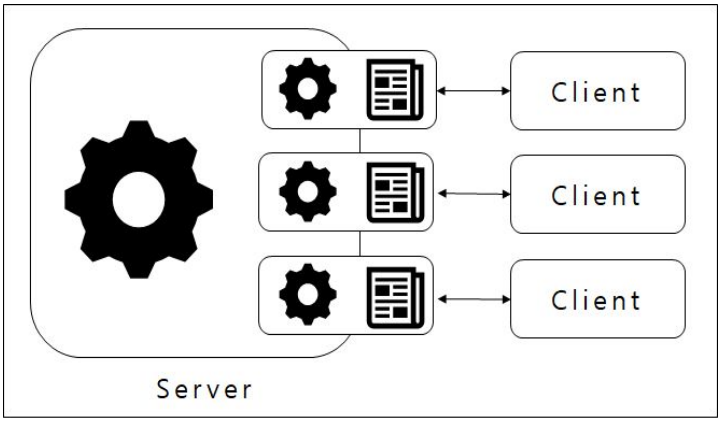
I/O Multiplexing

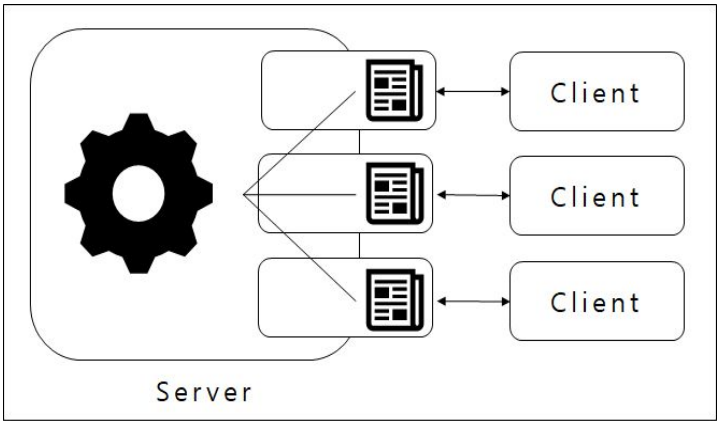
- 프로세스의 생성의 단점
    1. 많은 양의 연산이 요구
    2. 필요한 메모리 공간이 비교적 크다
    3. 프로세스마다 별도의 메모리 공간을 유지하기 때문에 상호간에 데이터를 주고받기 어려움이 있다. (IPC 이용)

- 프로세스의 생성을 동반하지 않으면서 다수의 클라이언트에게 서비스를 제공할 수 있는 방법 = I/O Multiplexing

- Multiplexing
    1. 하나의 통신채널을 통해서 둘 이상의 데이터(시그널)를 전송하는데 사용되는 기술
    2. 물리적 장치의 효율성을 높이기 위해서 최소한의 물리적인 요소만 사용해서 최대한의 데이터를 전달하기 위한 기술



< MultiProcess Model >



< MultiPlexing Model >

Select Function

- select 함수는 윈도우와 리눅스 모두 동일한 이름으로 동일한 기능을 제공하여 이식성이 좋다.

- 기능 및 호출순서
  1. 기능 : 한곳에 여러 개의 파일 디스크립터를 모아놓고 동시에 이들을 관찰할 수 있다.
     (1) 관찰 항목(event) 1. 수신한 데이터를 지니고 있는 소켓이 존재하는가?
     2. 블로킹되지 않고 데이터의 전송이 가능한 소켓은 무엇인가?
     3. 예외상황이 발생한 소켓은 무엇인가?
  2. 함수 사용

selete

#include < sys/selete.h >
#include < sys/time.h >

int select (int maxfd, fd_set *readset, fd_set *writset, fd_set *seceptset, const struct timeval *timeout)

- maxfd : 검사 대상이 되는 파일 디스크립터의 수
- readset : '수신된 데이터의 존재여부' 이벤트 정보를 모두 등록해서 그 변수의 주소 값 절달
- writset : '블로킹 없는 데이터 전송 가능여부' 이벤트 정보를 모두 등록해서 그 변수의 주소 값 절달
- exceptset : '예외상황의 발생여부' 이벤트 정보를 모두 등록해서 그 변수의 주소 값 전달
- timeout : 무한정 블로킹 상태에 빠지지 않도록 타임아웃 설정하기 위한 인자
- return : error = -1 / timeout = 0 / 이벤트 발생 = 이벤트 발생 파일 디스크립터의 수 > 0

