유닉스

**유닉스 기본 설명**

UNIX는 주로 서버용 컴퓨터에서 사용되는 운영체제

* 포털이나 대기업의 서버에 사용되며 보안성이 매우 뛰어남

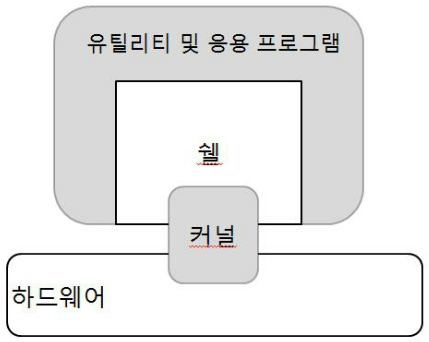
**UNIX는 미국 벨(Bell) 연구소에서 개발된 운영체제로, 프로그램 대부분이 C언어로 수정되면서 이식성이 높아지고 동시 다중 사용자 및 다중작업의 실행을 지원할 수 있는 대화형 소프트웨어이다.**

운영체제 코드가 공개되어 있지 않아 버그 이슈 같은 경우 IBM사에서 일괄 관리한다.

**특징**

1. 시분할 시스템(Time Sharing System)을 위해 설계된 대화식 운영체제 🡪 shell사용
2. 대부분 C언어로 작성되어 있어 이식성이 높으며 장치, 프로세스 간의 호환성이 높다.
3. 다중 사용자(Multi-user), 다중작업(Multi-Tasking)을 지원
4. 많은 네트워킹 기능을 제공하므로 통신망 관리용 운영체제로 적합
5. 트리구조의 파일 시스템을 가지고 있다
6. 프로그래머가 여러가지 언어를 사용하여 개발할 수 있도록 많은 컴파일러를 제공
7. 다양한 유틸리티 프로그램들이 존재 (ex. 통신 유틸리티)

**UNIX 시스템의 – 커널, 쉘 그리고 유틸리티 및 응용프로그램으로 나누어진다.**



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

CPU스케줄링, 기억장치 관리, 파일 관리, 입출력 관리, 프로세스간 통신, 데이터 전송 및 변환 등 **하드웨어와 관련된 작업을 수행하는 것으로 사용자들은 커널에 접근할 수 없다.**

-> 사용자는 접근 불가능하기 때문에 커널의 기능을 사용하기 위해 시스템콜이 존재!

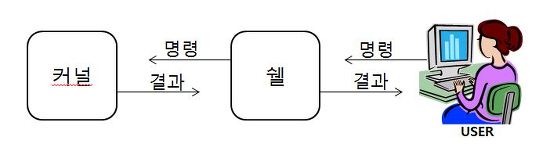
**시스템 호출(System call):** 커널과 사용자 사이의 인터페이스 역할 (프로그래밍 인터페이스)

운영체제가 제공하는 서비스를 유저가 이용할 수 있도록 해주는 것으로 쉘에서 명령어나 서브 루틴 형식으로 운영체제의 기능을 호출할 수 있다. 🡪 **운영체제의 기능을 호출하는 함수**

보통 시스템 콜을 직접 사용하기 보다는, 해당 시스템 콜을 사용해서 만든 각 언어별 라이브러리(API)를 사용한다.

1. **쉘(Shell)**

사용자의 명령어를 인식하여 프로그램을 호출하고 명령을 수행하는 **명령어 해석기**로 **사용자가 운영체제 기능과 서비스를 조작할 수 있도록 인터페이스를 제공하는 프로그램**이다. 유닉스 터미널



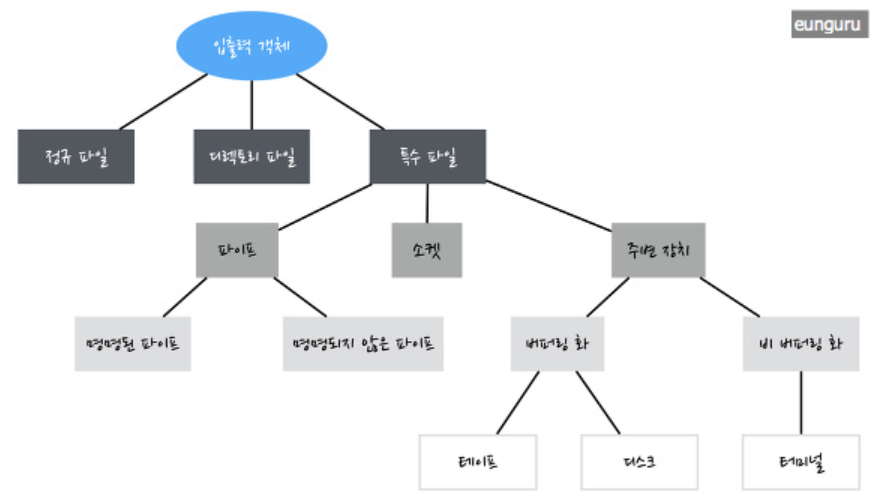
1. **유틸리티 및 응용 프로그램**: 유닉스는 많은 프로그래밍 언어를 지원하고, 파일작성, 파일 출력, 파일 내용을 선택적으로 변경시키는 기능 등과 같은 여러 가지 유틸리티와 다른 컴퓨터와 통신을 가능하게 해주는 여러 가지 응용 프로그램으로 구성된다.

**Unix File system (File and Directories)**

**File**

유닉스는 **일반적인 파일들 뿐 아니라 터미널, 프린터, 디스크 등 모든 주변 장치들도 하나의 파일로 취급** 🡺 장점: 옮기기, 쓰기 그리고 읽기를 간단하게 할 수 있다 (파일로 간주하지 않으면 프로그램을 작성해야 한다.)

파일 종류: 크게 세가지로 나눌 수 있다. (디렉토리, 일반파일, 특수파일)



1. 일반 파일, 정규파일 (regular file): 일련의 바이트 스트림으로 구성되고 표준 파일 입출력 시스템 호출을 통해 참조됨

각종 텍스트파일, 실행파일, 이미지 파일 등 유닉스에서 사용하는 대부분의 파일

(이미지, 실행 파일은 바이너리 파일에 해당)

1. 디렉토리 파일 (directory file): 디렉토리의 명시적인 시스템 호출을 통해 참조됨

우리가 아는 폴더로 유닉스는 파일로 취급한다. 즉, 디렉터리 파일은 다른 파일들의 목록을 가지고 있거나 그 파일들의 정보(주소)를 가리키는 포인터들을 가지는 파일

디렉토리 entry는 [inode number, character field(속해있는 file명)]로 구성된다.

읽는 권한이 있으면 그 디렉토리 안 파일 내용 또한 읽을 수 있다. 하지만 **디렉토리 파일에 직접적으로 쓰는 건 커널만 가능**

디렉토리 관련 시스템콜 – mkdir, chdir, chdir, opendir, closedir, readdir, rewinddir, getcwd, ftw

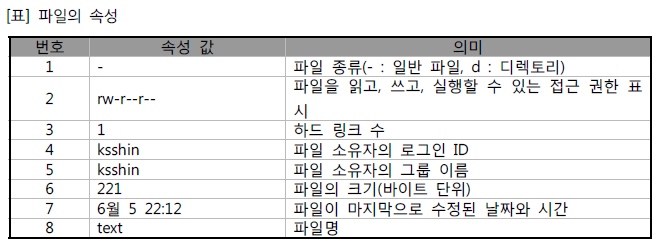
1. 특수 파일 (special file)
   1. 장치 파일 (Device File): 주변 장치나 프로세스 간 상호 통신기법에 해당되며 표준 입출력 시스템 호출을 통해 참조됨

보통 파일을 읽기 or 쓰기 요청을 하면 이 요청은 커널에게 전달돼서 파일스트림 드라이버에 의해 처리된다. 하지만 **장치파일 경우 요청이 그 장치의 드라이버에 의해 처리된다.** 장치 파일은 어떤 드라이버에 의해 사용되는지 구분하기 위해 연관 번호를 가진다.

