# 리눅스 커널에서 네트워크 멀티미디어 서비스를 위한 메모리 복사 감소 기법 구현

정회원 김 정 원*

# Implementation of Memory Copy Reduction Scheme for Networked Multimedia Service in Linux

Jeong-Won Kim* Regular Members

요 약

MPEG(Motion Picture Expert Group)과 같은 멀티미디어 스트림은 연속적 재생으로 인해 데이터의 지속적인 디스크 검색을 요구한다. 따라서, 커널의 효율적인 지원이 필요한데, 유닉스 계열의 리눅스 버퍼 캐시 시스템은 비정기적이고 비실시간 데이터인 텍스트 데이터용으로 설계되었다. 대용량의 연속 미디어의 경우 커널 주소공간에서 사용자 주소공간으로의 대량의 복사가 이루어지므로 이 과정에서 CPU의 과중한 오버헤드가 발생한다. 이것은 시스템 처리율을 저하시킬 뿐만 아니라 QOS(Quality of Service)도 보장할 수 없다. 본 논문에서 이 메모리 복사 오버헤드를 감소시키기 위한 direct I/O와 one copy 기법을 리눅스 커널에서 설계 및 구현하였다. direct I/O는 디스크의 데이터를 커널 버퍼로 복사하지 않고 사용자 버퍼로 직접 복사하므로 CPU 오버헤드를 획기적으로 감소시킬 수 있다. 그리고, one-copy는 사용자 버퍼로 데이터를 복사하지 않고 직접 네트워크로 전송하는 기법이다. 구현 결과, CPU 오버헤드의 상당한 감소와 시스템의 처리율이 향상됨을 확인하였다.

ABSTRACT

Multimedia streams, like MPEG continuously retrieve multimedia data because of their incessant playback. While these streams need an efficient support of kernel, the current buffer cache mechanism of Linux kernel such as Unix operating system was designed apt for small files, which is aperiodically requested as well as time uncritical. But, in case of continuous media, the CPU must enormously copy memory from kernel address space to user address space. This must lead to a large CPU overhead. This overhead both degrades system throughput and cannot guarantee QOS. In this paper, we have designed and implemented two memory copy reduction schemes in Linux kernel, direct I/O and one copy. The direct I/O skips the buffer cache layer of Linux kernel and results in dramatic reduction of CPU memory copy overhead. And, the one copy provides a fast disk-to-network data path without copying to user address space. The experimental results show considerable reduction of CPU overhead and throughput improvements.

## Ⅰ. 서 론

MOD(Multimedia on Demand), VOD(Video on Demand)와 같은 멀티미디어 서비스는 향후 멀티미디어 초고속통신망 시대에 필수적인 응용이 될 것이다. MPEG 2와 같은 멀티미디어 스트림은 시간당 2GB의 저장공간과 4~60Mbps의 대역폭을 요구한다[1]. MOD 서버는 이 멀티미디어 데이터의 요구 대역폭과 마감시간, 즉 QoS 보장해야 한다. 따라서, 커널의 효율적인 지원이 요구된다. 커널 차원에서

* 신라대학교 컴퓨터정보공학부(jwkim@silla.ac.kr)
논문번호 : 020142-0328, 접수일자 : 2002년 3월 일

멀티미디어를 효율적으로 지원하기 위한 방법으로는 크게 대용량 파일의 효과적인 저장 및 캐싱(cachin g) 그리고, 실시간 전송 및 마감시간보장 등이 있다. 특히 서버 응용프로그램에게 빠른 응답시간을 보장하기 위해서 하드웨어 측면에서 H/W RAID, SAN (Storage Area Network), NAS(Network Attached Storage)와 소프트웨어 측면에서 다양한 배치기법, S/W RAID, 계층형 저장 기법 및 캐싱 기법 등이 연구되어져 왔다[2,3,4].

한편, 리눅스는 안정적이고 공개 커널이므로 응용 프로그램의 목적에 부합하는 커널 기능의 개발이 용이하다. 리눅스 역시 유닉스 계열의 운영체제이므로 평균 100 킬로 바이트 미만의 소규모 비실시간 데이터에 적합하도록 설계되었다. 대용량의 연속미디어 파일의 경우 한 시간 이상씩 디스크에서 커널 주소 공간으로 복사되고, 다시 사용자 주소 공간으로 복사가 이루어진다. 디스크에서 커널 주소 공간으로의 복사는 DMA(Direct Memory Access) 등에 의해 CPU의 부하가 감소될 수 있지만, 커널 주소 공간에서 사용자 버퍼의 주소 공간으로의 복사는 C PU가 관여한다. 소규모의 비정기적인 요구를 가진 비연속형 데이터의 경우는 큰 문제가 되지 않지만, 대용량의 연속미디어의 경우 시스템 처리율을 저하시키고 QoS를 보장할 수 없다.

