입출력 다중화

-입출력 관련 함수들을 봉쇄/동기적으로 작동하게 하기 위해선 하나의 파일에 하나의 프로세스나 쓰레드가 붙어서 작업을 해야 한다. 하지만 멀티 프로세스나 멀티 쓰레드는 프로세스, 쓰레드간 통신, 프로세스 동기화 같은 복잡한 프로그래밍 이슈를 가지고 있고, 생성 비용이 높다는 단점이 있다.

하지만 입출력 다중화는 단일 프로세스에서 여러 개의 파일을 제어할 수 있도록 한다. 하지만 병렬처리가 아니기 때문에 데이터 처리 과정이 짧은 메시지 전달 서비스에 적합한 모델이다.

-

1.

입출력 다중화는 “비 동기 / 봉쇄 입출력 모델”의 응용이다.

입출력 다중화는 여러 개의 파일을 다루기 위해서, 파일 기술자를 배열로 관리한다. 데이터 변경을 감시할 파일 기술자를 배열에 포함시키고, 배열에 포함된 파일 기술자에 (읽기, 쓰기, 에러와 같은) 데이터 변경이 발생하면 파일 기술자에 대응되는 배열에 표시하는 방식이다. 개발자는 파일 기술자 배열의 값을 검사하는 것으로 여러 개의 파일을 처리할 수 있게 된다.

2.

제한:

ㄱ. 파일 기술자 테이블의 크기

처리할 수 있는 파일의 최대 크기는 "프로세스가 열 수 있는 파일의 최대 개수"와 별도로 "파일 기술자 테이블의 크기"의 영향을 받는다. 일반적으로 파일 기술자 테이블의 크기는 1024인데, ulimit(1) 등으로 변경할 수 없다.

ㄴ. 배열로서 가지는 성능 문제

이벤트 기반이 아니기 때문에, 배열에 있는 모든 값을 전부 검사해야 한다. 1000개의 클라이언트가 연결되어 있고 이중 어느 하나에 데이터 변경이 이루어졌다면, 최악의 경우 1000개의 필드를 모두 검사해야 한다.

ㄷ. 병렬 처리가 아니다.

이 모델은 멀티 쓰레드와 같은 병렬 처리가 아니다. "데이터를 읽어서 처리하고 응답하는"시간 동안 다른 파일은 대기 해야 한다. 그러므로 데이터 처리 과정이 긴 서비스에 적용하기에는 적당한 모델이 아니다. 데이터 처리 과정이 짧은 메시지 전달 서비스에 적합한 모델이다.

3.

입출력을 관리하고자 하는 파일의 그룹을 fd\_set이라는 파일 비트 배열에 집어 넣고, 비트 배열의 값이 변했는지를 확인하는 방식이다.

fd\_set구조체는 단일 비트 필드 테이블 이라는 것에 주목해야 한다. 즉 fd\_set구조체는 이전 상태를 기억하지 못한다. 그러므로 select함수를 호출하기 전에 이전 fd\_set구조체의 값을 저장해 두어야 한다. 매번 fd\_set 정보를 복사해야 한다는게 select함수의 또 다른 단점이다.

1. select()

입출력을 관리하고자 하는 파일의 그룹을 fd\_set이라는 파일 비트 배열에 집어 넣고, 비트 배열의 값이 변했는지를 확인하는 방식이다.

2. select()

nfds : 관리하는 파일의 개수를 등록한다. 파일의 개수는 최대 파일 지정 번호 + 1로 지정하면 된다.

fd\_set : 관리하는 파일의 지정번호가 등록되어 있는 비트 배열 구조체

readfds : 읽을 데이터가 있는지 검사하기 위한 파일 목록

writefds : 쓰여진 데이터가 있는지 검사하기 위한 파일 목록

exceptfds : 파일에 예외 사항들이 있는지 검사하기 위한 파일 목록

timeout : select함수는 fd\_set에 등록된 파일들에 데이터 변경이 있는지를 timeout동안 기다린다. 만약 timeout시간동안 변경이 없다면 0을 반환 한다. timeout을 NULL로 하면, 데이터가 있을 때까지 무한정 기다리고, 멤버 값이 모두 0이면 즉시 반환한다.

retrun value : select 함수는 데이터가 변경된 파일의 개수 즉 fd\_set에서 비트 값이 1인 필드의 개수를 반환한다. 데이터가 변경된 파일의 목록을 반환하지 않는다는 것에 주의해야 한다.