* + 1. Block Special file: 일반 파일 과정과 유사, 일반 파일처럼 바이트 배열로 구성되어 있고 수정하거나 가장 마지막에 저장된 그 주소에서 값들을 읽어오고, 캐시도 가능하며 데이터를 쓸 때 버퍼에 놨다가 가져오는 것 등 비슷하다. (블록 단위로 데이터를 쓰고 읽는다 해서 유래) (디스크 파일)
    2. Character Special file: 파이프와 유사, 버퍼를 쓰지 않고 데이터를 쓰거나 읽는 것이 즉시 일어난다. 한 바이트만 쳐도 모니터에 출력 or 소리로 변환 등 (문자 하나만 보내도 개별적으로 처리된다고 해서 유래) (ex. 키보드, 마우스 입출력)
  1. FIFO: 프로세스간 통신에 쓰이는 파이프로 이름이 있는 파이프 파일을 의미. 파이프 (Pipe)라고도 부른다. (나중에 자세히 설명)
  2. 소켓 (Socket): 네트워크의 입출력을 담당하는 API로 두 호스트 컴퓨터 사이의 정보를 전달 (네트워크 통신에 필요한 파일, 나중에 자세히 설명)
  3. (추가- 이건 특수 파일에 포함인가..? 그런듯) 심벌릭 링크 : (다른 파일을 가리키는 타입의 파일) 원본 파일을 대신해 다른 이름으로 파일명을 지정한 것으로 윈도우의 바로가기 파일과 비슷

**파일의 속성 ( ‘ls -l’ 명령어를 통해 해당 디렉토리와 거기에 속한 파일 정보 알 수 있다.)**





1. 파일 종류: 위에서 소개한 파일 중 어떤 유형인지 알 수 있다.



1. 파일의 접근 권한(file mode): 다중 사용자 시스템인 유닉스에서 다른 사용자의 파일을 마음대로 사용할 수 없도록 하는 보안 기능을 제공한다. 사용자가 자신의 파일에 접근 권한을 부여

권한 부여의 대상을 세가지로 구분하며 각각에 대해 읽기(r), 쓰기(w), 실행(x) 권한이 존재한다. 표기는 user, group, other 순이고 권한 없으면 -로 대체표시 (chmod 통해 소유자가 변경 가능)

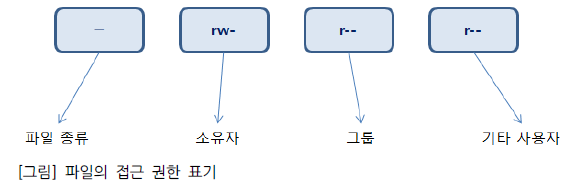
User: 파일을 만든(생성) 소유주

Group: 파일을 만든 소유주가 속한 그룹의 사용자

Other: 기타 사용자

접근 권한 기본값 – umask(새로 생성될 파일에 영향을 미치는 명령으로 지정한 8진수는 새로 만들어질 파일에서 제거될 권한을 명시)로 지정한다.

~~일반(정규)파일: 666(rw-rw-rw-), 디렉토리: 777(rwxrwxrwx)~~



\*\* 참고 \*\* 디렉토리의 permission도 파일과 마찬가지 형태지만 의미가 약간 다르다.

Read – 하위디렉토리와 파일들의 이름을 나열

Write – 파일 생성, 제거

Execute - search권한과 동일, chdir 시스템 콜을 사용할 수 있다. (특정 filename을 찾기 위한 탐색)

예를 들어, /usr/include/stdio.h 파일을 열기 위해서는 /, /usr, /usr/include 디렉토리의 실행 권한이 필요하다.

1. 하드 링크 수 (Hard link): 링크 카운터라고도 하며 동일한 inode를 참조하는 파일 수를 의미
2. 파일 소유자의 사용자 이름: 해당 디렉토리 및 파일의 소유주로 디렉토리 및 파일을 생성한 UNIX 시스템의 사용자 계정이 지정된다. 추후 변경이 가능(chown 명령 사용)
3. 파일이 속한 그룹의 이름: 소유주가 소속된 기본 그룹명이 지정되며 역시 추후 변경 가능(chgrp 명령)

**File system**

디렉토리와 그 안에 저장된 파일을 찾기 쉽도록 유지/관리하는 시스템을 말한다.

일반파일과 특수 파일로 구성

* 일반 파일: 아스키 파일, 링크된 파일, 실행 파일 등
* 특수 파일: 디렉터리, 장치 파일 등

계층구조로 되어있고 디렉토리의 시작은 root (/.)이다.

**디렉토리 이동 (cd 명령어 이용)**

Absolute(절대 경로): 루트 디렉토리에서 시작한다. 경로이름이 “/”로 시작하면 절대경로로 인식

Relative(상대 경로): working directory(현재 디렉토리, 모든 프로세스가 가진다.)기준으로 파일위치 표현

참고\*\* Home directory: login할 때 첫번째 directory로 초기 working directory는 home 디렉토리

**파일 접근 기본적인 시스템 콜 (파일접근 primitive)**

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: 한 파일에 연관된 속성을 제거

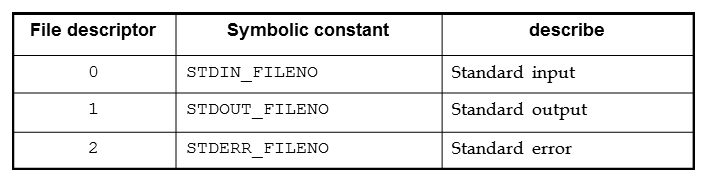
**File descriptor (파일 기술자)**

프로세스에서 특정파일에 접근할 때 사용하는 file을 가리키는 번호로 모든 오픈파일은 file descriptor를 갖는다. (시스템에 의해 결정되는 음이 아닌 정수 값)

이는 프로세스에서 열릴 파일의 목록을 관리하는 테이블의 인덱스이다.

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

프로그램이 실행될 때 기본적으로 할당되는 디스크립터들 (표준 입출력)



이러한 기본적으로 할당된 값들은 dup(), dup2()함수나 fcntl()함수 등을 통해 변경할 수 있다.

dup(), dup2()? 파일 디스크립터를 복사한다. Dup은 커널에서 알아서 디스크립터 번호를 할당해주는 것이며, dup2()는 프로그래머가 원하는 번호로 지정할 수 있다. (단, 이미 사용 중이면 불가.)

파일 디스크립터 번호가 반환되고 실패시 -1이 반환된다. (dup은 복사, dup2는 redirection)

예를 들어, fd = dup(0); 을 수행하면 새로운 파일 디스크립터 fd가 생성되고 fd는 표준입력(0)의 기능을 수행할 수 있다. 즉 fd를 이용해 scanf()함수 가능하다.

\*\*참고\*\* ssized\_t : UNIX버전마다 각 시스템콜에서 사용하는 자료형이 다를 수 있다. 버전 사이에 호환성을 유지하기 위해 사용되는 자료형이다. \_t로 끝나는 자료형들을 primitive system datatype이라 한다.

**프로세스의 파일 공유**

유닉스에서 프로세스간 파일을 공유하기 위해 3가지 테이블이 필요하다.

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

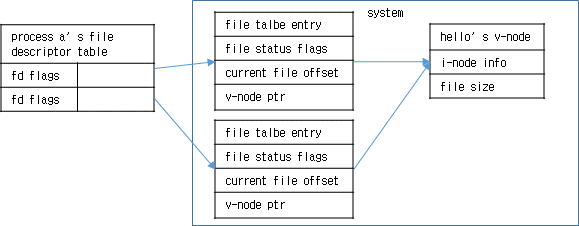
File descriptor table(=process table entry): 프로세스마다 한 개 생성되며 다음과 같이 구성된다.

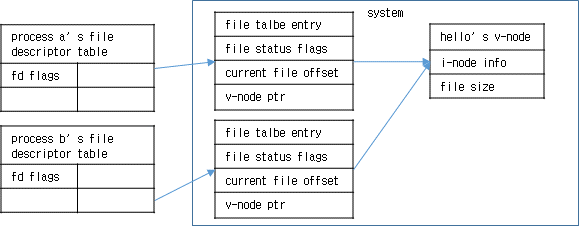
* Fd flag: open에서 사용한 상태 flags
* File pointer: 파일 테이블 요소의 위치 정보

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 같은 파일이 두 번 열린 경우: 서로 다른 파일 디스크립터 부여 및 서로 다른 entry를 구성한다. 🡪 v-node pointer는 동일
* 다른 프로세스에서 같은 파일 열었을 경우: 각 프로세스의 파일 디스크립터 부여 및 서로 다른 entry를 구성한다. 🡪 v-node pointer는 동일

v-node table: v-node table의 각 entry는 open된 파일의 i-node정보와 파일의 현재 크기를 포함한다. (같은 파일이 여러 번 open된 경우 하나의 v-node entry가 존재한다. 때문에 **같은 파일을 사용하는 프로세스들은 v-node를 공유한다.**)

 🡨한 개의 process

 🡨 여러 process

File in context – 다중 사용자 환경에서의 파일

파일은 복잡한 다중 사용자 시스템에서 추가적인 특성을 갖는다.

프로세스에 부여되는 번호들 PID(프로세스 식별자), RUID, EUID, RGID, EGID

Owner: 모든 파일은 시스템의 한 유저에게 소유되고 user-id(uid)로 구분한다.