따라서, 본 논문에서는 멀티미디어 파일의 입출력 응답시간을 개선시키기 위한 커널 지원 방법의 하나인 direct I/O와 one copy 기법을 리눅스 커널에서 설계 및 구현하였다. direct I/O 는 디바이스 드라이버가 디스크에서 읽은 데이터를 커널의 버퍼 캐시 공간으로 복사하지 않고, 사용자가 지정한 버퍼 공간으로 직접 복사하는 기법이다. 따라서, CPU의 메모리 복사에 따른 과중한 부담을 감소시킬 수 있다. one copy 기법은 사용자 버퍼공간으로 복사하지 않고 커널의 버퍼에서 네트워크로 직접 데이터를 전송하는 기법이다. 따라서, 메모리 복사 오버헤드를 획기적으로 감소시킬 수 있다. 본 논문에서는 실험을 통하여 기존 커널과 direct I/O 와 one copy 의 성능을 비교 분석하여 MOD 서버에서의 리눅스 커널의 효율적인 지원을 확인하고자 한다.

논문의 구성은 다음과 같다. 2장에서는 관련연구를 기술하고, 3장에서는 리눅스 주소 변환 기법과 설계된 direct I/O 메커니즘을 설명한다. 4장에서는 one copy 기법을 설명하고, 5장에서는 커널 구현 환경과 성능 측정 결과를 설명한다. 마지막으로 본 논문의 결론과 향후 연구 방향에 대해 논한다.

## Ⅱ. 관련연구

커널상에서 메모리 복사 오버헤드 감소를 위한 관련 연구에는 zero copy, IRIX의 direct I/O, 전통적인 유닉스의 raw I/O 등이 있다.

Milind[5] 는 데이터 복사 오버헤드를 감소시키기 위한 zero copy 기법을 4.4 BSD Unix에서 설계 및 구현하였다. 이 기법은 mmbuf 라는 새로운 버퍼 관리 시스템을 통하여 디스크에서 커널 버퍼로 전송된 데이터가 사용자 버퍼로 복사되지 않고 네트워크 소켓으로 직접 전송된다. 각 스트림은 버퍼 블록의 리스트를 유지하여 네트워크 드라이브로 블록을 전송한다. 기존 버퍼 관리 시스템과 호환되지만 독립적인 페이지 풀과 버퍼 관리 메커니즘을 제공해야 한다.

SGI IRIX 운영체제의 XFS는 파일 시스템의 데이터를 사용자 버퍼에 직접 전송하는 direct I/O를 제공한다[6]. IRIX의 버퍼 캐시 모듈은 XFS와 사용자 메모리의 버퍼 블록을 상호 연결하여 일반I/O를 수행하듯이 direct I/O를 수행한다. 이때, 사용자 주소의 버퍼 블록은 I/O도중 스왑핑(Swapping)이 되지 않기 위해서 락(lock)된다. 이 direct I/O는 대규모 데이터 베이스가 파일시스템에 저장되어 있을 때, 혹은 대규모 파일이 고속으로 전송될 경우 효율적임이 증명되었다.

SUN Solaris와 같은 기존 유닉스 계열의 커널은 디바이스 파일을 유지하여 장치와 파일시스템에게 추상화(abstraction)와 독립성(independency)을 제공한다. 예를 들어 fopen("/dev/rsda1", "r") 라이브러리 호출을 하면 커널의 버퍼 캐시를 스킵하는 기능을 제공한다[7]. 따라서, 메모리 복사 오버헤드를 감소시킬 수 있다. 그러나, 이것은 파일 시스템내의 일반 파일에 대해서는 적용할 수 없다. 현재 리눅스 커널도 디바이스 파일에 대한 I/O를 수행할 수 있지만 버퍼 캐시를 사용한다. Stephen Tweedie 는 이 raw I/O를 리눅스 커널의 개발자 버전에서 구현하여 패치 수준으로 제공하고 있고 커널 버전2.4에서 공식 포함되었다[8].

## Ⅲ. Direct I/O의 설계 및 구현

리눅스 파일시스템의 읽기 및 쓰기 연산은 항상 버퍼 캐쉬 시스템을 통과한다. 캐쉬의 적중률이 높

은 경우에는 디스크 접근 회수를 줄여 응답시간의 향상을 가져온다. 그러나, MOD와 같은 멀티미디어 데이터의 경우는 다음의 결과들을 초래할 수 있다. 1) 연속성의 특성을 가지고, 데이터의 재사용성이 낮아 높은 캐쉬 적중률을 기대하기 어렵다. 2) 사용 가능한 메모리가 부족할 경우에는 잦은 페이지 스왑핑이 발생하여 기대하는 버퍼 캐쉬의 효과를 얻기 힘들다. 3) 대량의 데이터를 장시간 전송해야 하므로 CPU의 메모리 복사 오버헤드가 상당히 증가할 것이다.

따라서, 리눅스 커널에서 버퍼 캐시 계층을 스킵하여 사용자 메모리에 데이터를 직접 전송하는 direct I/O 는 CPU 의 메모리 복사 오버헤드를 감소시키고, 응용 프로그램에게 신속한 응답시간을 제공할 수 있다. 또한, 대용량 파일의 연속적인 읽기 요구에 잘 적응할 수 있다. IRIX 의 direct I/O 에서도 그 성능이 검증된 바 있다[9,10].