  3. 호출 순서

(1) 파일 디스크립터의 설정
- 관찰하고자 하는 파일 디스크립터를 모은다. 이때 관찰항목(수신, 전송, 예외)에 따라서 구분해서 모아야 한다.
- fd_set형 변수를 통해 파일 디스크립터를 묶는다. [fd_set : 비트단위로 이뤄진 배열]
- fd_set에서 1로 설정되면 해당 파일 디스크립터가 관찰의 대상임을 의미한다.
- fd_set에 파일 디스크립터 등록을 위한 매크로
  1. FD_ZERO(fd_set *fdset) : 인자로 전달된 주소의 fd_set형 변수의 모든 비트를 0으로 초기화한다.
  2. FD_SET(int fd, fd_set *fdset) : fdset에 fd 정보 등록
  3. FD_CLR(int fd, fd_set *fdset) : fdset에서 fd 정보 삭제
  4. FD_ISSET(int fd, fd_set *fdset) : fdset에 fd 정보가 있으면 양수를 반환 [selet 호출 결과 확인용]

(2) 검사의 범위 지정
- select 함수의 첫 번째 매개변수를 통해 지정 가능하다.
- 값 : 가장 큰 파일 디스크립터의 값 + 1 [이유 : 파일 디스크립터의 값이 0에서 부터 시작하기 때문에]

(3) 타임아웃 설정
- timeval { long tv_sec; // seconds long tv_usec; //microsencods }
- timeout을 설정하고 싶지 않다면 NULL을 인자로 전달하면 된다.

Socket flags Option

- MSG_OOB : Out-of-band 형태로 데이터가 전송되려면 별도의 통신 경로가 확보되어서 고속으로 데이터가 전달되어야 하지만 TCP에서는 지원하지 않아 전송순서는 유지하되 앞의 데이터 처리를 신속하게 할 것을 요구한다.

fcntl(recv_sock, F_SETOWN, getpid( )) : recv_sock에 의해 발생하는 SIGURG 시그널을 처리하는 프로세스를 getpid 함수가 반환하는 ID의 프로세스로 변경시킨다.

- MSG_DONTWAIT : Non-blocking IO의 요구

- MSG_PEEK 옵션 : MSG_DONTWAIT 옵션과 함께 설정되어 입력버퍼에 수신 된 데이터가 존재하는지 확인하는 용도

  * recv함수를 호출하면 입력버퍼에 존재하는 데이터가 읽혀지더라도 입력버퍼에서 데이터가 지워지지 않는다.

readv & writev IO function

- 데이터 송수신의 효율성을 향상시키는데 도움이 되는 함수
- 데이터를 모아서 전송하고, 모아서 수신하는 기능의 함수

writev : 여러 버퍼에 나뉘어 저장되어 있는 데이터를 한번에 전송

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

ssize_t  writev( int filedes, const struct iovec * iov, int iovcnt )
```

- filedes : 데이터 전송의 목적지 소켓의 파일 디스크립터 / 파일이나 콘솔도 대상이 될 수 있다.
- iov : 구조체 iovec 배열의 주소 값
- iovcnt : iov가 가리키는 배열의 길이정보 전달

```
struct iovec {
    void *  iov_base;  // 버퍼의 주소 정보
    size_t iov_len;    // 버퍼의 크기 정보
}
```

readv : 데이터를 여러 버퍼에 나눠서 수신

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

ssize_t  readv( int filedes, const struct iovec * iov, int iovcnt )
```

- filedes : 데이터 수신할 파일 / 소켓의 파일 디스크립터
- iov : 데이터를 저장할 위치와 크기 정보를 담고 있는 iovec 구조체 배열의 주소 값
- iovcnt : iov가 가리키는 배열의 길이정보 전달

Multicast

- 멀티캐스트 방식의 데이터 전송은 UDP를 기반으로 하며 UDP 서버/클라이언트의 구현방식이 매우 유사하다.

- 단 한번에 데이터 전송으로 다수의 호스트에게 데이터를 전송할 수 있다.

