리눅스 운영체제를 위한 POSIX 호환 네트워크 비동기 입출력의 구현 및 성능 평가

안백송*, 홍성흔**, 김강호*, 정성인*
*한국전자통신연구원 인터넷서버그룹
**㈜매크로임팩트
e-mail : bsahn@etri.re.kr

Implementation and Evaluation of POSIX Network Asynchronous I/O for Linux Operating System

Baik-song Ahn*, Seong-Heun Hong**, Kang-Ho Kim*, Sung-In Jung*
*Internet Server Group, Electronics and Telecommunications Research Institute
**MacroImpact, Inc.

요 약

고성능의 확장성 있는 대규모 네트워크 서버 구현시 입출력 기능 향상을 위한 방법 중 하나는 비동기 입출력 기능을 이용하는 것이다. 비동기 입출력은 기존의 poll() / select()와 같은 입출력 멀티플렉싱 기법의 불필요한 CPU 부하를 방지하고, 입출력 완료를 기다리면서 블록되지 않으므로 시스템 부하를 감소할 수 있다. 본 논문에서는 리눅스 운영체제를 위한 POSIX 표준 네트워크 비동기 입출력 기능을 설계 및 구현하였고, 실험을 통해 기존 메커니즘과의 성능 차이를 비교하였다.

1. 서론

웹 서비스, 네트워크 게임, 스트리밍 서비스 등과 같은 대규모 네트워크 서비스를 구축함에 있어서, 서버 측의 주된 설계 이슈는 단연 고성능의 확장성있는 서버를 설계하는 것이다. 이러한 대규모 서버는 수천, 수만에 이르는 동시 접속 사용자 수를 수용할 수 있어야 할 뿐 아니라, 짧은 시간에 순간적으로 접속이 집중되는 상황에도 성능 저하 없이 유연하게 대처할 수 있어야 한다.

고성능이면서 대용량의 처리를 하려면 패킷의 송수신 처리부와 데이터 처리부의 최적화가 핵심이며 이에 대한 여러 가지 구현 방법이 존재한다. TCP/IP 기반의 네트워크 환경에서 일반적으로 사용되는 BSD socket 의 고전적인 입출력 방식은 다수의 소켓 접속을 select/poll 등의 시스템 호출을 사용하여 multiplexing(혹은 polling)하는 방식이다. 이 방법은 낱말 그대로 운영체제 커널에서 처리되는 네트워크 입출력 상태를 주기적인 polling 방식에 의해 감시하는 것으로, 오래된 만큼 안정적이기는 하나, 고성능 대용량 서버에서는 불필요한 CPU 부하를 유발함으로써 서버의 성능 관점에서 분명한 한계가 존재한다. 이러한 한계를 극복하기 위하여 (특히 네트워크) 입출력 방식에 대한 개선 또는 대안이 나와있으며, 그 중의 하나가 asynchronous I/O(AIO)이다. Ben LaHaise 의 AIO 는 현재 리눅스 커널 2.6.6 이후에 기본 구조가 포함되어 있고, 추가 기능들이 패치 파일 형태로 존재한다. 그중에 Suparna Bhattacharya 가 제작한 리눅스 AIO 용 패치는 기존의 리눅스 AIO 에서 제공하지 않는 파일 시스템 버퍼 입출력에 대한 비동기 입출력 기능을 포함한 부가적인 기능을 추가하며, 전반적으로 코드의 분석 및 수정이 용이하도록 내부 구조가 수정되었다.

본 논문에서는 리눅스 커널 2.6.7 에 포함되어 있는 Ben LaHaise AIO 에 Suparna 의 패치를 가한 버전을 기반 코드로 선정하고 그 위에 네트워크 AIO 를 구현하였다. 현재 구현된 Ben LaHaise 의 AIO 는 파일에 대해서만 AIO 를 지원하고 네트워크에 대해서는 AIO 를 지원하지 않기 때문에 네트워크 AIO 를 위한 모듈을 개발하였다. 또한 libaio 라는 라이브러리를 통하여 사용자 수준의 AIO 인터페이스를 제공하며, 최종적으로

사용자에게 POSIX AIO 인터페이스를 제공할 수 있도록 libaio 위에 POSIX AIO 인터페이스를 구현하였다.