3. pselect()

select() 함수의 만료 시간 값은 timeval 구조체로 하지만, pselect() 함수는 timespec 구조체를 사용한다. timespec 구조체는 나노초 단위로 표현하기 때문에 시간 정밀도가 높다.

pselect() 함수의 만료시간 인자는 const 로 선언되어 있기 때문에 pselect() 호출에 의해 값이 변경되지 않는다.

pselect() 함수의 마지막 인자 sigmask 를 통해서 신호 마스크를 지정할 수 있다. sigmask 가 NULL 이면 신호에 관해서 select() 함수와 동일한 기능을 하게된다. NULL 이 아니면 sigmask 가 가리키는 신호 마스크가 자동으로 설정되어 차단되고 pselect() 호출이 반환 될 때 신호 마스크가 복원되어 실행하게 된다.

1. poll의 함수원형 :

poll이 여러개의 파일을 다루는 방법은 select 와 마찬가지로 파일지시자의 이벤트를 기다리다가 이벤트가 발생하면, poll 에서의 block 이 해제되고, 다음 루틴에서 어떤 파일지시자에 이벤트가 발생했는지 검사하는 방식을 사용하게 된다.

2. pollfd 구조체

pollfd 구조체는 3개의 멤버변수가 있는데, 이 구조체에 우리가 관심있어하는 파일지시자를 세팅하고(fd), 관심있어 하는 파일지시자가 어떤 이벤트가 발생하는걸 기다릴것인지(events)를 지정하게 된다. 그럼 poll 은 해당 fd 에 해당 events 가 발생하는지를 검사하게 되고, 해당 events 가 발생하면 revents 를 채워서 돌려주게 된다.   
revents 는 events 가 발생했을때 커널에서 이 events 에 어떻게 반응 했는지에 대한 반응 값이다.  
후에 revent 값을 조사함으로써, 해당 파일지시자에 어떠한 event 가 최해지고 커널에서 그 event를 어떻게 처리했는지 (입력/출력이 제대로 이루어졌는지, 아니면 에러가 발생했는지)를 알아내서 적절한 조취(읽을 데이타가 있으면 읽거나 하는등의 일)를 취할수 있게 된다.

3. events

이 값들은 <sys/poll.h> 에 디파인 되어 있다.

4.

2번째 인자인 nfds 는 pollfd 의 배열의 크기 즉 우리가 조사할 파일지시자의 크기(네트웍프로그래밍측면에서 보자면 받아들일수 있는 클라이언트의 크기) 로, 보통 프로그래밍 할때 그크기를 지정해준다.   
마지막 아규먼트인 timeout 는 select 의 time 와 같은 역할을 한다.값을 지정하지 않을경우 이벤트가 발생하기 전까지 영원히 기다린다.

0일경우는 기다리지 않고 곧바로 다음 루틴을 진행하고자

0보다 큰 양의 정수일 경우에는 해당 시간만큼을 기다리게 된다. 해당 시간내에 어떤 이벤트가 발생하면 즉시 되돌려 주며, 시간을 초과하게 될경우 0을 return 한다.

5. Return 값

위의 3가지 아규먼트를 채워넣음으로써 poll을 사용할수 있다. poll 함수의 return 값은 int 형인데, 에러일경우 -1 이 리턴되고, 그렇지 않을경우 revent 가 발생한 pollfd 구조체의 숫자를 돌려주게 된다.