- 특성
  1. 멀티캐스트 서버는 특정 멀티캐스트 그룹을 대상으로 데이터를 딱 한번 전송한다.
  2. 멀티캐스트 그룹의 수는 IP주소 범위 내에서 얼마든지 추가가 가능하다.
  3. 특정 멀티캐스트 그룹으로 전송되는 데이터를 수신하려면 해당 그룹에 가입하면 된다.

- 다수의 UDP 패킷을 전송하는 것이 아닌 하나의 패킷을 네트워크상에서 라우터들이 복사하여 다수의 호스트에게 전달

- 멀티미디어 데이터의 실시간 전송 : 하나의 영역에 동일한 패킷이 둘 이상 전송되지 않는다.

- 적지 않은 수의 라우터가 멀티캐스트를 지원하지 않기 때문에 터널링 (Tunneling) 기술을 사용

터널링 (Tunneling)
- 한 네트워크에서 다른 네트워크의 접속을 거쳐 데이터를 전송하는 기술 - 두 번째 네트워크에 의해 운송되는 패킷들 내에 네트워크 프로토콜을 캡슐화함으로써 운영 - 멀티캐스트 주소를 가진 패킷 헤더 앞에 멀티캐스트 라우터간에 설정된 터널의 양 끝단의 IP를 덧붙여 라우팅을 함으로써 멀티캐스트를 지원하지 않는 라우터를 거칠 때 유니캐스트 패킷과 같은 방법으로 리 이루어 지도록 한다.

- 멀티캐스트 패킷의 전송을 위해서는 '패킷을 얼마나 멀리 전달할 것인가'를 나타내는 TTL을 반드시 설정해야 한다.

TTL (Time to Live) 설정 방법
- 프로그램상에서의 TTL 설정은 소켓의 옵션설정을 통해 이루어 진다. (setsockopt 함수) - TTL의 설정과 관련된 프로토콜의 레벨은 IPPROTO_IP이며 옵션의 이름은 IP_MULTICAST_TTL이다. - ex) setsockopt(send_sock, IPPROTO_IP, IP_MULTICAST_TTL, (void*)&time_live, sizeof(time_live))

멀티캐스트 그룹으로의 가입
- 소켓의 옵션설정을 통해 이루어진다. (setsockopt 함수) - 프로토콜의 레벨은 IPPROTO_IP이고, 옵션의 이름은 IP_ADD_MEMBERSHIP이다. - struct ip_mreq 구조체를 사용한다. - ex) setsockopt(recv_sock, IPPROTO_IP, IP_ADD..., (void*)&join_adr, sizeof(join_adr));

```
struct ip_mreq {
    struct in_addr imr_multiaddr;      // 멀티캐스트 그룹의 주소 정보
    struct in_addr imr_interface;      // 그룹에 가입할 호스트의 주소 정보
}
```

Broadcast

- 한번에 여러 호스트에게 데이터를 전송한다는 점에서 멀티캐스트와 유사하지만 전송이 이뤄지는 범위에서 차이가 있다.
  (멀티캐스트는 다른 네트워크상에 존재하는 호스트라 할지라도, 멀티캐스트 그룹에 가입만 되어 있으면 데이터의 수신이 가능하지만 브로드캐스트는 동일한 네트워크로 연결되어 있는 호스트만 제한된다.)

- Directed Boradcast VS Local Broadcast
  -> Directed Boradcast : 네트워크 주소를 제외한 나머지 호스트 주소를 전부 1로 설정 [ex) 192.12.34.255]
  -> Local Broadcast : 255.255.255.255라는 IP주소가 특별히 예약되어 있다. = 해당 네트워크 모든 호스트에게 전달

- CODE

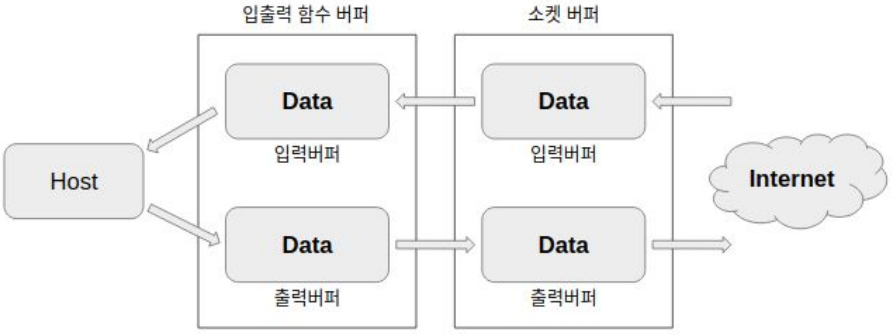
```
int send_sock;
int bcast = 1; // SO_BROADCAST의 옵션정보를 1로 변경하기 위한 변수 초기화
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
setsockopt( send_sock, SOL_SOCKET, SO_BROADCAST, (void*)&bcast, sizeof(bcast));
```

