Zero-copy 기반 파일 전송 메커니즘 성능 비교 및 분석

서준호, 김재석, 윤건우, 김승혁,
밀라티 프라티위, 한진호, 최윤호
부산대학교

tjwnsgh1110@gmail.com, tjr3117@gmail.com, glove2275@gmail.com,
senghyuk66@gmail.com, pratiwimillati@gmail.com, csrc@pusan.ac.kr, yhchoi@pusan.ac.kr

Comparison and analysis of Zero-copy based file transfer mechanisms

Joonho Seo, Jae-Seok Kim, Gun-Wo Yoon, Seung-Hyuk Kim,
Millati Pratiwi, Han Jinho, Yoon-Ho Choi
Pusan National Univ.

요 약

클라우드 컴퓨팅, 빅데이터, 스트리밍 서비스 같은 대용량 데이터 전송이 필요한 시스템에서는 데이터 전송 과정에서 발생하는 데이터 복사에 의한 서버 자원 소모와 성능 저하 문제가 발생할 수 있다. 이에 본 논문에서는 리눅스 기반 시스템에서 네트워크 소켓 프로그래밍을 위해 사용되는 read/write 시스템 콜과 Zero-copy 기반의 sendfile 시스템 콜의 메커니즘을 분석하고 성능을 비교한다. 이를 통해 sendfile 이 대용량 데이터 전송에 대해 더 효율적임을 보여주며, 네트워크 소켓 프로그래밍에서 어떤 파일 전송 메커니즘을 선택해야 하는지에 대해 중요한 지침을 제공한다.

Ⅰ. 서 론

현대의 IT 기술에서 데이터 전송은 매우 빈번히 발생한다. 클라우드 컴퓨팅, 빅데이터, 스트리밍 서비스 등 대규모 데이터를 다루는 환경에서는 지속적으로 방대한 양의 데이터가 서버에서 네트워크로 전송된다. 이때, 서버에서는 파일 시스템의 데이터를 네트워크 소켓으로 복사하는 과정을 거친다. 이러한 복사 과정은 서버 내에서 CPU 와 같은 자원을 상당량 소모하는 성능 저하 이슈를 발생시키기 때문에, 효율적인 데이터 전송을 위한 기술적인 개선이 요구된다[1].

네트워크 소켓에 데이터를 복사하는 소켓 프로그래밍은 전통적으로 read/write 시스템 콜을 사용하여 구현되어왔다. 하지만, 이러한 구현 방식은 사용자 공간과 커널 공간 간의 데이터 복사로 인해 CPU 오버헤드를 발생시키기 때문에. 이를 해결하기 위해 sendfile 과 같은 Zero-copy 기술[2]을 활용한 시스템 콜이 등장하였으며, 많이 활용되어 왔다[3]. 본 논문에서는 최신 리눅스 버전 기반의 read/write 및 sendfile 로 구현된 네트워크 소켓 통신 프로그램의 작동 과정을 비교하고, 각각의 성능을 분석하였다. 또한, 이를 바탕으로 소켓 프로그래밍에서의 시스템 콜 선택 기준을 제시한다.

Ⅱ. 본 론

본 장에서는 read/write 및 sendfile 데이터 전송 메커니즘에 대해 분석하고, 리눅스 함수 트레이싱 툴인 uftrace[4]를 사용하여 전송 성능을 비교한다.