2. POSIX 호환 Network AIO 의 기능

Network Asynchronous I/O 에서는 POSIX 호환 인터페이스를 통해 aio_read(), aio_write(), aio_cancel(), aio_return(), aio_suspend() 등과 같은 AIO 표준 인터페이스를 제공한다. 기본 기능은 소켓을 통한 기본적인 데이터 입출력과 관련된 기능으로, 소켓을 통한 데이터 read/write, 작업 취소, 리턴값 확인 등의 기능을 제공한다.

POSIX Asynchronous I/O 인터페이스는 입출력 작업에 필요한 각종 정보를 aiocb 구조체 하나에 넣어 그 구조체의 주소값을 파라메터로 넘겨서 함수를 호출한다. aiocb 구조체는 다음과 같은 속성값을 갖는다.

```
struct aiocb {
     int             aio_filedes;
     off_t           aio_offset;
     void           *aio_buf;
     size_t          aio_nbytes;
     structsigevent  aio_sigevent;
     int             aio_lio_opcode;
     int             aio_reqprio;
};
```

- aio_fildes : 네트워크 소켓 파일기술자
- aio_buf : 사용자 수준 메모리 버퍼
- aio_nbytes : 입출력할 데이터의 크기
- aio_sigevent : 실제 입출력 완료 후 통지를 받을 signal 관련 정보
- aio_lio_opcode : 입출력 작업 종류(read or write)

aio_read(struct aiocb *aiocbp)

소켓을 통한 데이터 수신 기능을 요청하는 함수이며, 작업요청 후 blocking 없이 바로 리턴된다.

aio_write(struct aiocb *aiocbp)

소켓을 통한 데이터 전송 기능을 요청하는 함수이다. 작업요청 후 blocking 없이 바로 리턴된다.

aio_cancel(int fildes, struct aiocb *aiocbp)

요청한 입출력 작업을 취소하는 함수이다.

aio_error(struct aiocb *aiocbp)

aiocbp 가 가리키는 aiocb 구조체와 관련된 입출력 작업의 오류 상황을 통보하는 함수이다.

aio_return(struct aiocb *aiocbp)

aiocbp 가 가리키는 aiocb 구조체에 해당하는 입출력 작업의 완료 후 리턴값을 확인하는 함수이다.

aio_suspend(struct aiocb *lacb[], int nent, const struct timespec *timeout)

lacb 포인터가 가리키고 있는 리스트의 처음부터 nent 개의 aiocb 구조체에 해당하는 입출력 작업들을 기다린다.

3. POSIX 호환 Network AIO 의 구조

Network Asynchronous I/O 의 구조는 크게 다음과 같은 세 계층으로 구분되어진다.

- 커널 수준 Network Asynchronous I/O 메커니즘
- libaio 사용자 수준 라이브러리
- POSIX Asynchronous I/O 인터페이스를 제공하는 사용자 수준 라이브러리

Linux Native Asynchronous I/O 메커니즘은 리눅스 커널 2.6 에서 제공하는 AIO 기능을 네트워크 소켓으로 확장시킨 것이며, libaio 사용자 수준 라이브러리는 커널의 Native AIO 메커니즘을 사용자 수준에서 사용할 수 있도록 하는 간단한 라이브러리이다. POSIX Asynchronous I/O 사용자 수준 라이브러리는 Linux Native AIO 메커니즘 상위 수준에서 사용자에게 최종적으로 보여지는 aio_read()/aio_write()와 같은 POSIX 호환 AIO 관련 API 를 제공하는 기능을 담당한다.

4. 커널 수준 Netowrk AIO 구조

커널 2.6 에 구현된 리눅스 Network AIO 작업은 [**오류! 참조 원본을 찾을 수 없습니다.**]에 나타난 것처럼 기본적으로 kioctx 라 불리는 입출력 컨텍스트 구조체와 kiocb 라 불리는 다수의 입출력 컨트롤 블록을 이용하여 진행된다. kioctx 에는 프로세스별로 유지해야 하는 각종 메타정보들이 저장되어 있으며, kiocb 에는 각각의 입출력 작업들에 관련된 정보(사용자 버퍼 주소, 파일 기술자, 입출력 작업 종류 등)들이 저장된다. 또한 kioctx 구조체마다 ring buffer 가 하나씩 할당되어 작업이 완료된 입출력 작업들에 대한 event notification 에 이용된다.