소켓과 표준 입출력

표준 입출력 함수의 장점

- 표준 입출력 함수의 두 가지 장점
  1. 표준 입출력 함수는 이식성(Portability)이 좋다 // 표준 함수 자체적 장점
  2. 표준 입출력 함수는 버퍼링을 통한 성능의 향상에 도움이 된다.

- 버퍼링을 통한 성능 향상
  1. 표준 입출력 함수에 의한 버퍼



< 버퍼는 기본적으로 성능의 향상을 목적으로 한다.>

  2. 소켓 버퍼는 성능의 향상보다는 TCP 구현을 위한 목적이 더 강하다. (재전송을 위한 데이터 저장 공간)
     -> 표준 입추력 함수 사용시 제공되는 버퍼는 오로지 성능 향상만을 목적으로 제공한다.
        - 성능 향상점
          (1) 전송하는 데이터의 양 [패킷 헤더에 의해 낭비되는 데이터 양을 절약]
          (2) 출력버퍼로의 데이터 이동 횟수

  3. 실제 파일 복사 프로그램을 통해 비교해보았을 때 데이터의 크기가 커질 수록 성능의 차이 또한 커진다.

- 표준 입출력 함수 사용의 단점
  1. 양방향 통신이 쉽지 않다.
  2. 상황에 따라서 읽기에서 쓰기로, 쓰기에서 읽기로 작업의 형태를 전환하는 fflush 함수의 호출이 빈번하다.
  3. 파일(소켓) 디스크립터를 FILE 구조체의 포인터로 변환해야 한다.

표준 입출력 함수 사용

- 표준 입출력 함수의 사용을 위해서는 이를 FILE 구조체의 포인터로 변환해야 한다.

fdoepn : FILE 구조체 포인터로의 변환
#include <stdio.h> FILE *fdopen (int fildes, const char *mode); - fildes : 변환할 파일 디스크립터를 인자로 전달 - mode : 생설할 FILE 구조체 포인터의 모드 정보 전달 ex) “r”, “w”

fileno : 파일 디스크립터로의 변환
#include <stdio.h> int fileno (FILE *stream) - 성공 시 변환된 파일 디스크립터, 실패 시 -1 반환

입력 출력 스트림의 분리

- 스트림 분리의 이점

1. 분리의 목적

   * 표준 입출력 함수을 이용하기 위해서 FILE 포인터가 필요하며 이는 읽기모드와 쓰기모드를 구분해야 한다.

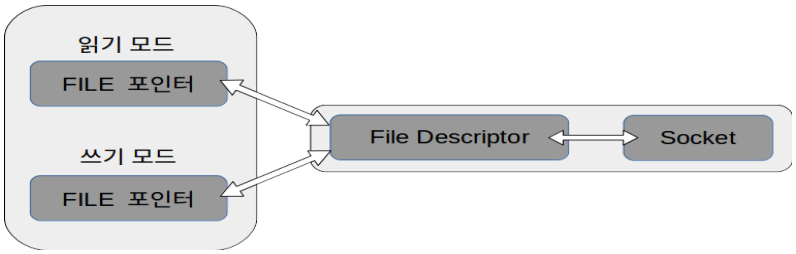
2. 분리의 장점

   읽기모드와 쓰기모드의 구분을 통한 구현의 편의성 증대, 입출력 버퍼 구분으로 통한 버퍼링 기능 향상

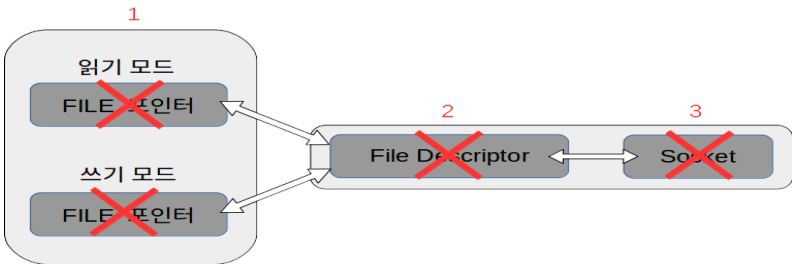
- 스트림 분리의 문제점 : fclose 시 Half-close가 아닌 full-close로 인한 EOF 문제