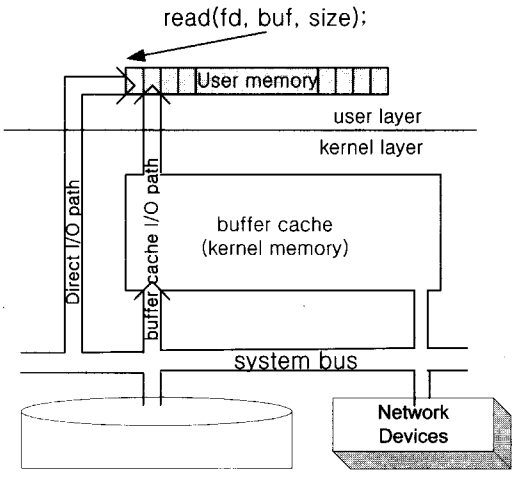


그림 4 버퍼 캐시 I/O경로와 direct I/O경로

그림 1 은 버퍼 캐시 I/O경로와 direct I/O 의 데이터 경로이다. 버퍼 캐시 I/O의 경우, 모든 읽기 연산에 대해 2번의 복사가 이루어진다. 네트워크 전송의 경우를 포함하면 소켓 버퍼로의 복사를 합쳐서3번의 복사가 발생한다. 반면 direct I/O의 경우 그림 1에서 보듯이 블록 디바이스 드라이브는 사용자 메모리로 데이터 블록을 직접 복사하기 때문에 1번의 복사만 발생한다. 따라서, CPU의 메모리 데이터의 복사 오버헤드는 존재하지 않는다.

i386 용 리눅스 커널에서 direct I/O를 구현하기 위한 핵심 과제는 사용자가 할당한 버퍼의 주소, 즉 가상주소(virtual address)를 실제 물리적 주소로 변환하는 것이다. 블록 디바이스 드라이브는 물리적 장치로부터 읽어 들인 블록을 물리 매모리로 전송해야 한다. 이때 자신의 모듈로 넘겨진 버퍼 헤드의 데이터 주소 공간은 가상 주소가 아닌 물리 주소여야 한다. 따라서, direct I/O를 커널에서 구현하기 위해서는 사용자가 제공한 가상 주소를 물리적 주소로 변환하여 블록 디바이스 드라이브로 제공해야 한다.

### 1. 가상주소에서 물리주소로 변환

i386 계열의 CPU는 페이지화된 세그멘테이션 기법으로 주소를 변환한다[11]. 메모리에 대한 사용자관점과 물리적인 관점이 동일한 세그멘테이션 기법의 장점과 외부 단편화를 제거할 수 있는 페이징 기법의 장점을 결합한 것이다. CPU, 리눅스 커널은 가상 주소에서 동작하므로 라이브러리 호출 read(fd, buffer, size) 에 의해 제공된 가상주소는 시스템 호출 인터페이스 sys_read()에서 선형주소(linear address)로 변환된다[12]. 이 선형 주소는 페이지 테이블을 참조하여 물리주소(physical address)로 변환된다. 본 연구에서는 사용자 버퍼의 물리주소를 얻기 위해 page_map() 함수를 제공한다. 그림 2는 주소 변환의 과정을 보여준다.

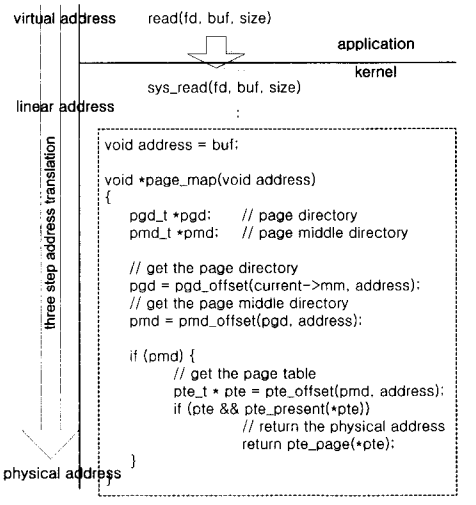


그림 6 주소 변환

리눅스 커널은 3단계 페이징을 제공한다. 즉, page directory, page middle directory, page table이다. 알파 프로세서 계열에 탑재되는 리눅스 커널은 페이지 디렉토리, 페이지 중간 디렉토리, 페이지 테이블의 3 단계 참조를 통해 물리주소를 얻는다. 반면 i386 계열의 프로세서에 탑재되는 리눅스 커널은 2

단계 주소 변환을 하므로 page middle directory는 하나만 존재한다[13]. 사용자가 제공한 버퍼의 가상주소는 페이지 디렉토리와 페이지 테이블을 참조하여 페이지 프레임의 시작주소를 찾아낸다. 이 시작 주소에 선형주소의 오프셋 값을 더하면 실제 물리주소로 변환된다[12]. 다음은 본 논문에서 구현한 그림 2 의 주소 변환 과정 주요 부분을 설명한다. read(), write() 라이브러리 호출 시 sys_read(), sys_write() 시스템 호출이 각각 발생하는데 이때 파라메터로 넘겨진 사용자 버퍼의 주소는 선형주소이다. 먼저 시스템 콜 인터페이스로 넘겨진 선형주소와 시스템 호출을 수행한 프로세스의 가상메모리 구조체를 가리키는 포인터(current->mm)를 pgd_offset()함수에 입력하면 페이지 디렉토리의 시작 주소를 얻을 수 있다. 이 반환된 값(pgd)과 선형주소를 다시 pmd_offset()함수에 입력하면 페이지 중간 디렉토리의 주소를 얻을 수 있다. 마지막으로 pte_offset() 함수를 호출하면 원하는 물리주소에 해당하는 페이지의 시작주소를 얻을 수 있다.