Ⅱ-I. 데이터 전송 메커니즘 분석

데이터 통신을 위한 read/write 기반의 전통적인 소켓 프로그램은 다음과 같이 동작한다. 먼저, read

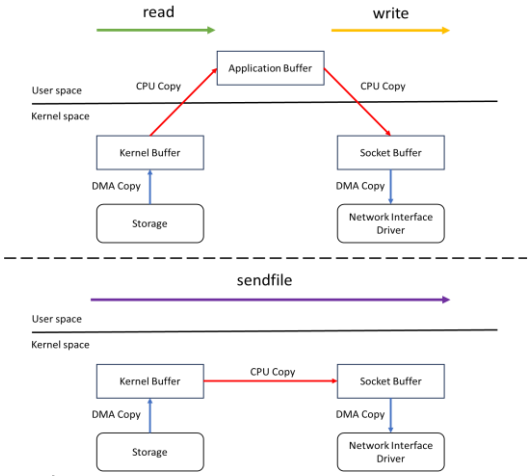

그림 1 read/write 및 sendfile 동작 과정 비교

호출 시 사용자 모드에서 커널 모드로 context switch 가 발생한다. 이후, 파일 시스템은 DMA(Direct Memory Access)를 통해 스토리지에서 커널 버퍼로 CPU 의 개입 없이 데이터 복사(DMA Copy)를 수행한다. 데이터는 다시 파일 시스템에 의해 커널 버퍼에서 유저 버퍼로 복사되고 (CPU Copy) 커널 모드에서 유저 모드로 context switch 가 발생한다. 이후, 유저 버퍼 내 데이터를 소켓 버퍼로 복사하기 위해 write 를 호출한다. 이 때, 유저 모드에서 커널 모드로 context switch 가 다시 발생하고 유저 버퍼에서 소켓 버퍼로의 CPU Copy 가 발생한다. 이후, 네트워크 드라이버는 비동기적으로 DMA Copy 를 통해 소켓 버퍼에서 네트워크로 데이터를 전송한다.
sendfile 은 CPU Copy 에 대한 오버헤드를 최소화할 수 있다. 초기 sendfile(Linux 2.6.1 이전 버전)은

read/write 와 달리 커널 버퍼에 복사된 데이터를 유저 버퍼가 아닌 소켓 버퍼로 직접 복사하여, CPU Copy 를 최소화하였으며, 이는 그림 1 에서 볼 수 있다. 현재 sendfile(Linux 2.6.1 이후)은 커널 버퍼의 데이터를 pipe 자료 구조를 통해 소켓 버퍼로 전달하는 splice 시스템 콜 기반으로 구현되어, 모든 CPU Copy 과정이 제거되었다.[5]

Ⅱ-Ⅱ. 데이터 전송 메커니즘 성능 비교

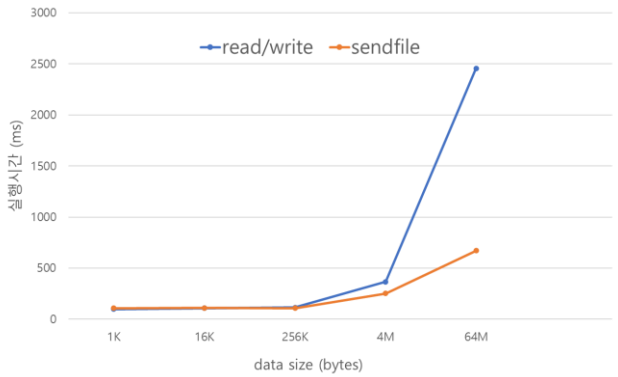

그림 2 파일 크기 별 실행 시간 비교 그래프

```
# process information
# ===================
# number of tasks     : 1
# task list           : 5411(rw)
# exe image           : /home/                    /rw/rw
# build id            : 01dddd24727ed523b65a944fe5d9a17ea1d9e1b4
# pattern             : regex
# exit status         : exited with code: 0
# elapsed time        : 3.198704240 sec
# cpu time            : 2.456 / 0.011 sec (sys / user)
# context switch      : 43 / 8031 (voluntary / involuntary)
# max rss             : 13568 KB
# page fault          : 47 / 1138 (major / minor)
# disk iops           : 10096 / 16 (read / write)
```

```
# process information
# ===================
# number of tasks     : 1
# task list           : 5200(sendfile)
# exe image           : /home/                    /sendfile/sendfile
# build id            : ab529e7fedf56ccfea370330e09c018277be9882
# pattern             : regex
# exit status         : exited with code: 0
# elapsed time        : 1.851168347 sec
# cpu time            : 0.672 / 0.000 sec (sys / user)
# context switch      : 271 / 891 (voluntary / involuntary)
# max rss             : 13312 KB
# page fault          : 47 / 1078 (major / minor)
# disk iops           : 10096 / 16 (read / write)
```

그림 3 read/write 와 sendfile 의 실행 시간 측정 결과 (64MB 기준)