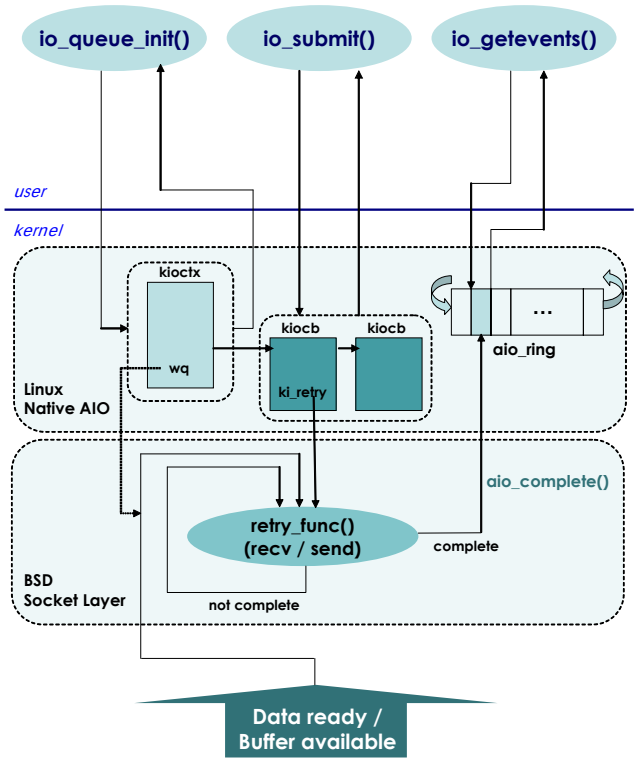
커널 수준의 비동기 입출력 기능은 다음과 같이 크게 3 단계로 진행된다.

- 입출력 컨텍스트 구조체 할당 및 초기화
- 입출력 컨트롤 블록 할당 및 등록 작업
- 완료된 입출력 작업들에 대한 event catching

4.1 입출력 컨텍스트 구조체 할당 및 초기화

입출력 작업을 수행하기 이전에 커널 내부에 입출력 컨텍스트 구조체(kioctx)를 할당 및 등록하는 작업

이 필요하다. 커널 내부에 kioctx 구조체를 위한 메모리 할당 및 구조체의 member 들에 대한 초기화 작업이 수행된 후 커널 자료구조인 task_struct 의 mm 필드에 등록된다.



[그림 1] 커널 2.6 의 리눅스 AIO

4.2 입출력 컨트롤 블록 할당 및 등록 작업

입출력 컨텍스트 구조체 초기화 작업이 끝난 후 실제 비동기 입출력 작업 수행을 위해 io_submit() 시스템 호출을 수행하여 kiocb 구조체를 할당 및 설정한 후 kioctx 구조체에 등록한다. io_submit() 시스템 호출 시 실제 입출력 작업을 위해 kiocb 구조체에 등록된 retry 함수가 호출되며, block 되지 않고 입출력 작업을 완료 가능한 경우 그 즉시 aio_complete()를 호출하여 완료하고, 그렇지 않고 blocking 되어 기다려야 하는 경우는 해당하는 소켓 구조체의 wait queue(sk_sleep)에 kiocb 를 등록하고 바로 리턴한다. 입출력 완료 이벤트가 디바이스로부터 발생하면 wait queue 에서 깨어난 kiocb 는 그 자신을 wait queue 에서 분리시켜서 다시 해당 kioctx 의 run_list 에 추가시킨다. 이후 workqueue 커널 스레드가 스케줄링 되면 kiocb 에 등록된 retry 함수를 다시 수행하게 된다.

프로세스 또는 workqueue 가 입출력 작업을 완료한 후 사용자 프로세스에게 알려주기 위해 aio_complete() 함수를 호출한다. 이 함수의 주된 역할은 kioctx 구조체에 등록되어 있는 ring buffer 에 완료된 입출력 작업의 개수만큼 io_event 구조체를 추가하는 것이다.

4.3 완료된 입출력 작업들에 대한 event catching

io_getevents() 시스템 호출을 통해 요청한 입출력 작업들에 대한 완료 여부 및 그 결과에 관한 정보를 알 수 있다.