Group: 모든 유저는 하나 이상의 그룹에 속한다. Group-id(gid)

Owner와 group모두 유저가 만든 프로세스에 상속적이다. (파일이나 디렉토리는 만든 사용자가 그 소유자가 되며 소유자의 기본그룹이 파일이나 디렉토리의 그룹이 된다. ls-l로 확인 가능)

유효 uid & gid

* Real user-id(ruid): 프로세스를 시작한 유저의 uid (계정관리에 사용)
* Effect user-id(euid): 특정한 수행을 위해 부여받는 특권 (유효 사용자, 접근 권한 결정에 사용)
* 보통 ruid==euid이지만, special case 존재. (euid와 egid는 파일 접근을 결정한다)

\*) special case: S\_ISUID permission (rwx중 x자리에 s로 되어있으면 x와 s 둘 다 포함된 것)

리눅스는 서버용으로 만들어진 운영체제이므로 멀티유저 시스템이다. 🡪 때문에 여러 사용자가 접속 시 서버에 올려진 비밀자료 열람과 그것을 변조, 수정, 삭제를 할 우려가 높아진다. (일부 사람에게만 공개할 비밀자료가 모든 사용자들에게 공개됨) 이러한 문제를 극복하기 위한 리눅스의 기능은 permission기능(시스템에서 유지되는 권한기능)이다. (참고, 윈도우도 퍼미션기능 존재하지만 여러 사람들이 접속하는 서버용이 아니기 때문에 리눅스처럼 활성화되지 않았다.)

permission기능을 통해 사용자들은 자신에게 읽기 or 쓰기 or 실행권한이 부여된 파일만 부여된 권한에 맞게 접속가능해졌다. (파일이 권한이 부여된 사용자들에게만 공개됨)

또한, 리눅스는 멀티유저 시스템이기 때문에 파일 소유자 개념이 생김

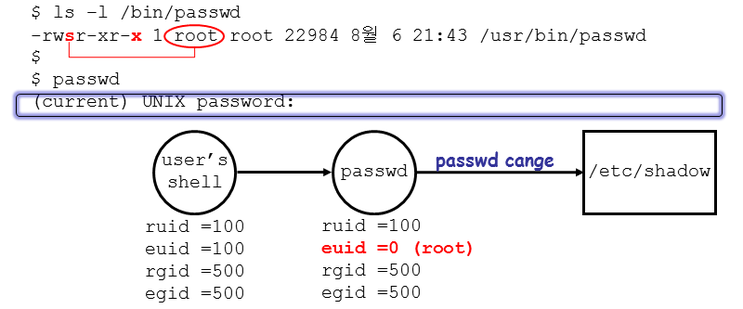
S\_ISUID가 set되어있으면 executable file이 실행될 때 그 프로세스를 시작한 유저의 uid가 아닌 그 파일의 소유주의 uid를 euid로 준다.

S가 set안된 경우: 다른 사람이 만든 프로그램 실행 시 실행중인 user의 id를 가져옴

S가 set된 경우: 소유주의 id를 상속받음 🡪 다른 사람이 내 서비스 파일을 접근할 때 내가 만든 프로그램 안에서만 가능하게 한 것

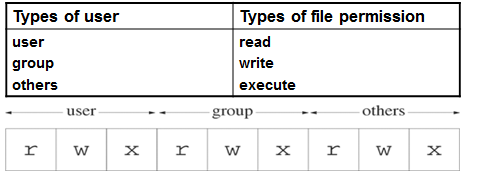
Ex) 비밀번호 변경

사용자는 passwd명령을 통해 자신의 패스워드 변경가능하다. 패스워드는 보통 /etc/shadow파일에 저장되고 이 파일의 소유자는 root며 사용자 접근이 불가능(-r--------)하다. 따라서 간접적으로 접근해 바꿔줘야 한다. /usr/bin/passwd의 소유자는 root이고 소유자의 접근권한은 rws이다. Suid가 설정되어있고 이것은 어떤 사용자가 프로그램을 실행해도 파일 소유자의 권한으로 실행된다는 것을 의미한다. 즉, 파일 실행 시 euid=소유자의 uid가 되어 /etc/shadow파일에 일반 사용자가 실행한 결과가 저장될 수 있다. 🡨 SUID가 설정된 파일의 소유자가 root라면 보안상 주의 필요!(이런 실행파일에 대한 공격이 성공하여 프로세스가 탈취하면 root권한까지 뺏길 수 있다. 밑에서 이 내용 다룸)



Ownership: 파일과 관련된 권한을 선택할 수 있다.

Permission: 여러 유형의 유저들(user, group, others)이 파일에 접근할 수 있는 방법(read, write, execute)을 결정한다. (superuser는 권한 고려 없이 접근 가능)



Open(): open함수 파라미터로 파일의 접근 모드가 전달되며 커널이 파일 접근 테스트를 수행한다. 이때 euid, gid를 기준으로 접근을 허가함

1. Euid == 0(superuser)이면 접근 허용
2. Euid == owner 이면 접근 허용 (단, user접근 허용 비트가 셋 되어야함 – rwx)
3. Egid == 파일의 gid이면 접근 허용 (단, group의 접근 허용 비트가 셋 되어야함)
4. Other 접근 허용 비트가 셋 되면 접근 허용

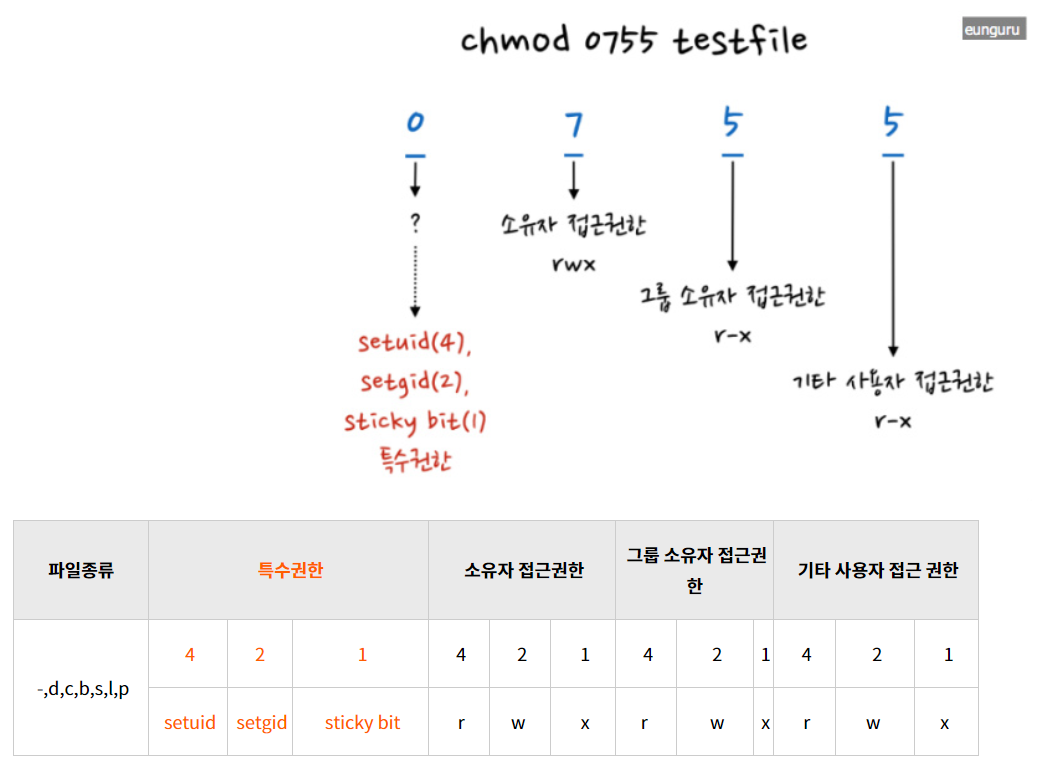
권한 관리 명령 chmod, umask / 소유주 변경 명령 chown, chgrp

파일 생성 마스크: 각 프로세스와 연관되어 있다. 파일이 생성될 때, 자동으로 특정 권한비트들을0으로 바꾼다. Permission이 실수로 켜지는 것을 방지해 파일 보호 (default로 작동됨)

* System call: umask(mode\_t cmask) 기존 mask값을 반환

Chmod 시스템 콜: 파일의 permission을 수정, 파일의 owner나 슈퍼유저만 가능

Chmod [옵션] [변경할 퍼미션 값] [변경할 파일]



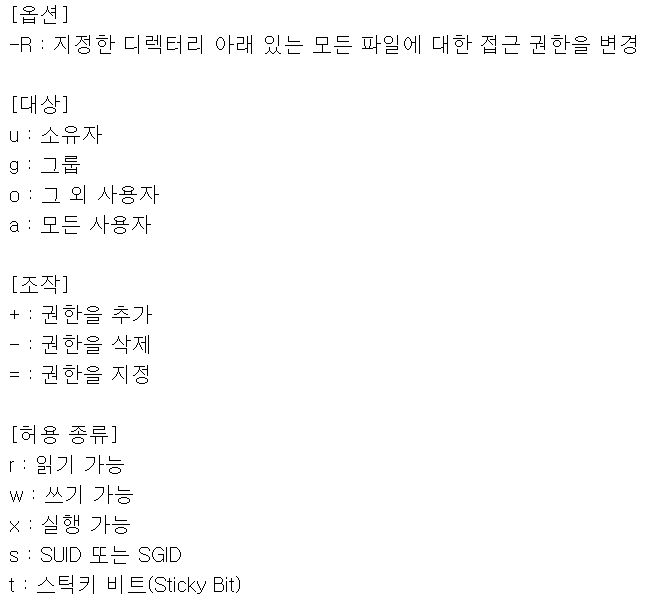
Unix 시스템은 파일에 대한 접근권한 및 파일 종류를 나타내기 위해 16bit를 사용한다.