6.  
pollfd 에 입력된 파일지시자의 event 에 입력event 가 발생하면, 커널은 입력event 에 대한 결과를 되돌려줄것이다.   
이결과는 입력 event 가 제대로 처리되었다면 POLLIN 을 되돌려줄것이고, 어딘가에서 에러가 발생했다면 POLLERR 을 되돌려주게 될것이다.   
그러므로 우리는 revent 를 검사함으로써, 해당 파일지시자에 읽을 데이타가 있다는걸 알게 되고, 데이타를 읽어서 적당한 행동(여기에서는 주소를 돌려주는)을 할수 있다.   
위의 프로그램은 이러한 일련의 과정을 보여준다.  
select 버젼과 별차이가 없으므로 select 버젼의 쏘쓰를 이해했다면 위의 쏘쓰를 이해하는데 별 어려움이 없을것이다.   
  
poll 은 보통 select 에 비해서 해당파일지시자에 대해서 보다 많은 정보를 되돌려줌으로, 보통 select 보다 선호되는 추세이다.  
select 버젼과 마찬가지로 polling 중간에 파일 I/O 가 들어갈경우, 파일 I/O 작업에서의 block 때문에 짧은시간에 다수의 메시지를 처리할경우 문제가 될 소지가 있다.   
그러므로 되도록이면 polling 중간에 파일 I/O 가 일어나지 않도록 해주어야 한다.  
위의 쏘쓰의 경우도 주소정보를 미리 메모리 상에 올려놓고 쓰는게 더욱 좋은 방법이 될것이다.

1.Edge Triggered VS Level Triggered

ET로 작동하게 하려면 non-blocking 소켓에 사용해야만 합니다. non-block 파일디스크립터를 사용한다.

read(2) 나 write(2) 가 errno로 EGAIN을 반환할때만 wait 을 하도록 하자 (epoll\_wait)

LT 방식으로 epoll을 사용할 경우, select(:2)와 동일한 방식으로 사용할 수 있습니다. 물론 더 빠르겠죠

epoll은 LT 방식을 기본으로 실행됩니다. 만약 ET방식으로 지정하고 싶다면, 다음과 같이 하면 됩니다.

mev.events = EPOLLIN | EPOLLET ....;

epoll\_ctl(mepollfd, EPOLL\_CTL\_ADD, afd, &mev);

2. epoll\_create()

epoll\_create()는 이벤트를 저장하기 위한 **size**만큼의 공간을 커널에 요청합니다. 커널에 요청한다고 해서 반드시 size만큼의 공간이 확보되는 건 아니지만 커널이 대략 어느 정도의 공간을 만들어야 할지는 정해줄 수 있습니다. 수행된 후 파일 지정자를 되돌려 주는데, 더 이상 사용하지 않을 거라면 close(:2) 함수로 닫아주면 됩니다.

크기는 서비스 마다 다른데요. 일반적으로 예상 최대 동접 x 1.5 정도면 된다고 합니다.

retval:

해당 커널의 fd를 리턴합니다 error는 -1;

3. epoll\_wait()

실제 이벤트가 발생하는걸 기다리고 있다가, 이벤트가 발생하면 이벤트 관련 정보를 넘겨주는 일을 합니다.

**epfd**는 epoll\_create(2)를 이용해서 생성된 epoll 지정번호구요. 만약 이벤트가 발생하면 반환하는데, 이벤트에 관한 정보는 **events**에 기록됩니다 . **maxevents**는 epoll이벤트 풀의 크기고요. **timeout**는 기다리는 시간입니다. 0보다 작다면 이벤트가 발생할 때까지 기다리고, 0이면 바로 리턴, 0보다 크면 **timeout** 밀리세컨드 만큼 기다린다. 만약 timeout시간에 이벤트가 발생하지 않는다면 0을 반환합니다.

이벤트가 발생했다면 발생한 이벤트의 갯수를 반환합니다.