4.4 retry 함수의 구현

리눅스 커널에 AIO 기능을 구현하기 위해서 개발자가 해야 하는 주요한 일은 입출력 요청 구조체(kiocb)에 등록되는 retry 함수를 구현하는 것이다. 네트워크 AIO 기능을 추가하기 위해 소켓 수신 및 송신 기능에 해당하는 retry 함수인 aio_precv() / aio_psend()를 각각 추가하였다.

aio_precv()

네트워크 AIO 의 소켓 recv 기능을 위해 구현된 retry 함수는 aio_precv() 함수이다. aio_precv() 함수는 먼저 nonblocking 방식으로 BSD 소켓구조체의 recvmsg 메소드를 호출한 후, 그 리턴값을 체크한다. 리턴값이 유효한 양수이면 이를 그대로 리턴하며 함수를 종료한 후 aio_complete()를 호출하여 입출력 이벤트 구조체를 ring buffer 에 등록한 후 작업을 종료한다. 리턴값이 –EAGAIN 일 경우 nonblocking 방식으로 즉시 작업을 완료할 수 없는 상황이므로, kiocb 구조체를 sk_sleep 에 추가한 후 리턴하면서 함수를 종료한다.

sk_sleep 에 추가된 kiocb 구조체는 요청한 패킷이 도착할 때 sk_sleep 에서 깨어나게 되며, 이때 다시 retry 함수인 aio_precv()함수가 호출되면서 작업을 진행하게 된다. 이러한 방식으로 작업이 완료될 때까지 sk_sleep 에 매달렸다가 깨어나면서 반복적으로 retry 함수를 호출하게 된다.

aio_psend()

네트워크 AIO 의 소켓 send 기능을 위해 구현된 retry 함수는 aio_psend() 함수로, 전반적인 구조는 aio_precv() 함수와 유사하다.

aio_psend() 함수는 aio_precv()와 마찬가지로 BSD 소켓구조체의 sendmsg 메소드를 호출한 후 그 리턴값이 양수이면 aio_complete()를 호출하여 작업을 종료하고, 그렇지 않은 경우 소켓의 sk_sleep 에 추가한 후 향후 깨어날 때 다시 aio_psend() 함수를 호출하여 작업을 진행하게 된다.

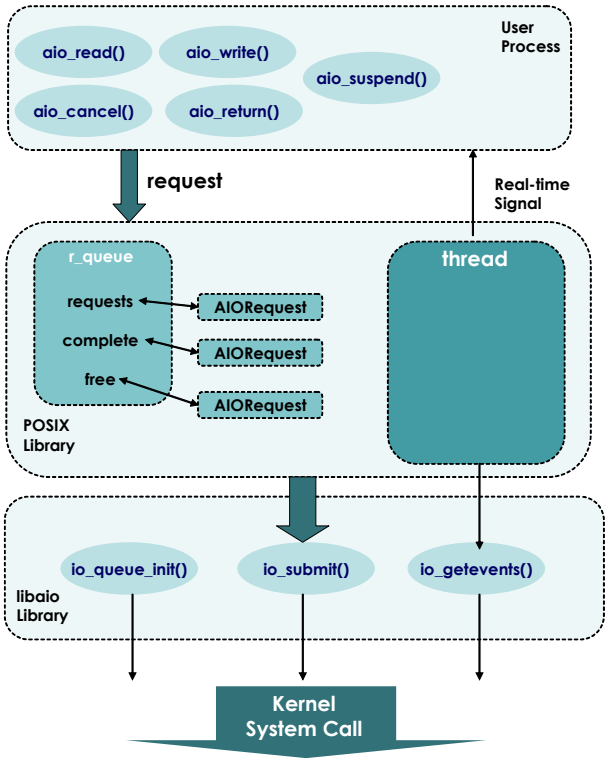
5. Network AIO 를 위한 사용자 수준 라이브러리

커널 수준 Network AIO 기능의 상위에서 작동하는 사용자 수준 Network AIO 라이브러리는 크게 libaio 라이브러리와 POSIX 라이브러리의 두 계층으로 구성되어있다. [그림 2]는 사용자 수준 Network AIO 라이브러리의 구조를 나타낸 것이다.

libaio 라이브러리는 시스템 호출을 통해 커널 내의 AIO 메커니즘을 사용하기 위한 간단한 사용자 수준

라이브러리로, 커널 AIO 기능을 사용하기 위해 io_queue_init(), io_submit(), io_getevents() 등의 시스템 호출을 담당한다.

