플래시 메모리를 저장매체로 사용하는 임베디드 시스템에서의 정규파일 접근

이 은 주* · 박 현 주**

Regular File Access of Embedded System Using Flash Memory as a Storage

Eun-Joo Lee* · Hyun-Ju Park**

Abstract

Recently Flash Memory which is small and low-powered is widely used as a storage of embedded system, because an embedded system requests portability and a fast response. To resolve a difference of access time between a storage and RAM, Linux is using disk caching which copies a part of file on disk into RAM. It is not also an exception on embedded system. A READ access-time of flash memory is similar to RAMs. So, when a process on an embedded system reads data , it is similar to the time to access cached data in RAM and to access directly data on a flash memory. On the embedded system using limited memory, using a disk cache is that wastes much time and memory spaces to manage it and can not reflects the characteristic of a flash memory. This paper proposes the regular file access of limited using a page cache in the file system based on a flash memory and reflects the characteristic of a flash memory. The proposed algorithm minimizes power consumption because access numbers of the RAM are reduced and doesn't waste a memory space because it accesses directly to a flash memory. Therefore, the performance improvement of the system applying the proposed algorithm is expected.

Keywords : Flash Memory, Embedded System, File Access, Page Cache, Disk Cache

논문접수일 : 2003년 9월 3일　　논문게재확정일 : 2004년 1월 5일
* (주)세인정보통신
** 한밭대학교 정보통신컴퓨터공학부

1. 서 론

정보통신 산업의 발전에 따라, 기존의 가전제품이나 전자제품들이 단순히 자체의 기능만을 수행하는 것을 넘어 정보가전으로 발전하고 있으며, 통신기기 및 휴대기기 등도 프로세싱이 필요한 임베디드 시스템(Embedded System)으로 발전하고 있다. 이런 임베디드 시스템은 휴대성, 무선통신, 소비전력, 사용자 인터페이스, 보안 등 점진적으로 좀 더 복잡한 기능을 요구하고 있으며, 제한된 자원을 효율적으로 이용하고 복잡한 환경에서도 높은 성능을 발휘할 수 있도록 운영체제가 효율적으로 설계되어야 한다.

휴대가 용이하고 빠른 접근시간을 요구하는 임베디드 시스템의 특성상 부피와 무게가 크고 소비전력이 큰 하드디스크 같은 저장장치를 사용하는 것은 비효율적이다. 따라서 크기도 작고, 빠르고 쉽게 저장할 수 있는 플래시 메모리(Flash Memory)가 임베디드 시스템의 저장매체로써 널리 이용되고 있다. 또한 플래시 메모리는 자기식 디스크와는 달리 소비전력이 적고 외부충격에 강하며, 램(RAM) 만큼 접근 시간이 빨라 시스템의 성능 향상이 기대된다. 램과는 달리 전원이 꺼진 상태에서도 데이터를 계속적으로 유지(non-volatile)할 수 있지만, 플래시 메모리는 지움 정책(Cleaning policy) 때문에 수정 시 시간이 오래 걸리며, 수명이 영구적이지 못한 단점이 있다[4].

<표 1>은 플래시 메모리의 특성을 요약한 것이다.

임베디드 시스템에 사용되는 운영체제로는 WindowsCE, PocketPC, Palm OS, 리눅스(Linux) 등이 있다. 그중 리눅스는 커널(Kernel) 소스가 공개되어 있으며, 운영체제의 안정성과 신뢰성이 검증되어 있다. 또한 개발 도구가 모두 개방되어 있기 때문에 개발비용이 적게 들어 경쟁력을 높일 수 있다는 장점이 있어 임베디드 시스템의 운영체제로 각광을 받고 있다. 리눅스를 운영체제로 사용하는 임베디드 시스템은 리눅스가 지원하는 파일시스템을 모두 사용할 수 있다. 따라서 임베디드 시스템의 장점을 살릴 수 있고 플래시 메모리에 운영되기 적합한 파일시스템을 사용할 수 있다. 이러한 파일시스템에는 Initial Ramdisk, JFFS(Journaling flash file-system) 등이 대표적이며, Initial Ramdisk의 경우는 플래시 메모리에 이미지의 형태로 존재하며 운영체제가 지원하는 램 디스크로 복사되어 사용된다. 램 디스크는 전원이 끊어지면 내용을 보존할 수 없는 비영구적인 저장 공간이지만, 램을 디스크로 사용하기 때문에 가장 빠르다는 장점을 가진다. JFFS는 비교적 대량의 기억장치인 플래시 메모리를 목표로 개발되었고 로그 파일시스템 구조를 채택하고 있기 때문에 시스템의 예기치 않은 오류에 대하여 파일시스템의 내용이 파괴되지 않는 장점이 있다[2, 3].

〈표 1〉 플래시 메모리의 특징

특징 항목	값
Read Cycle	70~110ns
Write Cycle	12μs/byte(Typical)
Erase Cycle	0.5~1s/block
Erase Cycles limit	100,000cycles
Erase Unit Size	8~64kbytes
Power Consumption	8~55 mA(active state) 7~250 μA(standby state)