4. epoll\_ctl()

이벤트풀을 제어하기 위해서 사용한다. poll(2)와 매우 비슷하게 작동한다. **op**는 **fd**에 대해서 어떤 작업을 할것인지를 정의하기 위해서 사용된다. **op**가 실행된 결과는 **event**구조체에 적용된다

5. Struct epoll\_event

**epoll\_data\_t**를 유심히 볼필요가 있다. union이라서 사용자 정의 데이터에 대한 포인터를 넘길 수가 있습니다. 예를들어 여기에 pid값이라든지 소켓지정번호 혹은 메시지를 포함한 정보를 구조체로 넘겨줄 수 있는 거죠.

6. Op

EPOLL\_CTL\_ADD

**fd**를 epoll 이벤트 풀에 추가하기위해서 사용한다.

EPOLL\_CTL\_DEL

**fd**를 epoll 이벤트 풀에서 제거하기 위해서 사용한다.

EPOLL\_CTL\_MOD

이미 이벤트 풀에 들어 있는 **fd**에 대해서 **event**의 멤버값을 변경하기 위해서 사용한다.

7. events

EPOLLIN 0x1

입력(read)이벤트에 대해서 검사한다.

EPOLLOUT 0x4

출력(write)이벤트에 대해서 검사한다.

EPOLLERR 0x8

파일지정자에 에러가 발생했는지를 검사한다.

EPOLLHUP 0x10

Hang up이 발생했는지 검사한다.

EPOLLPRI 0x2

파일지정자에 중요한 데이터가 발생했는지 검사한다.

EPOLLET 0x80000000

파일지정자에 대해서 ET 행동을 설정한다. 기본 값은 LT.

[2.3.](https://www.joinc.co.kr/w/Site/Network_Programing/AdvancedComm/epoll24) 최대 등록 파일 개수

/proc/sys/fs/epoll/max\_user\_watches 에서 확인할 수 있습니다. 여기에는 유저가 등록할 수 있는 파일의 최대 개수가 명시되어 있는데요. User ID당 제한이 설정됩니다. 32bit 커널에서는 파일 하나당 약 90byte의 메모리가 64bit에서는 160바이트가 필요하다고 합니다.

[4.1.](https://www.joinc.co.kr/w/Site/Network_Programing/AdvancedComm/epoll24) 장점

좀더 적은 자원을 차지하면서 효율은 기존의 기술보다 좋다.

select, poll등에 비해 충분히 효율적이면서도 RealTime Signal에 비해서 사용하기 쉽다.

[4.2.](https://www.joinc.co.kr/w/Site/Network_Programing/AdvancedComm/epoll24) 단점

표준 지원사항이 아니라서 다른 유닉스에서는 사용할 수 없다.

1. kqueue()

kqueue() (은)는 신규의 커널 이벤트 큐를 생성해 기술자를 돌려줍니다. 큐는 [fork(2)](https://manned.org/fork.2) 그리고 생성된 child process에는 계승되지 않습니다. 그렇지만, RFFDG 플래그없이 [rfork(2)](https://manned.org/rfork.2) 하지만 불려 갔을 경우에는, 기술자 테이블이 공유되어 2 개(살)의 프로세스간에 kqueue 의 공유가 가능하게 됩니다.

retval :

kqueue() (은)는 신규의 커널 이벤트 큐를 생성해, 파일 기술자를 돌려줍니다. 커널 이벤트 큐의 생성시에 에러가 있었을 경우에는, 값 -1 이 돌려주어져 errno 가 세트 됩니다.

2. kevent()

kevent() (은)는, 큐에 이벤트를 등록해, 보류중의 모든 이벤트를 유저에게 돌려주기 위해서(때문에) 사용됩니다.

*changelist* 는 *kevent* 구조체의 배열에의 포인터입니다. 이 구조체는 ⟨*sys/event.h*⟩ 그리고 정의되고 있습니다. 보류중의 이벤트를 큐로부터 읽어들이기 전에, *changelist* 에 포함되어 있는 모든 변경을 적용합니다. *nchanges 는* *changelist* 의 크기를 줍니다.

*eventlist* 는 kevent 구조체의 배열에의 포인터입니다.