파일 디스크립터의 복사와 Half-Close

- 스트림 종료 시 Half-close가 진행되지 않는 이유



< 그림1 : FILE 포인터의 관계 >

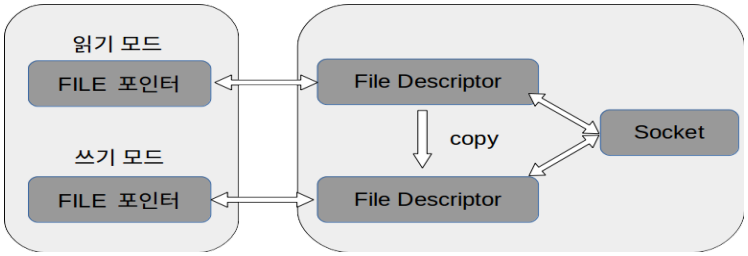


< 그림2 : fclose 함수호출의 결과 >

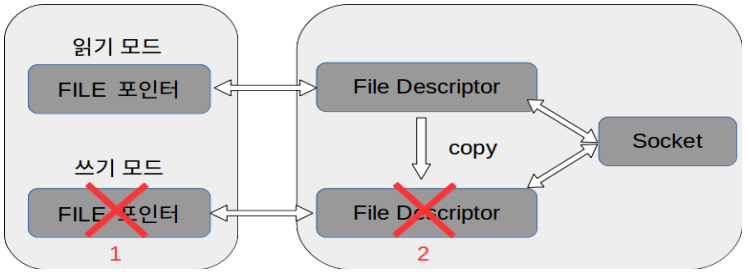
그림2에서 입력 or 출력 버퍼 둘중 하나가 소멸되면 File Descriptor와 소켓이 차례로 소멸된다. 소켓이 소멸되면 더 이상 데이터의 송수신은 불가능하다.

- Half-close 구현 방법

1. FILE 포인터 생성 전에 파일 디스크립터 복사



< 그림1 : Half-close를 위한 모델 1 >



< 그림2 : Half-close를 위한 모델 2>

위 과정을 통해 완전한 Half-close가 이루어진 것은 아니다. 그림 1, 2는 단순히 Half-close를 위한 환경이다.

- 파일 디스크립터의 복사

1. 파일 디스크립터 복사란? 동일한 파일 또는 소켓의 접근을 위한 또 다른 파일 디스크립터 생성
2. 구현

dup & dup2 : 파일 디스크립터 복사

```
#include <unistd.h>

int dup ( int fildes );
int dup2 ( int fildes, int fildes2 );
```

- fildes : 복사할 파일 디스크립터 전달
- fildes2 : 명시적으로 지정할 파일 디스크립터의 정수 값 전달
- 성공 시 복사된 파일 디스크립터, 실패시 -1

Selete VS epoll

- select기반의 IO Multiplexing이 느린 이유
  1. select 함수호출 이후에 항상 등장하는, 모든 파일 디스크립터를 대상으로 하는 반복문
  2. select 함수를 호출할 때마다 인자로 매번 전달해야 하는 관찰대상에 대한 정보들 [main 문제점]
     = select 함수를 호출할 때마다 관찰대상에 대한 정보를 매번 OS에 전달
     [관찰대상에 대한 정보를 OS에 절달해야 하는 이유 : select는 OS에 의해 기능이 완성되는 함수 = fd, socket OS 관리]

- 해결법 : OS에게 관찰대상 정보를 한번만 알려주고, 관찰대상의 범위 or 내용에 변경이 있다면 변경 사항만 알려준다.
  * 예시) 리눅스 : epoll | BSD : kqueue | 솔라리스 : /dev/poll | 윈도우 : IOCP

- select의 장점
  1. 서버의 접속자 수가 많지 않을 경우 유용하다.
  2. 다양한 OS에서 운영이 가능하다.

- epoll 함수 & 구조체