## 2. 구현

본 연구의 direct I/O 는 리눅스 커널 버전 2.4 에서 구현되었다. 1)에서는 일반 파일에 대한 direct I/O를 수행하기 위한 시나리오를 살펴보고 2)에서는 실제 구현한 함수 direct_rw()를 설명한다.

### 1) Direct I/O의 시나리오

direct I/O를 위해 새로운 open 시스템 콜을 커널에 추가할 필요는 없다. 사용자 프로그램의 파일 개방 옵션(open flags)에 direct I/O 임을 나타내는 새로운 플래그를 추가하고, 이 플래그를 커널이 인식하면 된다.

예를 들면, 파일 open 시,

open( filename, O_RDONLY | O_DIRECT); // O_DIRECT : 040000

와 같은 open 플래그를 사용하였다면 해당 프로세스의 개방 파일의 정보를 유지하는 구조체의 flags 필드에 이 정보를 저장한다. 실제로 read()나 write() 연산이 발생될 때, 그림3과 같이 기존 I/O패스를 거치지 않고 새로운 I/O경로(direct_rw())를 따른다. 이 방법은 파일의 개방 플래그만을 사용하므로 두 방식의 read() 연산을 위해 각각을 위한 커널 파일 포인터 구조체 및 inode 정보를 따로 저장할 필요가 없다. 또한, 각각의 시스템 콜 인터페이스를 만들 필요가 없으므로 매우 효율적이다.

### 2) 구현

direct I/O 에서 읽기와 쓰기 연산은 디바이스 드라이브로 읽기/쓰기 플래그만 다르고 모든 과정이 동일하다. 본 연구의 목적이 읽기 연산의 응답시간과 CPU 메모리 복사 오버헤드 감소이므로 읽기 연산 위주로 설명한다. 그림 3은 리눅스 커널의 read() 시스템 호출에 대한 단계를 보여주고 있다. 리눅스는 ext2, ufs, nfs, minix 등 지역파일시스템 들에 대한 공통 인터페이스로 가상파일시스템(virtual file system) 인터페이스를 제공한다. 그림 3의 두 번째 단계인 generic_readpage()는 모든 지역파일시스템의 읽기 연산 시 공통적으로 사용되는 함수로서 버퍼 캐시 I/O 를 제공한다. 따라서, direct I/O 에서는 이 generic_readpage()를 호출하지 않고 새로이 추가된 direct_rw()함수를 호출한다. direct_rw() 함수는 총 6 단계로 구성되는데 그림4는 이를 보여주고 있다. 다음은 direct_rw() 함수의 단계별 설명이다.

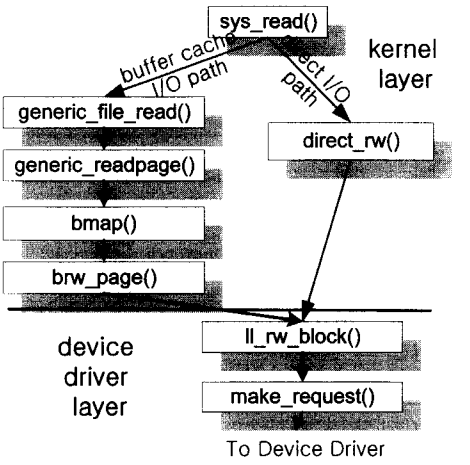


그림 8 direct I/O 경로와 buffer cache I/O 경로의 함수

(1) 플래그 검사

sys_read() 시스템 콜 인터페이스에서 커널의 개방 파일 구조체에 O_DIRECT 플래그가 설정되어 있는지 검사한다. 성공하면 단계 2로 분기한다.

(2) 커널 페이지 구조체 작성

사용자 버퍼의 가상주소공간에 대해 입출력을 수행하기 위한 커널 페이지 구조체를 만든다. 현재 x86 용 리눅스 커널은 4 킬로바이트의 페이지 크기를 지원하므로 사용자 버퍼공간을 4 킬로 바이트 단위로 분할하여 구조체를 생성하는 것이다. 그림5는 direct I/O를 위해서 커널 입출력 구조체(kiobuf)

이다.

(3) 주소 변환

각 페이지에 대해 가상주소를 물리주소로 변환한다. 이는 (1)에서 설명한 바와 같다.

(4) 페이지 잠금

주소 변환된 각 페이지에 대해 페이지를 잠근다(locking). 이는 커널이 해당 페이지에 대한 입출력 수행 도중 스왑핑이 발생하는 것을 막기 위해서이다.

