리눅스 운영체제를 위한 확장성있는 네트워크 비동기 입출력 메커니즘

안백송[O] 김강호 정성인
한국전자통신연구원 인터넷서버그룹
{bsahn[O], khk, sijung}@etri.re.kr

Scalable Network Asynchronous I/O Mechanism for Linux Operating System

Baik-song Ahn[O], Kang-Ho Kim, Sung-In Jung
Internet Server Group, Electronics and Telecommunications Research Institute

요 약

고품질의 확장성 있는 서비스를 제공함으로써 다수의 사용자의 요청을 처리하고 시스템을 견고하게 유지할 수 있는 인터넷 서버를 구현하기 위한 한 가지 방법은 비동기 입출력 기능(AIO)을 이용하는 것이다. 기존의 고전적인 입출력 방식인 poll() / select()에 비해 AIO는 CPU 부하를 비롯한 시스템 자원의 낭비를 줄일 수 있으며, 입출력 완료를 기다리면서 블록되지 않으므로 시스템 부하를 감소할 수 있다. 본 논문에서는 리눅스 커널에 구현되어 있는 파일 기반 AIO 기능을 네트워크 소켓 상에서 동작할 수 있도록 확장 구현하였으며, 그 구조에 대하여 자세히 설명하였다. 또한 클라이언트-서버 구조를 모델링한 실험을 통해 기존 메커니즘과의 성능 차이를 비교하였다.

1. 서 론

온라인 게임, 고화질 스트리밍 서비스 등을 위한 인터넷 서버는 다수의 네트워크 연결을 병목 없이 처리하여 고품질의 확장성 있는 서비스를 제공하는 것이 가장 큰 요구 조건이라 할 수 있다. 수만 명의 사용자의 작업 요청을 수용하여 처리할 수 있어야 할 뿐 아니라 순간적인 과부하에도 견고하게 시스템을 유지할 수 있어야 한다.

대규모 네트워크 서비스의 성능 향상을 위해 패킷 송수신 및 데이터 처리에 대한 최적화가 핵심이며 이에 대한 여러 가지 구현 방법이 존재한다. 일반적으로 사용되는 TCP/IP 기반 네트워크의 송수신 메커니즘으로 BSD socket의 고전적인 입출력 방식인 I/O multiplexing 기법이 있다. BSD socket의 태동과 함께 하나 이상의 소켓 접속을 동시 처리하는 서버에서 사용된 I/O multiplexing 기법은 select()와 poll() 시스템 콜을 통한 polling 방식으로, 이 방법은 지금까지도 Unix 서버 프로그램의 대부분에서 사용되는 전통적인 방식 중의 하나이다. 최근에는 쓰레드 패키지의 사용과 함께 polling 방식이 CPU 부하를 비롯한 시스템 자원의 낭비를 초래하고 있어 개선점을 찾으려는 시도가 많아졌다. 최근에 들어서 리눅스에서도 다양한 개선을 통해 새로운 입출력 처리 방식을 지원하고 있으며, 그 중의 하나가 asynchronous I/O(AIO)이다.

AIO 방식은 기본적으로 각 접속과 관련된 입출력을 하나의 쓰레드가 처리하는 것을 가정한다. 네트워크 송, 수신을 요청한 응용 서버의 쓰레드는 네트워크 입출력이 완료될 때까지 blocking 되거나, select 를 사용해서 완료 상태를 검사하는 방식이 아니라, 입출력 요청 직후 완료 여부와 관련 없이 즉시 다른 작업을 진행하는 비동기적인 방식이다. 커널에 의해 처리된 네트워크 입출력의 결과는 추후 시그널 등의 비동기적인 event notification 방식을 통해 응용 프로그램에게 전달된다. AIO 의 특징은 응용 서버가 입출력 도중에 blocking 될 필요가 없이 즉시 I/O 외의 다른 작업을 수행할 수 있으며, 입출력 상태를 파악하는 poll/select 등이 필요없어 I/O와 관련된 CPU 부하를 크게 줄일 수 있다는 점이다. 이러한 성능상의 특징으로 인해 AIO를 사용함으로써 동시 사용자 수를 증가시킬 수 있다.

리눅스 커널 2.6.5 이후 버전부터 기본적으로 Ben LaHaise의 AIO를 포함하고 있으나, 현재까지 파일 입출력 부분만을 지원할 뿐 네트워크 소켓 입출력 부분은 지원되지 않고 있다. 본 논문에서는 기존의 리눅스 커널에 포함된 파일 기반 AIO 기능을 확장하여 네트워크 소켓에 대해서도 동작하도록 구현하였으며, 사용자 수준 라이브러리를 제공함으로써 커널의 AIO 기능을 시스템 호출 인터페이스를 통하여 사용자가 이용할 수 있도록 구현하였다.