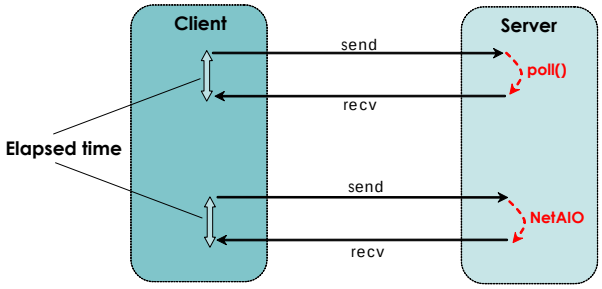
POSIX 인터페이스를 제공하는 사용자 수준 라이브러리는 사용자의 AIO 요청에 대한 요청을 시스템콜로 변환하고, 그 명령이 완료될 때까지 그것들을 관리하는, 비동기입출력들의 버퍼역할을 하는 사용자라이브러리이다. 비동기입출력의 단일명령에 해당하는 구조체의 이름은 struct AIORequest 이고, 이를 리스트형태로 엮어서, 그 각각의 구성원을 분류(진행중(requests)인것과 완료된 것(complete), 그리고 재사용가능(free))한다. 또한 POSIX 라이브러리는 별도의 쓰레드를 생성하여 주기적으로 io_getevents() 시스템 호출을 통해 완료된 입출력 작업이 있는지를 검색하고, 만약 있을 경우 Real-time Signal 을 이용하여 사용자 프로세스에게 통보한다.



[그림 2] 사용자 수준 Network AIO 라이브러리 구조

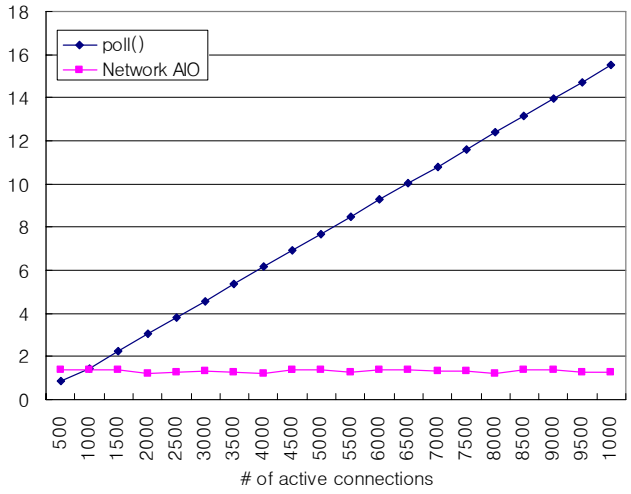
6. 성능 비교 실험

[그림 3]은 POSIX Network AIO 의 성능 비교 실험을 나타낸 것이다. 클라이언트와 서버는 Gigabit Ethernet 으로 연결되어 있으며 서버는 기존의 poll()과 POSIX Network AIO 두 버전을 구현하여 클라이언트로부터의 요청을 처리하도록 하였다. Gigabit Ethernet 으로 연결된 클라이언트와 서버간에 10000 개의 connection 을 맺은 후 그 중 일부 connection 에 대하여 클라이언트는 짧은 메시지를 서버에 전송하고 서버는 이를 수신한 후 동일한 메시지를 다시 클라이언트에 보낸다. 이때 소요된 round trip time 을 계산하여 여러 번 수행했을 때의 평균값을 계산한다.



[그림 3] POSIX Network AIO 의 성능 비교 실험

[그림 4]는 클라이언트와 서버가 메시지를 주고 받는 active connection 의 개수를 증가했을 때의 poll()과 Network AIO 의 평균 round trip time 을 정리한 결과이다. poll()의 경우 active connection 의 수가 늘어날수록 round-trip time 역시 linear 하게 증가하는 반면, Network AIO 를 사용한 경우 성능 감소가 거의 없음을 알 수 있다.



[그림 4] poll()과 Network AIO 의 성능 비교 결과

7. 결론

본 논문에서는 리눅스 운영체제를 위해 구현된 POSIX 호환 Network Asynchronous I/O 의 구조를 설명하고 실험을 통해 기존의 I/O multiplexing 기법과의 성능을 비교, 분석하였다. 향후 구현코드의 성숙도 향상 및 웹서버 벤치마크 도구와 같이 보다 현실적인 workload 상에서의 실험 및 분석 작업이 필요하다.

참고문헌

[1] Linux Kernel AIO Support Homepage, http://lse.sourceforge.net/io/aio.html
[2] Suparna Bhattacharya's AIO Patch Homepage, http://www.kernel.org/pub/linux/kernel/people/aio/
[3] Bill O. Gallmeister, POSIX.4 : Programming For The Real World, O'Reilly & Associates, Inc., 1994
[4] Daniel P. Bovet et al. Understanding The Linux Kernel (2nd Edition), O'Reilly & Associates, Inc., 2003