각 3bit씩 총 9bit는 user, group, other(기타 사용자)의 권한 기술하는데 사용

4bit는 파일의 종류 표현에 사용

3bit는 특수 권한에 사용

<https://eunguru.tistory.com/115>

SUID: 어떤 사용자가 프로그램을 실행해도 파일 소유자의 권한으로 실행된다

SGID: 그 디렉터리에 생성된 파일이나 디렉터리의 소유 그룹은 디렉터리 자체의 소유 그룹이 적용된다. 즉, 누가 파일을 생성해도 파일의 소유 그룹은 원래 디렉터리의 소유 그룹으로 적용됨. 여러 사용자가 공유하는 디렉토리에 설정하면 편리하다.

Sticky Bit: 디렉토리에 대해만 설정한다. 설정 시 other의 실행권한이 ‘t’로 된다. (rwt) 스티키 비트가 설정된 디렉터리는 쓰기 권한이 있어도 자신이 소유한 파일 외에는 삭제할 수 없다. /tmp 디렉터리의 접근권한은 drwxrwxrwt이며 이는 모든 유저에게 쓰기 권한이 있지만 스티키 비트가 설정되어 다른 사용자가 생성한 파일을 삭제하는 것이 불가능 (write권한을 가지면서 파일의 소유주, 디렉토리 소유주, 슈퍼유저 경우만 지우고 이름 변경 가능)

위 특수권한의 표시가 대문자면 실행권한이 없다는 뜻, 소문자면 실행권한도 포함된다는 뜻

Chown 시스템 콜: 파일의 uid와 gid를 수정(파일이나 디렉토리의 소유주나 그룹을 변경할 때 사용), 슈퍼 사용자나 owner만 가능 🡪 불법적 접근시 오류 EPERM반환된다. Chown을 통해 파일 소유권이 바뀌면 set-user-id와 set-group-id 권한이 꺼진다. 🡪 권한이 꺼진다는 말은 파일 소유주의 id를 effective uid로 부여하지 않는다는 것! 즉, 파일에 대한 이전 소유주는 해당 파일에 이 명령어를 다시 실행할 수 없다. 🡪 권한 꺼지는거 중요! Chwon [변경할 소유자] [변경할 파일]

**파일 시스템**

디렉토리와 그 안에 저장된 파일을 찾기 쉽도록 유지/관리하는 시스템

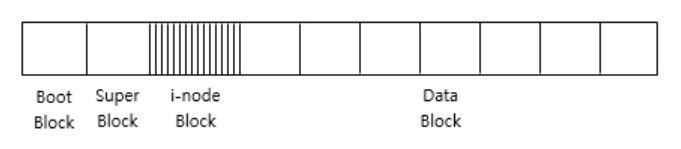
계층적 구조를 가지며 root가 존재.

**파일 시스템의 구조 (Unix File System, UFS)**

운영체제는 시스템의 disk partition상에 파일들을 연속적이고 일정한 규칙을 갖고 저장하는데 이러한 파일을 관리하는 규칙을 file system이라고 한다.

각 디스크 블록들은 4개 영역을 가진다.

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



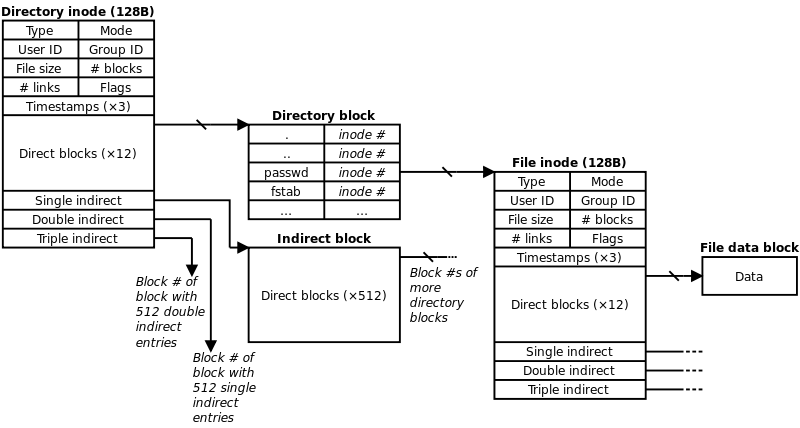
Boot block (부트 블록): 파일 시스템으로부터 UNIX 커널을 적재시키기 위한 프로그램이 저장됨

Super block (슈퍼 블록): 전체 파일 시스템에 관한 정보(파일 시스템에 있는 총 블록의 개수, 블록크기 등)를 저장

i-node: 파일이나 디렉터리에 대한 모든 정보를 가지고 있는 구조체

data block (데이터 블록): 실제 데이터가 파일의 형태로 저장되는 공간 (자유 블록을 포함하며 연결리스트로 유지)

**i-node**



파일의 소유권, 권한, 실제 데이터가 있는 물리적 주소, 파일의 형태, 크기, 링크 수, 수정/사용 시간 등과 같은 정보를 가진 약 120 byte 고정 크기의 구조체로 외부적으로는 번호로 표현된다.

(명령어 ls -l을 실행했을 때 나타나는 정보 중 파일 이름을 뺀 나머지 정보를 모두 포함)

(inode 포인터 구조를 통해 파일의 실제 데이터가 저장된 블록의 정보를 포함하여 파일의 메타 데이터 정보만 저장시킴.)

파일 이름은 inode의 번호와 함께 directory안에 저장된다. (한 inode를 가리키는 경로 개수를 의미하는 link count의 값이 0이되면 파일시스템에서 clear됨. 매칭되는 경로 없으므로)

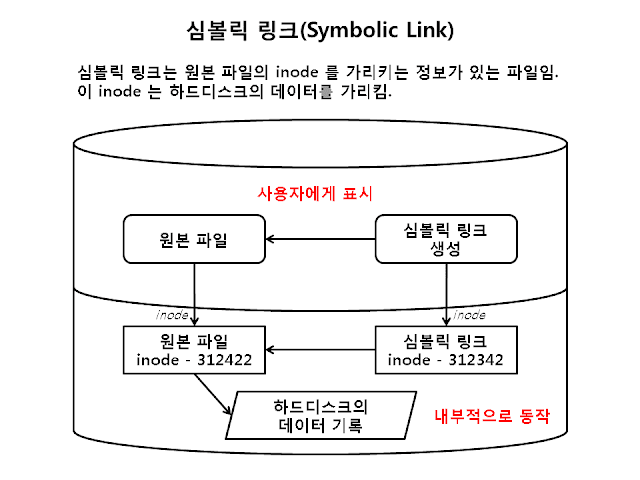
파일 시스탬 내에서 파일이나 디렉토리는 고유한 inode를 가지고 있으며 inode 번호를 통해 식별 가능하다. (key역할)

* 사용자가 파일 또는 파일과 관련된 정보에 접근할 때 파일 이름을 사용하지만 내부적으로는 파일 이름은 먼저 디렉토리 테이블에 저장된 inode 번호로 매핑되며 그 후 inode번호를 통해 해당 inode에 액세스 됨.

Inode 번호는 0,1을 빼고 사용한다. (0: no i-node를 의미, 1: bad disk block들을 모은다)

Inode 2는 root directory(/)에 예약되어있다.