2. 소켓 호환 네트워크 AIO의 구조

네트워크 AIO는 다음과 같은 두 계층으로 구분되어진다.

- 커널 수준의 core AIO 메커니즘
- 시스템 호출 관련 사용자 수준 라이브러리

커널 수준의 core AIO 메커니즘은 리눅스 커널 2.6에서 제공하는 Ben LaHaise의 AIO 기능을 네트워크 소켓으로 확장시킨 것이며, 시스템 호출 관련 사용자 수준 라이브러리는 커널의 core AIO 메커니즘을 사용할 수 있도록 시스템 호출 인터페이스를 사용자에게 제공하는 라이브러리이다.

3. 커널 수준 core AIO의 구조

[그림 1]에서 설명되는 커널 수준 core AIO 메커니즘은 다음과 같이 크게 3단계로 진행된다.

- kioctx 구조체 할당 및 초기화
- kiocb 구조체 할당 및 등록 작업
- 완료된 입출력 작업들에 대한 event catching

3.1 kioctx 구조체 할당 및 초기화

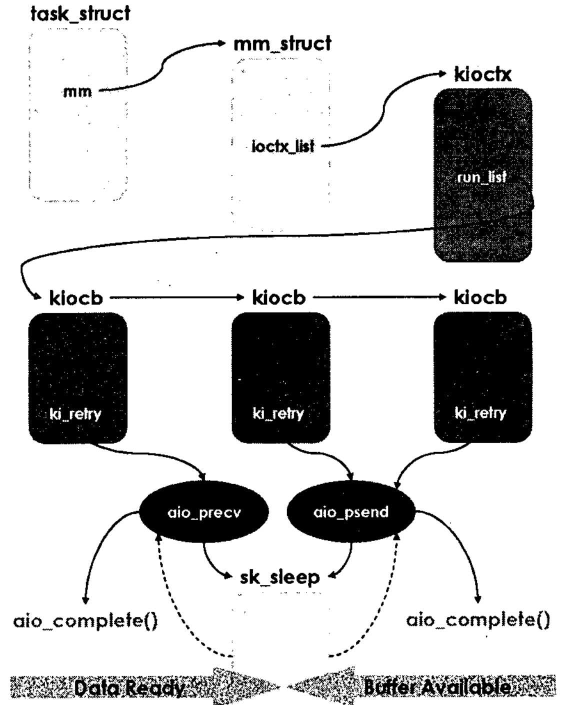
kioctx 구조체는 프로세스 별로 유지해야 하는 메타데이터들이 저장되는 구조체이다. task_struct 구조체와 연결된 mm_struct 구조체의 ioctx_list에 연결되며, 실제 입출력 작업을 담당하는 kiocb 구조체를 관리하는 역할을 담당한다. kioctx 구조체의 run_list에는 입출력 작업 수행을 기다리는 kiocb 구조체들이 등록된다.

사용자는 io_setup() 시스템 호출을 통해 커널 내부에 kioctx 구조체를 할당, 등록한다. 이는 AIO 기능을 이용한 입출력 작업 이전에 반드시 수행되어야 한다.

3.2 kiocb 구조체 할당 및 등록 작업

사용자는 io_submit() 시스템 호출을 통해 kiocb 구조체를 할당, 초기화한 후 커널 내에서 retry 함수를 호출하여 실제 입출력 작업을 수행한다. io_submit() 함수는 입출력 작업의 완료 여부와 관계없이 block되지 않고 즉시 리턴되는 함수이다. kiocb 구조체는 실제 입출력 작업별로 할당되는 구조체로서 소켓 디스크립터, 데이터의 메모리 주소 및 크기, 요청작업 종류 등의 메타데이터가 저장된다.

kiocb 구조체에는 retry 함수가 등록되어 실제 데이터의 입출력 작업을 수행하게 된다. retry 함수는 소켓 recv(aio_precv), 소켓 send(aio_psend) 작업을 위해 각각 구현되어 있다. retry함수 호출 후 즉시 완료될 수 없는 작업일 경우 해당 소켓 구조체의 sk_sleep wait