(5) 버퍼 헤드 초기화 및 블록 매핑

리눅스 커널의 각 페이지는 버퍼 캐시 헤드로 구성된다. 따라서, 파일의 오프셋이 소속된 파일시스템 상의 논리적인 블록 번호를 inode를 참조하여 얻는다. 이 단계는 direct I/O를 위해 구축된 각 페이지에 대하여 버퍼 헤드를 초기화함을 의미한다. 사실 이 버퍼 헤드 구조체는 기존 버퍼 캐쉬 I/O에 있어서 버퍼 헤드와 동일하지만 버퍼 헤드가 가리키는 데이터의 주소는 커널 세그먼트상의 주소가 아니라 사용자 프로그램의 데이터 세그먼트상의 물리적 주소가 된다.

(6) 디바이스 드라이버 입출력

단계 5에서 구축된 각 버퍼 헤드에 대하여 블록 디바이스 드라이버에 대한 함수를 호출하여 실제 입출력을 수행한다. 이 과정은 페이지가 남아 있을 때까지 계속 수행된다. 입출력이 완료되면 사용자 프로그램에게 총 입출력한 바이트 수를 반환한다.

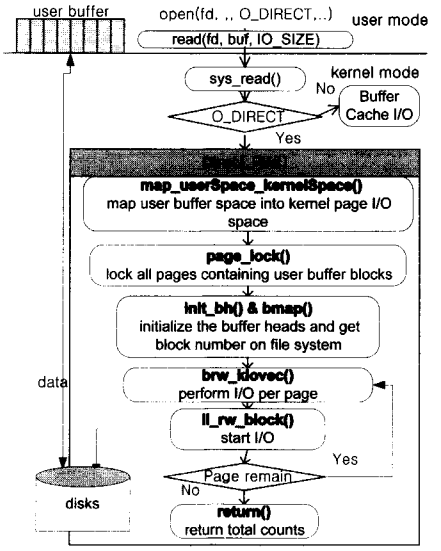


그림 10 direct_rw()의 플로우 차트

```
struct kiobuf
{
  int  nr_pages;  // number of mapped pages
  int  offset;     // offset of page
  int  length;     // total data size

  unsigned long * pagelist; // page list pointer
  unsigned int   locked : 1;  // page lock flag
  unsigned long   page_array[]; // page array
};
```

그림 11 커널 매핑을 위한 자료 구조

## Ⅳ. One Copy

본 연구에서 direct I/O의 구현결과 다음과 같은 특징을 발견하였다. 1) 사용자의 I/O단위는 512 바이트의 배수가 되어야 한다. 디스크 드라이브는 물리적인 섹타 크기 단위로 데이터를 전송하기 때문이다. 2) 버퍼 캐시 시스템은 미리읽기(read ahead)를 통하여 선반입을 수행하지만 direct I/O는 지원할 수 없다. 3) 비교적 큰 블록 크기의 파일 시스템에 효과적이다. 결과적으로 direct I/O는 CPU 오버헤드는 획기적으로 감소시킬 수 있지만, 응용프로그램의 융통성과 디스크 접근 회수를 감소시킬 수 없다.

따라서, one copy가 고안되었다. one copy는 스트림을 목적지로 전송할 때 디스크에서 커널 주소 공간으로 한번의 복사만 이루어짐을 의미한다. Zero copy는 커널 내에 독자적인 버퍼캐시 관리 시스템인 Mmbuf를 구축하여 메모리 복사 오버헤드를 감소시켰다. 커널 내에 추가되는 코드가 비교적 많을 수 있지만, One copy 에서는 간단한 방법으로 구현된다. 핵심은 기존 버퍼 캐시 블록을 그대로 사용하면서 memory inode와 소켓을 연결하는 것이다. 한 스트림이 개방되어 소켓을 통해서 전송될 때, 파일 디스크립터와 소켓 구조체는 1대1매핑관계를 유지한다. 1 에서는 one copy 입출력 경로에 대한 시나리오를 설명하고 2 에서는 커널 구현 내용을 상세히 설명한다.

### 1. One copy의 I/O 시나리오

먼저, One copy의 입출력 시나리오를 설명한다.

1) 파일오픈

open(filename, flag = O_ONECOPY, mode,,,) 으로 파일을 개방한다. 커널내의 sys_open() 시스템 호출 인터페이스는 파일의 struct file 포인터 구조

체에 개방 파일이 one copy 임을 명시한다.

2) 소켓 생성

접속에 필요한 소켓을 생성한다.

3) 연결

fcntl(socket ID, O_ONECOPY, open file ID) : 소켓 ioctl호출을 통해 커널 소켓 구조체에 파일 ID에 해당하는 memory inode를 연결한다.

4) 파일 읽기

read(open file ID, buffer, size) : 파일에 대한 읽기를 수행한다. inode는 현재 사용된 버퍼 헤드에 대한 포인터와 크기를 유지한다. 전송을 위해 버퍼 헤드를 잠근다.

5) 데이터 전송

send( socket ID, buffer, size) : 소켓에 연결된 i node를 참조하여 버퍼 헤드를 구한다. 소켓 버퍼의 포인터를 버퍼헤드의 데이터 포인터를 리다이렉션하여 패킷을 전송한다. 전송이 완료되면 버퍼헤드를 해제한다.