**심볼릭 링크 (Symbolic link) – symlink() system call**



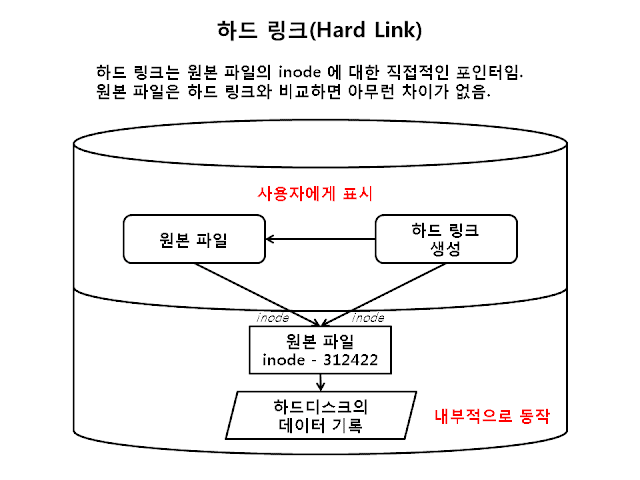
윈도우 시스템에서 제공하는 바로가기 기능과 매우 유사하다.

원본 파일에 대한 정보가 포함되어 있지 않으며 원본 파일 위치에 대한 포인터만 포함되므로 새로운 inode를 가진 링크 파일이 생성된다.

명령어: ln -s [원본 파일] [링크 파일]

Read()사용할 경우 원본 파일의 정보가 읽힌다. 심볼릭 링크 파일에 저장된 정보를 읽고 싶으면 readlink()사용

**하드 링크 (Hard Link) – link() system call**



하드 링크는 디렉토리 구조에 대한 항목만 파일이 생성되지만 원본 파일의 inode 위치를 가리키므로 원본 파일과 inode번호가 같다.

명령어: ln [원본 파일] [링크 파일]

하드링크 vs 심볼릭 링크: 가장 큰 차이는 새로운 inode생성 여부이다. 이에 따라 원본 파일이 삭제될 경우 접근 가능 여부도 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) |

둘 다 생성하면 해당 디렉토리 block에 추가된다.

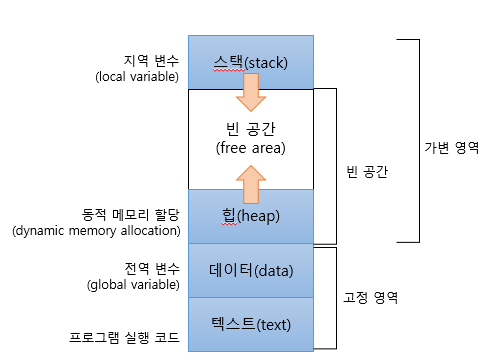
<https://12bme.tistory.com/215> (시스템콜)

**Process**

현재 실행 중인 프로그램을 의미

C언어(고급언어)로 작성한 프로그램 실행파일 만들기: 컴파일러를 통해 시스템이 이해할 수 있는 기계어 프로그램으로 변환, 기계어로 변환된 프로그램을 실행 파일이라고 한다.

**프로세스 구조**

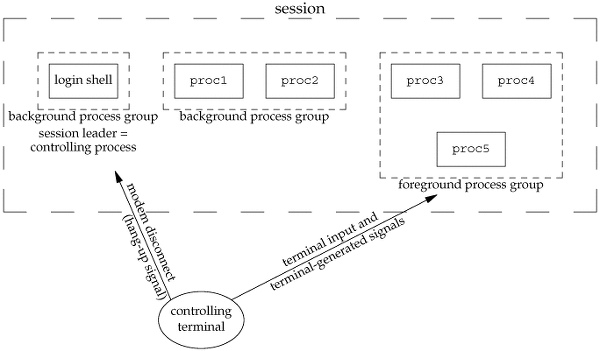


1. 텍스트 영역: 실행 코드를 저장한다. 텍스트 영역은 프로세스 실행 중 크기가 변하지 않는 고정 영역에 속한다.
2. 데이터 영역: 프로그램에서 정의한 전역 변수를 저장한다. 전역 변수는 프로그램을 작성할 때 크기가 고정되므로 고정영역에 할당된다. (컴파일 때 배열 크기가 정해진다.)
3. 힙(heap): 프로그램 실행 중에 동적으로 메모리를 요청하는 경우에 할당되는 영역으로, 빈 영역→할당→할당 해제처럼 상태가 변하는 가변 영역이다. (runtime시 배열 크기 유연)
4. 스택(stack): 프로그램에서 정의한 지역 변수를 저장하는 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.

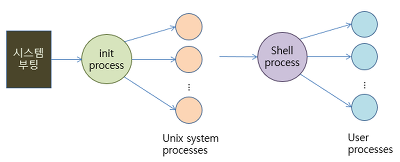
PID(process ID): 시스템에서 여러 프로세스를 식별하기 위해 사용 / getpid(), getppid()

PGID(Process group id): 관련 프로세스들이 모여 프로세스 그룹을 구성하며, 이를 구성하는 프로세스 중 하나가 그룹의 리더가 되고 리더의 PID가 PGID가 된다.

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. / getsid(), setsid() (세션 리더가 아닌 프로세스가 새로운 세션 생성하기 위해 사용)



Controlling terminal: 표출입출력에 연결된 treminal



시스템 프로세스: UNIX운영에 필요한 기능 수행

사용자 프로세스: 사용자들이 실행시킨 프로세스

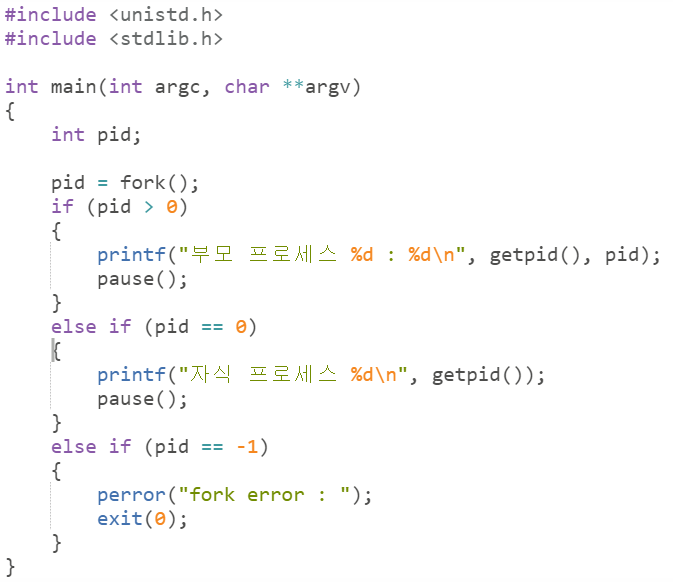
**프로세스 종류**

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 return & 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 종료해야함

**프로세스의 생성 (fork, exec)**

Fork: 새로운 프로세스를 위해 메모리를 할당받아 복사본의 형태로 프로세스를 실행, 기존의 프로세스는 그대로 실행

* 부모의 Stack, heap, data space를 copy (다른 메모리에 저장)
* 부모와 자식은 코드만 공유한다.
* Fork()함수의 반환값은 자식과 부모에서 다르다. (자식: 0반환, 부모: child의 pid반환)
* Fork전 open file이 있을 경우 부모와 자식은 같은 file table을 공유하게 된다. 이는 exec호출되어도 공유된다. (공유하기 싫은 경우 부모가 file lock을 걸면 된다 (close-on-exec flag 를 on으로 설정)



Exec: 원래의 프로세스를 새로운 프로세스로 대체하는 형태. 메모리에 새로운 프로세스의 코드를 덮어 씌움 (코드 영역을 바꿔주는 역할) – 코드의 첫 줄부터 수행된다.

**프로세스 종료 (exit())**

프로세스가 종료되는 경우는 두가지가 존재한다. 프로세스가 종료된 경우, kill 시그널 받은 경우

Normal termination: exit()실행 후 종료된 것, main()함수가 끝나면 자동으로 수행된다.

Abnormal: exit() 실행없이 종료된 것. (1) abort system call (잘 안 씀) (2) signal 받을 경우 (kill)

**Synchronizing process (wait함수를 이용해 자식 프로세스 종료까지 대기)**

부모 프로세스가 fork()함수를 사용하여 자식 프로세스를 생성했을 때, fork()함수가 반환되는 시점부터 2개의 프로세스가 동작하게 된다.

부모 프로세스가 자식 프로세스의 종료 상태를 얻기 위해 wait(&status)함수를 사용한다. 즉, wait()함수를 사용해 자식프로세스가 종료될 때까지 기다린 후, 그 이후에 wait(&status)다음 코드를 수행한다. 🡪 이를 하지 않을 경우 자식은 좀비 프로세스가 된다.

자식 프로세스의 pid를 반환, status에는 child가 어떤 이유로 종료됐는지 저장된다.