그림 2 는 크기가 다른 5 개의 파일(1K, 16K, 256K, 4M, 64M) 전송에 대해 read/write 와 sendfile 의 실행 시간을 비교하는 그래프를 보여준다. 그림 2 를 통해서, 두 시스템 콜은 작은 크기의 파일(256kb 이하)을 전송할 때에 실행 시간 차이가 적은 것을 확인할 수 있다. 하지만 파일의 크기가 증가할 수록 두 시스템 콜의 실행 시간 차이가 급격히 증가하는 모습을 보인다. 그림 3 은 64MB 데이터 전송 기준으로 sendfile 과 read/write 의 실행 결과를 상세하게 보여준다. 이를 통해 sendfile 이 read/write 에 비해 3.65 배 더 빠름을 알 수 있으며, 결과적으로 sendfile 이 read/write 보다 대용량 데이터 통신에 더 적합한 시스템 콜임을 알 수 있다.

read 와 sendfile 내부에서 동작하는 커널 함수들을 분석하면 차이는 더욱 명확해진다. 그림 4 는 read 와 sendfile 내부의 데이터 읽기와 관련된 함수들에 대한 콜 그래프(Call Graph)를 도식화하고, 실행 시간을 비교한다. 페이지 웜업(Warm Up)이 완료된 상태에서 각 시스템 콜을 실행시켜 DMA Copy 에 의한 실행 시간 오차를 없애고 커널 버퍼에서 일어나는 실행 시간만을 비교한다.

read 내부에서는 가상 파일 시스템(VFS)을 통해 데이터 읽기가 수행되며, VFS 내 정의되어 있는 read_iter 를 사용한다. 이 함수는 filemap_read 를 통해 페이지 캐시 내부의 데이터를 지정된 버퍼(유저 버퍼)로 복사한다. 반면, sendfile 은 file_splice_read 를 통해 데이터를 읽으며, filemap_read 의 목적지 버퍼를 내부에서 할당된 파이프로 설정한다. 이후 실제 복사가 이루어질 때, 파이프에는 페이지 캐시의 주소만 저장된다.

결과적으로, sendfile 내부의 filemap_read 는 버퍼에 직접 CPU Copy 를 수행하지 않기 때문에, read 내부의 filemap_read 와 비교하였을 때 절반 정도의 시간이 소요되는 것을 그림 4 를 통해 확인할 수 있다.

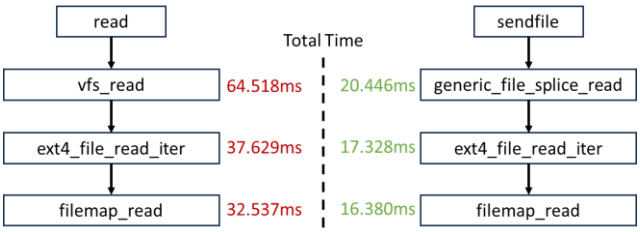

그림 4 read 와 sendfile 의 콜 그래프 및 함수 별 실행 시간 측정 결과

Ⅲ. 결론

본 논문에서는 리눅스 기반의 네트워크 소켓 프로그래밍에서 데이터 전송 메커니즘(read/write, sendfile) 간 성능과 효율성 측면에서 중요한 차이를 보여준다. 또한, 제시된 성능 비교를 통해 시스템 내 데이터 전송 방식을 선택하는 데 중요한 기준을 제공한다. 특히, 대규모 데이터 전송에서 sendfile 을 활용한 Zero-copy 기술의 구현은 CPU 오버헤드를 감소시키고, 전송 속도를 크게 향상시키는 중요한 방법이 될 수 있다. 이러한 분석은 향후 시스템 설계 및 개발에 있어 기초 자료로 활용될 수 있으며, 네트워크 소켓 프로그래밍의 효율성을 향상시키기 위한 방향을 제시한다.

ACKNOWLEDGMENT

이 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구이며(No. RS-2023-00217689), 또한 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업 결과임(2022R1I1A3055233)

참 고 문 헌

[1] Chen, M., Mao, S., & Liu, Y. "Big data: A survey." Mobile networks and applications 19 (2014): 171-209.

[2] Dragan, S. "Zero copy I: user-mode perspective." Linux Journal 2003.105 (2003): 3.

[3] 송민규, 위석오, 이성모, 강용우, 김승래, 제도흥, & 김효령. (2021). “대용량 데이터의 전송 효율 및 기록 성능 향상을 위한 Zero Copy 기술 적용에 관한 연구.” 한국전자통신학회 논문지, 16(6), 1133-1144.

[4] Kim, N.Y., "uftrace.", (https://github.com/namhyung/uftrace).

[5] corbet, “tee() with your splice()?”, 2006, (https://lwn.net/Articles/179492/)