*nevents 는* *eventlist* 의 크기를 결정합니다. *T*

*imeout* 하지만 NULL 가 아닌 포인터의 경우에는, timespec 구조체이다고 해석되어, 이벤트를 기다리는 최대 대기 시간을 지정합니다. *timeout* 하지만 NULL 포인터의 경우에는, kevent() (은)는 무기한으로 기다립니다. 폴링의 효과를 얻기 위해서(때문에)는, *timeout* 인수에, 0 을 나타낸다 *timespec* 구조체를 가리키는 비 NULL 의 포인터를 주어야 합니다. *changelist* (와)과 *eventlist* 용으로 같은 배열을 사용할 수가 있습니다.

retval :

*eventlist* 에 배열되고 있는 이벤트의 수를 돌려줍니다. 이 수는, 최대 *nevents* 그리고 주어진 값까지입니다. *changelist* 의 요소의 처리중에 에러가 발생해, 한편 *eventlist* 에 충분한 여지가 있는 경우에는, *flags* 에 EV\_ERROR 하지만 세트 되어 *data* 에 시스템 우류가 세트 된 이벤트가, *eventlist* 에 놓여집니다. 아주 없으면, -1 하지만 돌려주어 errno 하지만 에러 상태를 나타내기 위해서(때문에) 세트 됩니다. 마감 시간의 경우에는, kevent() (은)는 0 을 돌려줍니다.

3. Struct kevent

Ident : 이 이벤트를 식별하기 위해서 사용되는 값입니다. 엄밀한 해석은 연결시킬 수 있었던 필터에 의해 결정됩니다만, 보통은 파일 기술자로서 해석됩니다.

Filter : 이 이벤트를 처리하기 위해서 사용되는 커널 필터를 식별합니다. 미리 정의된 시스템 필터는 후술 되어 있습니다.

flags : 이벤트 발생시에 실행해야 할 액션입니다.

fflags : 필터 고유의 플래그입니다.

data : 필터 고유의 데이터의 값입니다.

udata : 변경되지 않고 커널을 통해 건네받는 불투명한 유저 정의의 값입니다.

4. Struct kevent.flags

EV\_ADD : 이벤트를 kqueue 에 추가합니다. 기존의 이벤트를 다시 추가하면(자), 원의 이벤트의 파라미터가 변경됩니다. 중복 하는 엔트리가 생기는 것은 아닙니다. 이벤트를 추가하면(자), EV\_DISABLE 플래그에 의해 덧쓰기되지 않는 한은 자동적으로 유효하게 됩니다.

EV\_ENABLE : 이벤트가 방아쇠 되었을 경우에, kevent() 하지만 그 이벤트를 돌려주는 것을 허가합니다.

EV\_DISABLE : 이벤트를 무효로 합니다. 이것에 의해 kevent() (은)는 그 이벤트를 돌려주지 않게 됩니다. 필터 자신은 무효로 되지 않습니다.

EV\_DELETE : kqueue 로부터 이벤트를 삭제합니다. 파일 기술자가 연결시킬 수 있고 있는 이벤트는, 그 기술자의 마지막 클로우즈시에 자동적으로 삭제됩니다.

EV\_ONESHOT : 필터가 최초 방아쇠 되었을 때에 마셔, 이벤트가 돌아가도록(듯이) 합니다. 유저가 이벤트를 kqueue 로부터 회수한 다음에, 그 이벤트는 삭제됩니다.

EV\_CLEAR : 유저가 이벤트를 회수한 후에, 그 상태를 리셋트 합니다. 이것은 현재 상태가 아니고, 상태의 변화를 보고하는 필터에 유용합니다. 몇개의 필터는 내부에서 이 플래그를 자동적으로 세트 하고 있다 일지도 모르는 것에 주의해 주세요.

EV\_EOF : 그 필터 고유의 EOF 상태인 것을 나타내기 위해서(때문에), 필터가 이 플래그를 세트 하는 일이 있습니다.

EV\_ERROR : 후술의 *반환값* (을)를 참조해 주세요.

EV\_SET() (은)는 kevent 구조체의 초기화를 간단하게 하는 매크로입니다.

-

-

-