Waitpid(pid, &status, options); 여러 자식들중 특정 pid에 대해서만 기다린다. Options는 [WCONTINUED, WNOHANG, WUNTRACED]가 있다. 특히, WNOHANG옵션을 쓰기 위해 이 시스템콜을 사용 🡪 부모도 놀지말고 계속 실행하라. 자식이 종료 안되더라도 기다리지 않고 진행

**포그라운드 & 백그라운드 프로세스**

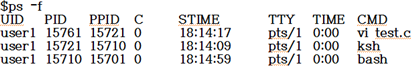
작업(job): 쉘이 프로세스를 관리하는 단위, 유닉스의 프로세스는 포그라운드 작업과 백그라운드 작업으로 구분 가능

포그라운드 처리: 사용자가 입력한 명령이 실행되어 결과가 출력될 때까지 프롬프트가 출력되지 않아서 다른 명령을 입력할 수 없는 처리방법

백그라운드 처리: 명령의 입력이 끝나면 결과에 상관없이 바로 다음 프롬프트가 출력되어 다른 명령을 입력할 수 있는 처리 방법, 명령의 끝에 &를 붙임 (명령의 처리시간이 오래 걸리는 경우 유용, 데몬 프로세스처럼 서비스 요청이 올 때까지 기다려야 하는 프로세스들 유용)

**프로세스 명령어**

1. 프로세스 목록 보기: ps(현재 실행중인 프로세스 정보 출력) – 옵션 (-e: 전체 프로세스 출력, -f: 자세한 정보 출력) (uid(소유자의 사용자 id), pid(프로세스 번호), ppid(부모 프로세스 번호))



1. 프로세스 종료: kill (root 계정이 아니면 사용자 프로세스만 제거 가능)
2. 포그라운드와 백그라운드 작업 제어: fg, bg, jobs [%작업번호] (작업번호 지정하지 않으면 백그라운드 작업 모두 출력, 특정작업번호 지정하면 해당 작업의 정보 출력 🡪 +가 가장 최근에 접근한 작업, -바로 전의 작업)



1. 작업 전환 및 종료 명령: bg[%번호], fg[%번호] (현재 작업이나 특정 작업을 백그라운드/포그라운드로 전환시켜 실행), ctrl+z (fg작업 중지 🡪 bg로 보냄), ctrl+c (fg작업 중지), kill %n (작업번호가 n인 작업 종료)
2. 사용자 전환: su(옵션 없는 경우 root계정으로 변경), su user1 (기존 사용자 환경을 유지한 상태에서 사용자만 user1로 변경), su – user1 (사용자 환경이 기존환경에서 suer1 사용자의 사용자 환경으로 변경된다. 즉, 작업 디렉토리가 user1의 홈 디렉토리로 변경됨)

**Signal (시그널)**

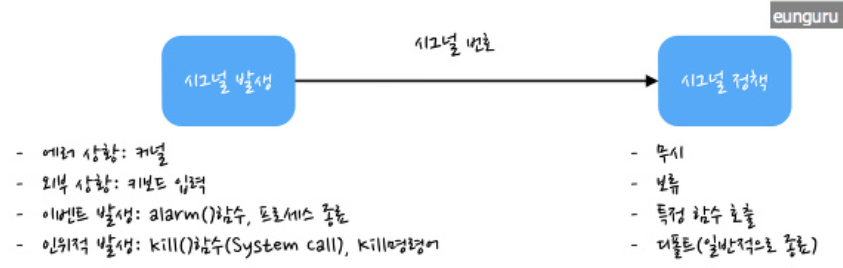
<https://jhnyang.tistory.com/143>

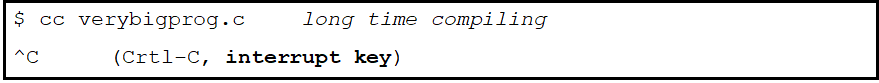
특정 이벤트가 발생했을 때 프로세스에게 전달하는 신호로 **시그널은 소프트웨어 인터럽트**이다.

시그널은 비동기식 이벤트를 다루는 방법을 제공 (에러가 아니므로 에러처리와 혼동하지 말 것)

시그널이 발생하는 경우 🡪 프로세스는 시그널을 받으면 처리를 위한 액션을 취한다.

* 외부에 의해 (ctrl + c같은 키보드 입력)
* 에러에 의해 (나누기 0 등)
* 이벤트에 의해
* 인위적 발생 (kill명령)



Ex) 

Kernel은 키보드 입력에서 인터럽트 종류를 확인 🡪 SIGINT 시그널을 foreground group에 속한 프로세스들에게 보내기 🡪 cc가 시그널을 받으면 SIGINT와 연관된 디폴트 액션을 수행하고 종료

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

시그널에 따른 기본처리(default action)는 프로세스 종료 or 코어 덤프를 생성

(코어 덤프 – 비정상적 종료 시 특정시점에 작업 중이던 메모리 상태를 기록)

**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.

**Signal handling**

프로세스가 시그널을 받으면 반드시 다음 3가지 액션 중 하나를 취한다.

Ignore Action

User-defined action – 커널에게 시그널 발생 시 호출한 함수를 말한다. (signal handler – 인터럽트 처리 루틴과 유사)

Default action – 모든 시그널은 default를 가지며 기본처리는 종료 (SIGSTOP, SIGTSTP, STICONT제외한 나머지)

**Process signal mask – process attribute**

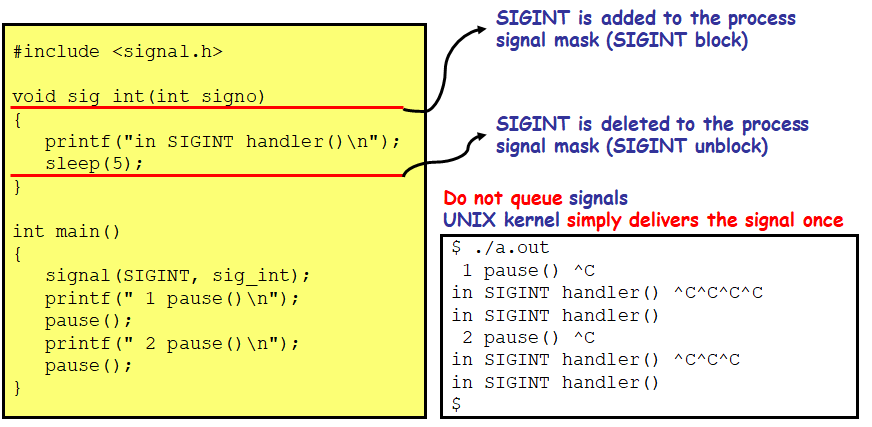
시그널이 생성될 때 취해지는 조치는 해당 시그널의 현재 시그널 핸들러와 프로세스 시그널 마스크에 따라 다르다.

프로세스 signal mask는 프로세스 attribute로서 signal blocking list이다. 이목록에 있는 시그널들은 blocking된다.

같은 시그널이 여러 번 들어온 경우 의미 없으므로 한번만 수행될 수 있도록 한다. 🡪 signal catching함수가 호출되기 전에 프로세스의 시그널 마스크를 추가하여 같은 시그널 수행 중일 경우 block처리 된다. 처리 완료되면 pending을 풀고 block됐던 시그널을 처리한다.

시그널 처리 중 ^C를 계속 입력해도 한 개만 입력으로 저장해 뒀다가 sleep끝나면 처리

즉, signal mask에 의해 blocking되고 이 blocking된 시그널을 pending 시그널이라고 한다.



프로그램은 sigprocmask를 이용해 프로세스 시그널 마스크를 변경하여 신호를 차단

프로세스는 fork()와 exec()후에 시그널 마스크를 상속

fork까지는 signal hander 유효하지만 exec()는 프로그램이 달라졌으므로 유효하지 않다.

System call: signal(int signo, void (\*func) (int))

* 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))

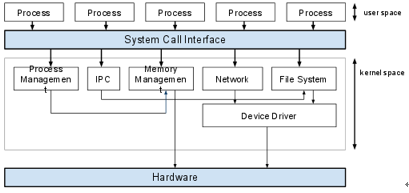
System call은 두가지 카테고리로 나뉜다.

1. “slow” system call – 중요하지 않은 시스템 콜이며 수행 중 시그널이 보내지면 시스템 콜 중단된다. (영원히 블락될 수 있다.)
   1. Pipes, terminal devices, and network devices
   2. Pause(), wait()
   3. 특정 ioctl 연산들
   4. 몇몇 interprocess communication functions
   5. Disk I/O와 관련되지 않는 것

시스템 콜이 중단되면 수행 실패로 -1을 반환한다. Sigaction은 시스템 콜이 중단되었을 때 자동적으로 재시작하는 것을 허용한다. (sigaction 구조체에서 sa\_flags변수를 SA\_RESTART로 설정)

1. All the other – read, write같은 시스템콜 수행 중 시그널 보내지면 시스템콜 완료 후 시그널이 수행된다.

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



그림과 같이 프로세스는 완전히 독립된 실행 객체이다.