## 2. One copy를 위한 커널 수정

이 절에서는 One copy를 위한 커널의 주요 변경 사항을 설명한다. 그림 6은 inode와 소켓 구조체를 상호 연결하는 one copy의 구조를 나타낸다.

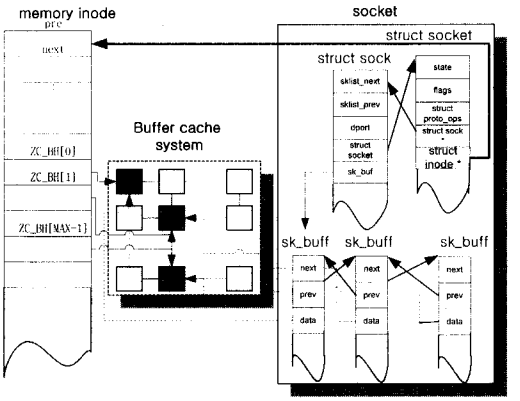


그림 13 One copy의 블록 다이어그램

1) 변경된 메모리 inode

변경된 메모리 inode는 매 읽기 연산에 사용된 버퍼 헤드에 대한 포인터 배열(zc_bh[])과, 버퍼 헤드의 개수(current_zc_nr)를 유지한다. current_zc_nr의 최대수는 멀티미디어 서버의 주기 당 최대 읽기 크기로 결정된다.

2) 변경된 소켓 구조체

변경된 소켓의 struct socket 구조체는 현재 프로세스의 개방파일 테이블을 검색하여 one-to-one 사상되는 inode 포인터를 유지한다. 이때 개방 파일 기술자는 멀티미디어 서버의 해당 소켓에 대한 fcntl () 호출에 의해 파라메터로 커널 함수로 넘겨진다.

3) 변경된 generic_readpage() 함수

리눅스 커널은 파일 시스템의 종류에 독립적으로 generic_readpage() 함수를 통해 읽기 연산을 수행한다. 이 함수는 각 페이지에 사상된 버퍼 헤드에 대한I/O를 블록 디바이스 드라이브의 ll_rw_block()을 호출하여 수행한다. 그리고 버퍼 캐시 블록에 대한 I/O가 완료되면 copy_to_user()함수를 수행하여 사용자 버퍼 공간으로 데이터를 복사한다. 그러나, One copy에서는 이 copy_to_user() 함수를 수행하지 않고 리턴한다. 즉, inode에 현재 읽기 연산에 사용된 buffer_head에 대한 포인터를 유지한다.

4) 변경된 소켓 버퍼 구조체

소켓에 대한 send() 연산이 수행되면 소켓은 one-copy 인지를 먼저 검사한다. 소켓은 그림6 에서 보듯이 sk_buff 의data 포인터를 inode의 버퍼 헤드의 data 포인터로 사상시킨다. 프로토콜 스택은 이 포인터의 데이터를 네트워크로 전송하게 된다.

그림6 에서 보듯이 메모리 inode와 소켓구조체의 sk_buff는 동일한 버퍼 헤드를 가리킨다. one copy는 커널 코드의 최소 변경으로 디스크와 네트워크로의 신속한 데이터 경로를 제공한다. 또한 기존 버퍼 캐시 시스템과 호환되며, 미리읽기 기능도 이용할 수 있다. 따라서, direct I/O보다는 나은 융통성을 제공하고 디스크 접근 회수도 감소시킬 수 있다.

# V. 실험 결과

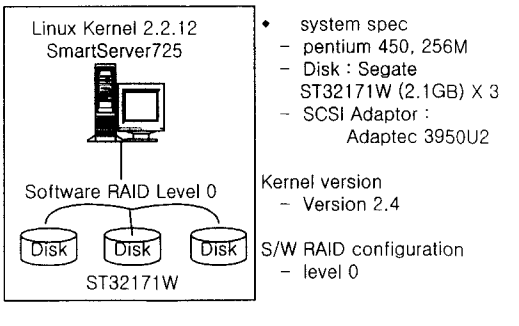


그림 14 실험 환경

본 장에서는 direct I/O, one copy I/O의 구현 환경과 성능 측정 결과를 설명한다. 커널 버젼 2.4 에서 direct_I/O 와One copy를 설계 및 구현하였는데 그림 7은 성능 측정을 위한 실험 환경이다. stand a

lone의 시스템에 소프트웨어 RAID 툴(raidtools) 을 사용하여 다중 디스크 환경을 구축하였다. [표 1]은 사용된 디스크의 종류와 저장된 파일의 특성을 구체적으로 기술하고 있다.

