: 다른 프로세스가 write을 위한 fifo를 open할 때까지 block
  + Write\_only로 open: 다른 프로세스가 read를 위한 fifo를 open할 때까지 block
* 지정된 경우
  + only로 open: 파일 디스크립터를 즉시 리턴
  + Write\_only로 open: read를 위한 FIFO를 open한 프로세스가 없다면 errno를 ENXIO설정하고 -1 리턴

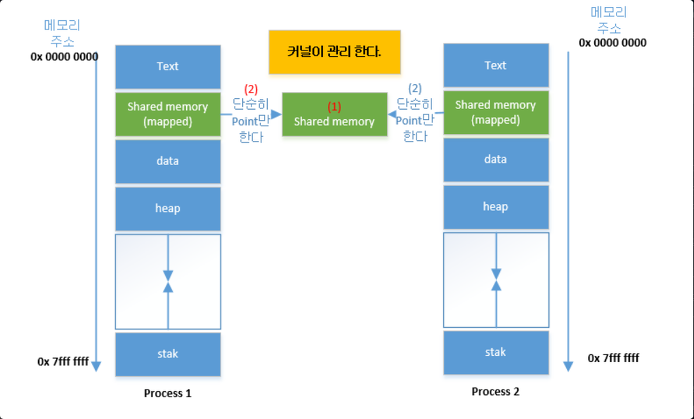


**Message Queue**



선입선출의 자료구조를 가지는 통신설비로 커널에서 관리한다. 메시지 큐에 쓸 데이터에 번호를 붙임으로써 여러 프로세스가 동시에 데이터를 쉽게 다룰 수 있다.

**Shared memory**



데이터를 공유하는 방법: 통신을 이용(pipe, named pipe, message queue)과 데이터를 함께 사용하는 방법(shared memory)이 존재

프로세스의 개별 메모리 영역이외에 프로세스간 메모리 영역을 공유해서 사용하는 것을 허용하는 공간을 지칭한다. 프로세스가 공유메모리 할당을 커널에 요청하면 커널은 해당 프로세스에 메모리 공간을 할당해주고 어떤 프로세스든 해당 메모리 영역에 접근할 수 있다.

**공유 메모리는 중개자가 없이 곧바로 메모리에 접근가능하기 때문에 다른 모든 IPC들 중 가장 빠르게 작동한다.**

**Semaphore**

다른 IPC설비들이 대부분 프로세스간 메시지 전송을 목적으로 하지만, 세마포는 프로세스간 데이터를 동기화하고 보호하는데 목적을 둔다.

공유된 자원에 여러 개의 프로세스가 동시 접근하지 않고 단 하나의 프로세스만 접근 가능하도록 만들어 주는 역할

**Socket**

프로그램이 네트워크에서 데이터를 송수신할 수 있도록 “네트워크 환경에 연결할 수 있게 만들어진 연결부”

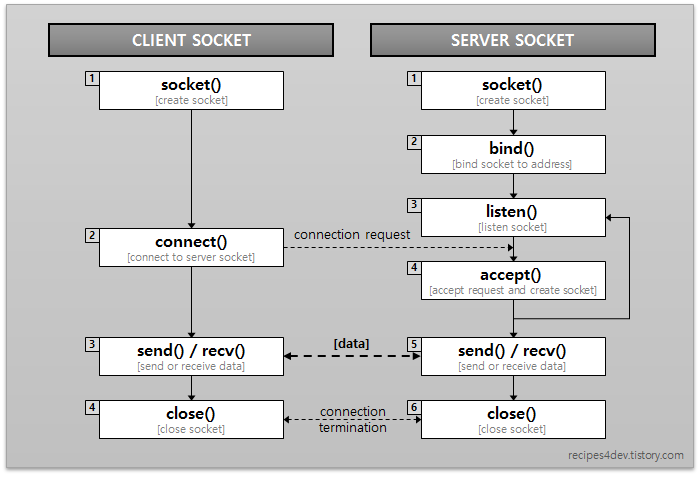
소켓 프로그래밍: 소켓을 사용하여 네트워크 통신 기능을 구현하는 과정

두가지 유형: UDP/TCP

1. UDP (User Datagram Protocol, 사용자 데이터그램 프로토콜) 🡪 신뢰성 없는 데이터 전송
2. TCP (Transmission Control Protocol, 전송 제어 프로토콜) 🡪 신뢰서 있는 데이터 전송

TCP 소켓 구현 과정

1. 서버 소켓
2. Socket(): 소켓의 생성을 요구
3. Bind(): 주소 할당을 요구 (IP주소와 Port 번호)
4. Listen(): 연결 요청 대기 상태로 진입 (client의 접속을 기다린다)
5. Accept(): 요청을 수락해서 클라이언트와 연결한다.
6. 이후 sent(), recv()를 통해 데이터 주고받음 (read(), write()함수와 같다)
7. 클라이언트 소켓
8. Socket(): 소켓 생성을 요구
9. Connect(): 서버에 연결을 요청하는 것으로 bind()가 자동으로 됨



UDP 서버와 클라이언트 소켓 구현 과정

* TCP와 달리 서버에게 연결요청을 하지 않고 바로 데이터 전송

1. Socket(): 클라이언트와 서버 둘 다 소켓 생성 요구
2. Bind(): 클라이언트와 서버에서 주소할당을 요구
3. 이후 바로 client가 server에게 데이터 전송하며 주고받는다.
4. Sendto(), recvfrom()