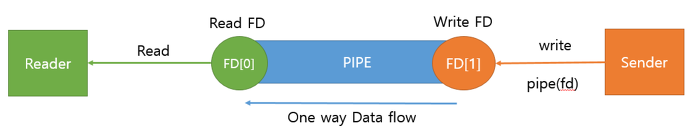
장점: 프로세스간 영향을 받지 않는다.

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

IPC 종류: 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**



파이프는 두개의 프로세스를 연결하고 한쪽 방향으로만 통신이 가능하기 때문에 Half-Duplex(단방향)통신이라고 부르기도 한다.

즉, 하나의 프로세스는 데이터를 쓰기만, 다른 하나는 데이터를 읽기만 할 수 있다. 따라서 하나의 프로세스가 송/수신을 모두 하길 원한다면 두개의 파이프를 만들어야 가능하다.



First-in-first-out 방식

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

프로세스가 파일에 read 호출한 경우

* 파이프 비어있지 않다면 – 즉시 read 리턴
* 비어있다면 – 파이프에 내용 적힐 때까지 read가 block됨 (기다림)

프로세스가 파일에 write 호출할 경우

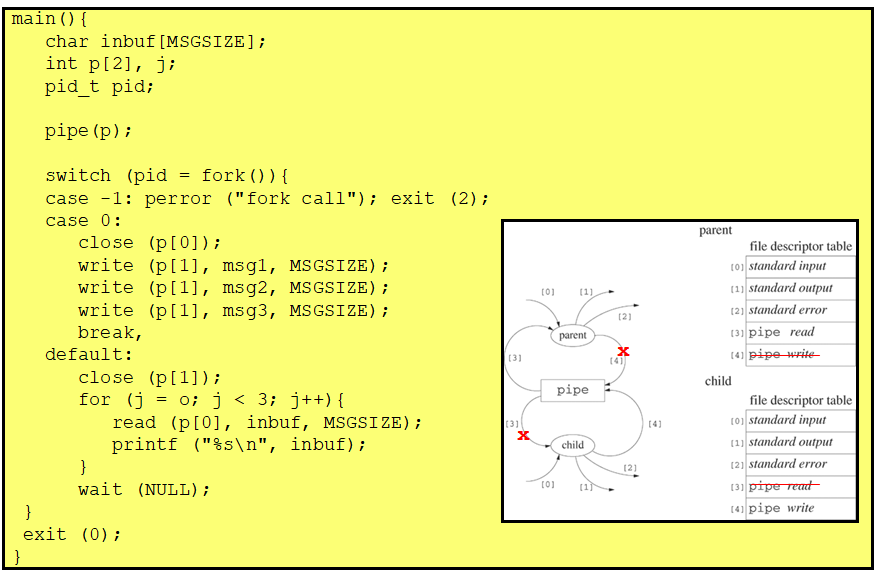
* 파이프가 꽉 차있지 않다면 – 즉시 write 리턴
* 파이프 꽉 차 있다면 – 파이프에 여유 공간 생길 때까지 write block

파이프 한쪽이 닫히는 경우

* write쪽이 닫히는 경우 (더 이상 write해줄 프로세스 없는 경우)
  + read는 모든 데이터를 읽은 후 EOF를 나타내기 위해 0을 리턴
  + read가 죽으면 write도 따라 죽음
* read쪽이 닫히는 경우 (더 이상 read할 프로세스 없는 경우)
  + SIGPIPE를 발생 (write가 더 이상 필요 없어진다)
  + 만약 시그널을 무시하거나 catch한 후 시그널 핸들러로부터 리턴하면, write()는 errno를 EPIPE로 설정하고 -1 반환
  + Write가 죽는다고 read가 죽지는 않는다.

두가지 제약

1. 단방향 통신에서 사용 (양방향 통신구현이 복잡하므로 가능하면 쓰지 않는 것이 좋다 – 한쪽 방향 close해주기)
2. 인자로 fd를 사용한다. 즉 변수가 공유되는 관계에서 pipe 사용 가능하다. (부모-자식)



Blocking & Non-blocking

시스템 콜은 호출자를 영원히 block할 수 있다. - read할 내용 없을 경우, write할 자리 없을 경우, 특정 파일 타입(FIFO)에 특정 조건이 발생할 때까지 block됨

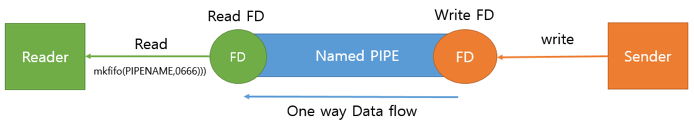
지정된 디스크립터에 대해 nonblocking I/O를 지정하는 두가지 방법

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)**

통신을 위해 이름이 있는 파일을 사용하기 때문에 부모 프로세스와 무관하게 전혀 다른 모든 프로세스들 사이에서 통신이 가능하다.

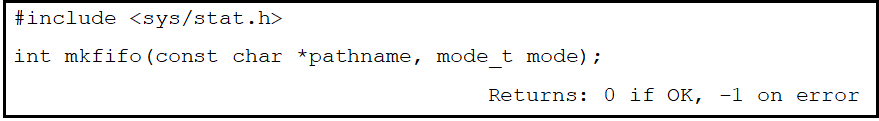


사용법은 파이프와 같다

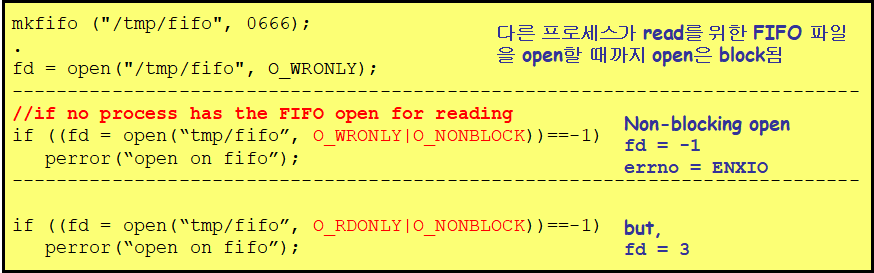
* First-in-first-out
* Half-duplex

|  |  |  |
| --- | --- | --- |
|  | 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 |

Mkfifo() system call – 주어진 파일을 FIFO파일로 만들어 줌

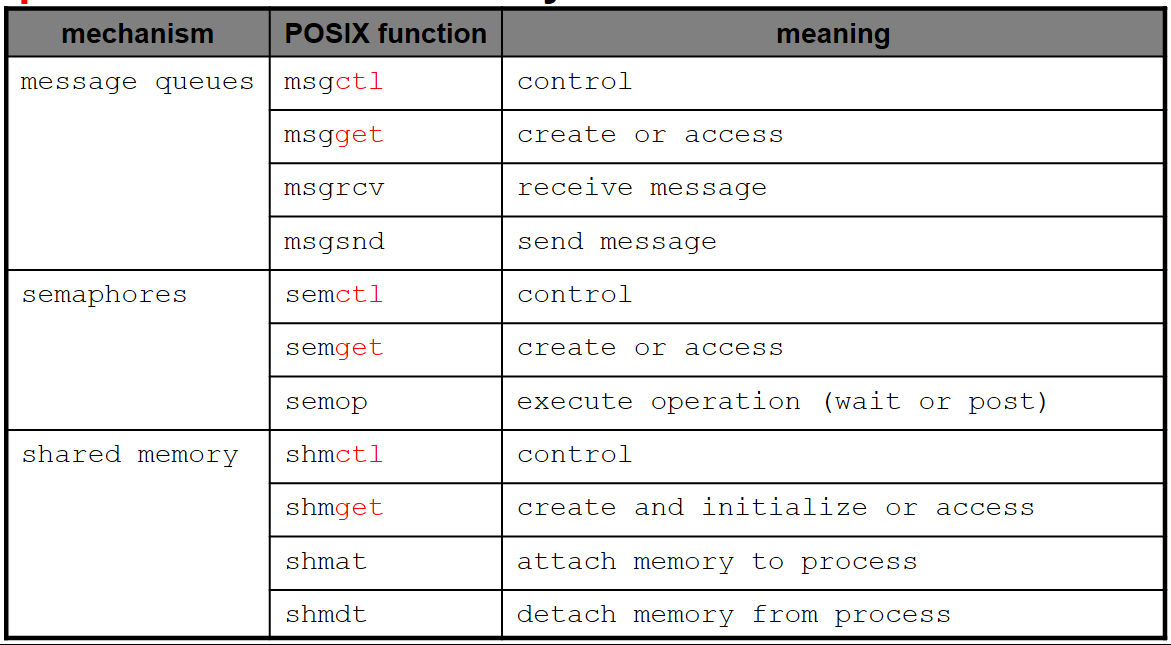


인자: pathname(FIFO 파일 이름), mode(권한 mask)

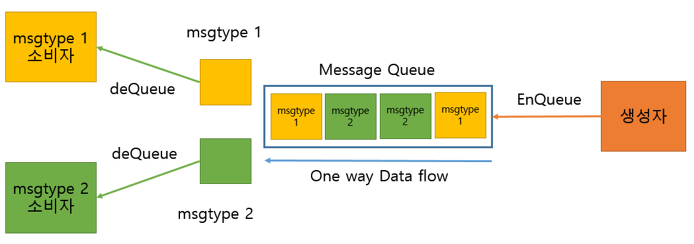


Open()함수에서 O\_NONBLOCK flag

* 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
* 지정 된 경우
  + Read\_only로 open: 파일 디스크립터를 즉시 리턴
  + Write\_only로 open: read를 위한 FIFO를 open한 프로세스가 없다면 errno를 ENXIO설정하고 -1 리턴



**Message Queue**



Queue는 선입선출의 자료구조를 가지는 통신설비로 커널에서 관리.