각 실험은 단일 파일에 대한 성능 측정과 MOD

표 3 Experiment parameters

| parameter | description |
|---|---|
| disk | seagate ST32171W (2.1GB) x 3<br>max bandwidth : 80Mbps<br>min bandwidth : 120Mbps |
| total capacity | 6.3GB |
| RAID | level 0(simple striping) |
| video file | MPEG1, 300MB |
| total videos | 30 |

환경에서의 성능 측정을 수행하였다. 두 경우 모두 비디오 파일은4KB 블록 크기의 ext2파일시스템에 저장되었고, normal read, no readahead와 비교되었다. 단일 파일 read의 경우는 CPU 부하, 파일의 크기에 따른 응답 시간, 그리고 read()/send() 조합에 대한 응답 시간을 측정하였다. MOD 환경에서는 연구의 목적과 부합하도록 가상의 요구 쓰레드(request thread)를 발생시켰다. 실험에 앞서 실험 시간, 시간당 평균 사용자수 및 접근 확률을 입력받아서 요구의 도착시간간격(interarrival time)과 비디오 파일 번호를 각각 생성하여 로그 파일에 저장한다. 이 도착 시간 간격은 포아송(poisson) 분포 함수에 의해 계산되고, 비디오 파일 번호는 Zipf 분포(θ : 0,271) [14] 를 따른다. 또한, 각 작업 쓰레드는 초당 1.5 M bits 로 디스크에 읽기를 요구하는데 각 주기의 읽기 경과 시간(elapsed time)을 로그 파일 형태로 저장하여 성능 평가의 자료로 사용되었다.

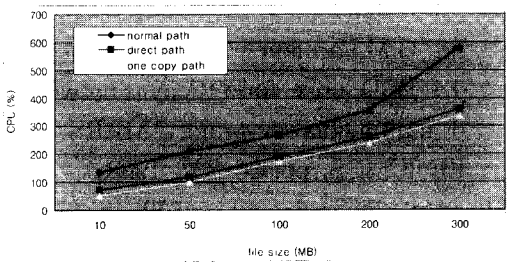


그림 16 단일파일 읽기

그림 8 의 실험 목적은 단일 파일을 읽기에 대한 direct I/O 와 one copy가 기존 읽기 연산보다 성능 향상을 보임을 증명하는 것이다. 이 목적을 위해서 각각의 읽기에 대해서 100번의 실험을 한 후 CPU의 평균 오버헤드를 기록하였다. 그림의 결과는 direct I/O 와one copy는 디스크에서 사용자 메모리로 또는 커널 메모리로 한번의 복사만 발생하지만 normal path의 경우는 두 번의 복사가 발생하므로 CPU의 오버헤드가 커짐을 확인할 수 있다.

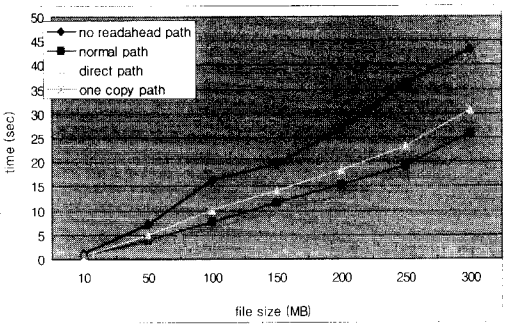


그림 17 파일 크기에 따른 성능 비교

그림 9의 실험 목적은 읽기 연산에 대한 direct path와 one path의 성능 향상을 보이는 것이다. 이 실험에서는 normal path에 대해서 no readahead path 실험을 추가하였다. 리눅스 커널은 디스크 자원의 효율적 사용을 위해 디스크 블록의 선반입 시켜서 계속적인 읽기에 대하여 디스크 입출력을 감소시킨다. 따라서 커널에서 readahead 기능을 수행하는 부분을 스킵하는 읽기 연산을 비교대상으로 하여 실험하였다. direct path는 no readahead path 보다는 성능 향상을 나타내지만 normal path보다 감소된 응답시간을 보인다. 이것은 미리읽기 기능으로 인해서 normal path의 경우는 디스크 입출력이 감소되기 때문이다. 그러나 one copy는 다른 세path 보다 성능의 향상을 보이고 있고 normal path 보다 10%의 성능 차이를 나타내고 있다.

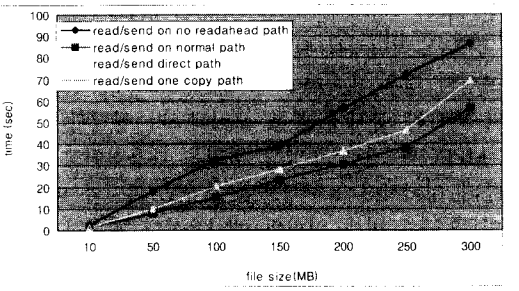


그림 18 디스크에서 네트워크로의 전송

본 실험에서 동일 파일에 대하여 32킬로바이트의 단위 읽기 요구를 다중 디스크상에서 읽고 네트워크로 전송하는 실험을 반복하였다. 그림 10은 그 결과를 보여주고 있다. one copy는 지속적으로 다른 기법보다도 적은 완료시간을 보임을 확인하였다. 그 성능의 차이는5%에서 15 %에 이르고 있다.