[그림 1] 커널 수준 core AIO의 구조

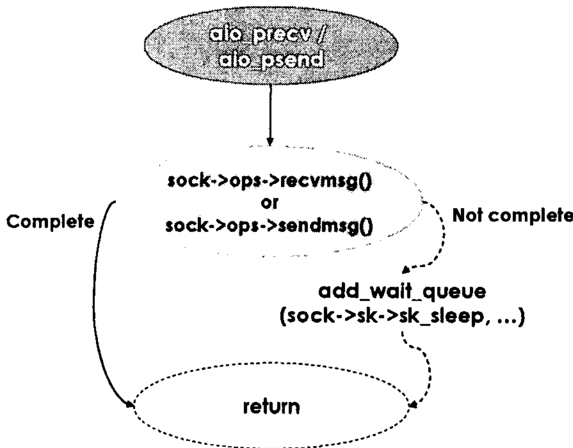
queue에 kiocb 구조체를 등록한다. 후에 입출력 작업이 완료될 수 있는 상황에 sk_sleep으로부터 깨어나 다시 retry함수를 호출하여 작업이 완료되면 aio_complete() 함수를 호출한다.

3.3 retry 함수의 구조

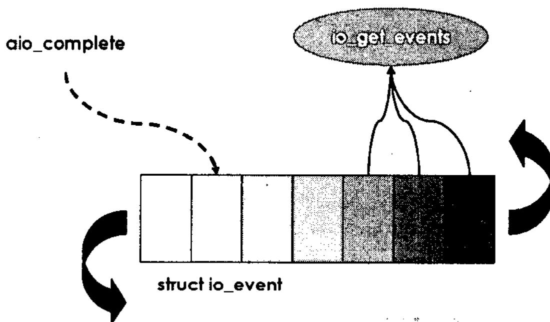
실제 입출력 작업을 담당하는 함수인 retry 함수는 먼저 nonblocking 방식으로 해당 입출력 작업에 해당하는 INET 계층 메소드를 호출한다. 요청한 입출력 작업이 recv일 경우 recvmsg(), send일 경우 sendmsg() 메소드가 각각 호출된다. 메소드의 리턴값을 체크하여 유효한 양수일 경우 바로 리턴하여 작업을 완료하며, EAGAIN일 경우 sk_sleep wait queue에 kiocb 구조체를 추가하여 차후에 작업이 완료될 수 있는 상황을 기다린다. [그림 2]는 retry 함수의 구조를 나타낸 것이다.

3.4 완료된 입출력 작업들에 대한 event catching

retry 함수 호출 후 완료된 작업들은 aio_complete() 함수 호출을 통해 kioctx 구조체에 등록된 ring buffer에 작업이 종료되었음을 알리는 io_event 구조체를 등록한다.



[그림 2] retry 함수의 구조



[그림 3] 완료된 입출력 작업 처리

사용자는 io_get_events() 시스템 호출을 통해 ring buffer를 뒤져 io_event 구조체를 검색함으로써 완료된 입출력 작업이 있는지를 찾는다. [그림 3]은 완료된 입출력 작업 처리 과정을 그림으로 나타낸 것이다.

4. 성능 비교

클라이언트와 서버는 Gigabit Ethernet으로 연결되어 있으며 서버는 기존의 poll()과 AIO 두 버전을 구현하여 클라이언트로부터의 요청을 처리하도록 하였다. Gigabit Ethernet으로 연결된 클라이언트와 서버간에 10000개의 connection을 맺은 후 그 중 일부 connection에 대하여 클라이언트는 짧은 메시지를 서버에 전송하고 서버는 이를 수신한 후 동일한 메시지를 다시 클라이언트에 보낸다. 이때 소요된 round trip time을 계산하여 평균값을 계산하였다.

[표 1]은 클라이언트와 서버가 메시지를 주고 받는 active connection의 개수를 증가했을 때의 poll()과 AIO의 평균 round trip time을 정리한 결과이다. poll()의 경우 active connection의 수가 늘어날수록 round-trip time 역시 linear하게 증가하는 반면, AIO를 사용한 경우 성능 감소가 거의 없음을 알 수 있다.

[표 1] poll/AIO 성능 비교 결과 (단위 : ms)

Active connections	poll()	AIO
500	0.862	**1.384**
1000	1.459	**1.369**
1500	2.256	**1.384**
2000	3.031	**1.222**
2500	3.806	**1.289**
3000	4.572	**1.341**
3500	5.357	**1.247**
4000	6.149	**1.226**
4500	6.923	**1.396**
5000	7.701	**1.388**
5500	8.483	**1.28**
6000	9.263	**1.409**
6500	10.055	**1.411**
7000	10.808	**1.318**
7500	11.596	**1.317**
8000	12.381	**1.224**
8500	13.157	**1.38**
9000	13.982	**1.369**
9500	14.736	**1.292**
10000	15.532	**1.287**

5. 결론

본 논문에서는 네트워크 소켓 상에서 동작하도록 확장된 리눅스 AIO 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