입출력 방식이 FIFO와 동일하다.

FIFO와 다른점은 FIFO는 데이터의 흐름이라면 메시지 큐는 메모리 공간이라는 점이다.

구조체 기반으로 통신을 한다.

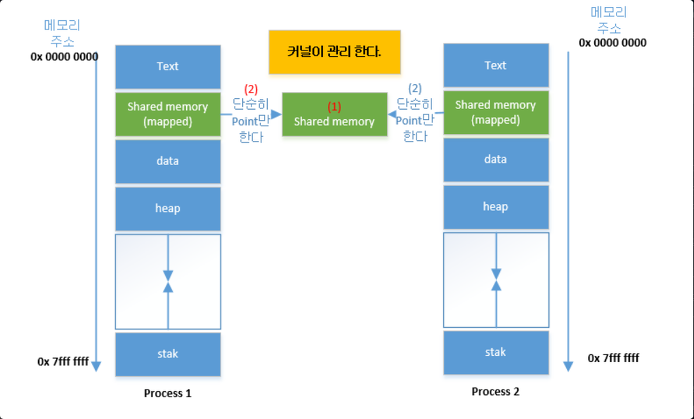
장점: 메시지 큐에 쓸 데이터에 번호를 붙임으로써 여러 개의 프로세스가 동시에 데이터를 쉽게 다룰 수 있다.

**Shared memory**

데이터를 공유하는 방법: 통신을 이용한 방법, 데이터 함께 사용 방법

통신을 이용한 방법: pipe, named pipe, message queue

데이터 함께 사용 방법: shared memory (공유메모리가 데이터 자체를 공유하도록 지원)



프로세스는 자신만의 메모리 영역이 존재하고 다른 프로세스가 접근해서 사용하려고 할 겨우 커널은 침범 프로세스에 SIGSEGV(경고 시그널 – 할당된 범위를 벗어나는 곳에서 읽거나, 쓰기를 시도할 때 발생)을 보낸다.

보호되는 영역이외에 프로세스간 메모리 영역을 공유해서 사용하는 것을 허용하는 공간을 shared memory라 한다. 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 접근할 수 있기 때문에 다른 모든 IPC들 중에서 가장 빠르게 작동한다.

**Semaphore**

다른 IPC설비들이 대부분 프로세스간 메시지 전송을 목적으로하는데 반해, 세마포는 프로세스간 데이터를 동기화하고 보호하는데 목적을 둔다.

프로세스간 메시지 전송 or shared memory사용하여 데이터 공유하게 될 경우 발생하는 문제를 보호한다. 즉, 공유된 자원에 여러 개의 프로세스가 동시 접근하지 않고 단 하나의 프로세스만 접근 가능하도록 만들어주는 역할을 한다. (자세한 내용 생략 – os)

**Socket**

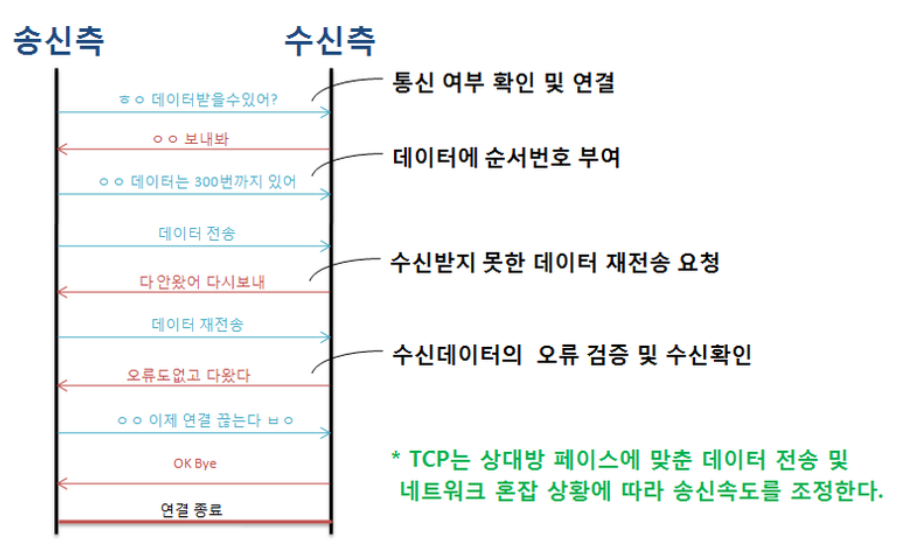
프로그램이 네트워크에서 데이터를 송수신할 수 있도록, “네트워크 환경에 연결할 수 있게 만들어진 연결부”

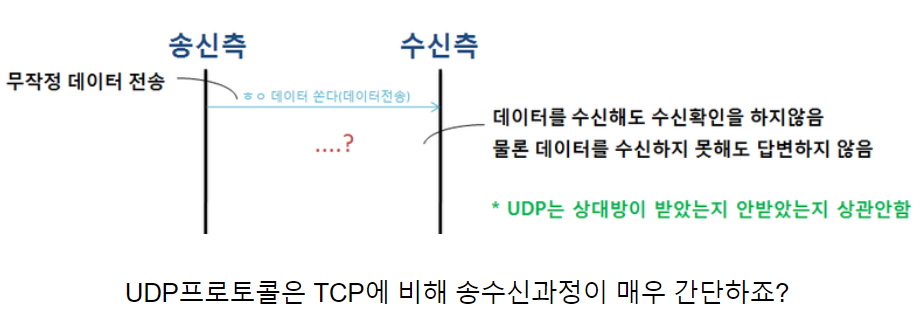
소켓 프로그래밍: 소켓을 사용하여 네트워크 통신 기능을 구현하는 과정

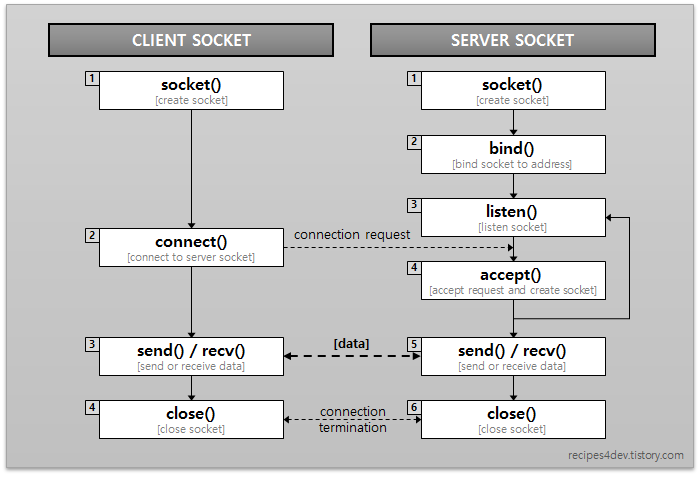
두가지 유형: UDP/ TCP (비교한 내용은 네트워크에서) – 간단 그림 설명

TCP (Transmission Control Protocol, 전송 제어 프로토콜) -> 신뢰성 있는 데이터 전송

UDP (User Datagram Protocol, 사용자 데이터그램 프로토콜) -> 신뢰성 없는 데이터 전송







서버 소켓 구현 과정 (TCP)

1. Socket(): 소켓의 생성을 요구
2. Bind(): 주소 할당을 요구 (IP주소와 Port 번호)
3. Listen(): 연결 요청 대기 상태로 진입 (client의 접속을 기다린다)
4. Accept(): 요청을 수락해서 클라이언트와 연결한다.
5. 이후 sent(), recv()를 통해 데이터 주고 받음 (read(), write()함수와 같다)

클라이언트 소켓 구현 과정

1. Socket(): 소켓 생성을 요구
2. Connect(): 서버에 연결을 요청하는 것으로 bind()가 자동으로 됨

서버와 클라이언트 소켓 구현 과정 UDP

TCP와 달리 서버에게 연결요청을 하지 않고 바로 데이터를 전송한다.

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()