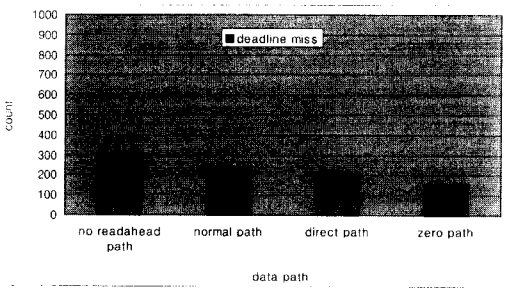


그림 19 마감시간 측정

그림 11 에서는 MOD 환경에서 각 기법의 성능을 측정하였다. 실험은 1시간 동안 100명의 사용자가 발생하는 포아송 분포, 초당 1.5Mbps의 전송률, 30개의 비디오 파일에 대해 0.271 Zipf패턴으로 파일을 접근한다. 총1000 주기(1주기 3초) 동안 발생한 deadline miss 발생 회수를 측정하였다. 그 결과 미리 읽기를 하지 않는 no readahead path가 최악의 성능을 보였다. 그리고 direct path가 normal path 보다도 약간 나은 성능을 보인다. 이것은 MOD와 같이 과부하의 상태에서는 단일 파일 읽기에서처럼 미리읽기를 수행할 수 없음을 의미한다. 또한 미리 읽기를 하더라도 캐싱의 효과를 얻을 수 없기 때문이다.

## VI. 결 론

본 연구에서는 멀티미디어를 효율적으로 서비스하기 위해 CPU 오버헤드와 응답시간을 감소시키기 위한 커널 지원의 한 방법인 direct I/O와 one copy를 리눅스 커널에서 설계 및 구현하였다. 연속 매체 스트림의 경우 계속적인 읽기/전송 요구로 인해 CPU의 메모리 복사에 따른 오버헤드가 상당하다. 또한 MPEG과 같은 동영상 파일은 대용량이고 읽기 위주의 정기적인 요구이므로 소규모 파일의 비정기적인 요구에 비해 버퍼의 재사용성이 현저하게 낮다. direct I/O는 단일 파일 읽기 요구에서는 CPU 오버헤드는 낮지만 미리읽기 기능의 결여로 정상적인 읽기 요구보다 낮은 응답시간을 보였다. 그러나, MOD와 같이 버퍼 캐시 시스템이 과부하일 경우 나은 성능을 보였다. one copy 는 CPU 오버헤드는 direct I/O보다 높지만 단일 파일 읽기, 읽기/전송 및 MOD 환경에서는 나은 성능을 보였다.

본 논문에서는 리눅스 커널상에서 멀티미디어 데이터를 효율적으로 지원하기 위해 다양한 메모리 복사 오버헤드 감소기법을 설계 및 실험하였는데 제안한 one copy기법이 최상의 응답시간을 보임을 확인하였다. 향후 연구과제로는 제안한 시스템을 VOD, NOD 등의 시스템에 직접 적용하여 성능을 확인하고, 분산 환경에서 제안한 기법을 확장하는 것이다.

## 참고문헌

[1] Prabhat k. andleigh, Kiran thakrar, Multimedia Systems Design, pp.112, Prentice Hall PTR, 1996.

[2] Yuewei wang, David H.C.Du, "Weighted striping in multimedia servers", Proc.of IEEE on multimedia computing and systems. pp 102-109, June, 1997.

[3] Yuewei wang, Johathan C.L. Liu, David H.C. Du and Jenwei, 갋fficient video file allocation schemes for video on demand services", ACM multimedia systems journal, vol.5, no.5, 1997.

[4] Renu tewari, Daniel M. Dias, ajit mukherjee, harrick M. Vin, "High availability in clustered multimedia servers," Proc. of the USENIX annual technical conference, Jan, 1996.

[5] M.M.Buddihikot, X,J.Chen, D.Wu, and G.M.Parulkar, "Enhancements to 4.4 BSD unix for efficient networked multimedia in project MARS," IEEE ICMCS, pp.326-337, 1998.

[6] http://techpubs.sgi.com/library/manpages/open.html

[7] Maurice J. Bach, The design of the Unix operation system, Englewood Cliffs, NJ 07632: Prentice-Hall, Inc., 1986.

[8] Stephen Tweedie, PATCH : Raw device I/O for 2.1.131, http://www.linuxhq.com/lnxlists/linux-kernel/lk_9812-02/msg00686.html, December 1998.

[9] Steen R. Siltis, Thomas M.Ruwart, Matthew T.O'keefe, "The global file system", Proc.of the fifth NASA goodard space center conference on mass storage systems and technologies, sept 17-19, 1996.

[10] Jim Mostek, William earl, and dan Koren, Porting the SGI XFS File System to Linux , white paper, http://oss.sgi.com/projects/xfs/

[11] Silberschatz, galvin, Operating systems concepts, fifth edition, pp304,1998.

[12] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Vervorner, Linux Kernel Internals, pp.148-151, Addison Wesley.

[13] Remy card, eric dumans, and frank mevel, The Linux kernel book, Wiley, pp 286~298, 1999.

[14] D.E. Knuth, The art of computer programming, vol.3 : Sorting and searching. Addison-wesley, 1973.

김 정 원(Jeong-Won Kim) 정회원

1995년 2월 : 부산대학교 전자계산학과 학사

1997년 2월 : 부산대학교 전자계산학과 석사

2000년 6월 : 부산대학교 전자계산학과 박사

2000년 9월 ~ 2002년 2월 : 기술신용보증기금 전문직차장

2002년 3월~현재 : 신라대학교 컴퓨터정보공학부 전임강사

<관심분야> 운영체제, 내장형 컴퓨터, 멀티미디어