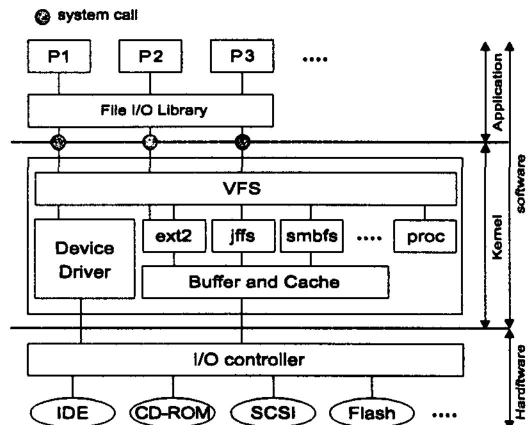
최근에는 인터넷 및 통신 서비스의 발전으로 텍스트기반의 정보뿐만 아니라 이미지, 오디오, 동영상, 게임 등 대용량의 데이터 사용이 폭발적으로 늘어나고 있고, 개인정보의 활용이 높아지면서 임베디드 시스템에서도 다량의 정보를 저장 할 수 있는 저장장치가 요구되고 있고, 보다 안정적인 플래시 메모리의 사용이 급증하고

있다. 따라서, 시스템은 다양한 멀티미디어 데이터를 처리하기 위해 메모리 사용이 빈번해지고, 전력 소비가 높아진다. 임베디드 시스템의 기능이 복잡해지고 다양해짐에 따라 제한된 메모리의 활용도를 높이고, 전력 소모를 낮추는 것은 경쟁력 향상의 요인이다.

PC 기반의 리눅스는 시스템의 성능 향상을 목표로 디스크 캐시(Disk cache)를 사용하고 있다. 이것은 임베디드 시스템에서도 그대로 채택되어 사용되고 있고, 램이나 플래시 메모리를 기반으로 한 파일시스템도 예외는 아니다. 디스크 캐시는 파일시스템의 접근 속도를 향상시키기 위한 목적으로 사용되고 있고, 리눅스에서는 페이지 캐시와 버퍼 캐시로 구성되어 있다[4]. 그러나 임베디드 운영체제는 그 시스템의 특징과 플래시 메모리의 특성을 고려하여 설계되어야 한다. 임베디드 시스템의 저장장치로 사용되는 플래시 메모리의 READ 속도는 램과 유사한 성능을 가지고 있고, 램 디스크의 성능은 다른 매체로 구성된 파일시스템보다 우수한 성능을 나타낸다. 그러므로 임베디드 시스템에서 많이 사용되는 램 디스크나 플래시 메모리 파일시스템을 사용하는 운영체제는 그 특징을 반영하여 효율적으로 설계되어야 한다. 따라서 본 논문에서는 파일 시스템이 RAM이나 플래시 메모리에서 운영될 경우, 파일 Read 시 페이지 캐싱 단계를 축약하여 메모리의 활용도를 높이고, 시스템의 성능을 향상시키고자 하였다. 본 논문의 구성은 다음과 같다. 2장에서는 리눅스의 페이지 캐시에 대해서 살펴보고, 3장에서는 기존의 정규파일 접근과 본 논문에서 제안하는 임베디드 시스템에서의 효율적인 정규파일 접근 기법을 기술한다. 4장에서는 기존의 방법과 제안된 알고리즘의 성능을 비교, 분석하고, 5장에서는 결론을 맺고, 향후 연구과제에 대해 살펴본다.

2. 리눅스의 페이지 캐시

리눅스의 파일시스템은 <그림 1>과 같은 구조로 VFS(Virtual Filesystem) 계층을 기준으로 상위로는 시스템 콜 인터페이스를 가지며, 하위로 여러 파일시스템을 둔 구조로 되어 있다.



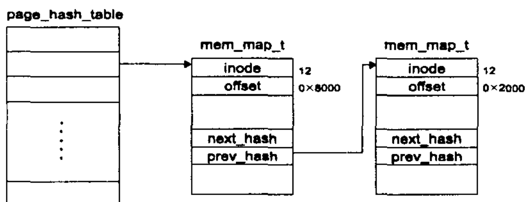
〈그림 1〉 파일 시스템 구조

VFS는 표준 유닉스 파일시스템과 관련한 모든 시스템 콜(System Call)을 처리하는 커널 소프트웨어 계층으로 여러 종류의 파일시스템을 표현할 수 있는 공통 파일 모델 계층이다. VFS 아래에 존재하는 각각의 파일시스템들은 물리적인 구성을 VFS의 공통 파일 모델로 변환하여 접근한다. 그러므로 리눅스 커널은 read()나 ioctl()과 같은 연산을 처리하는 특정 함수를 직접 고정시켜(hardcode) 놓을 수 없다. 대신 각 연산에 대해 포인터를 사용해야 한다. 포인터는 접근할 특정 파일시스템을 위해 적절한 함수를 가리킨다.

VFS는 모든 파일시스템에 대한 공통 인터페이스를 제공하는 일 외에도 시스템 성능과 관련한 중요한 역할을 한다. 일반적으로 디스크 캐시는 보통 디스크에 저장하는 정보를 램에 저장하여 프로세스에서 이 자료에 빠르게 접근하도

록 하는 소프트웨어 메커니즘이다. 리눅스는 버퍼 캐시, 페이지 캐시 같은 디스크 캐시가 있는데 이들은 리눅스가 디스크 접근을 최대한 감소시켜 시스템 성능을 높이는 기술들이다.

버퍼 캐시는 프로세스에 비해 상대적으로 느린 디스크가 데이터를 읽고 쓰는 동안 프로세스를 자유롭게 놓아주기 위해 디스크의 데이터를 버퍼로 저장하기 위한 캐시이다. 페이지 캐시는 정규 파일의 데이터를 포함한 페이지 프레임을 저장하는 디스크 캐시이고, 디스크에 있는 파일에 대한 빠른 접근을 가능하게 한다. 메모리 매핑된 파일은 한번에 한 페이지씩 읽혀지고, <그림 2>와 같이 해시 체이닝(hash chaining) 기법을 이용하여 이들 페이지들을 페이지 캐시에 저장한다.

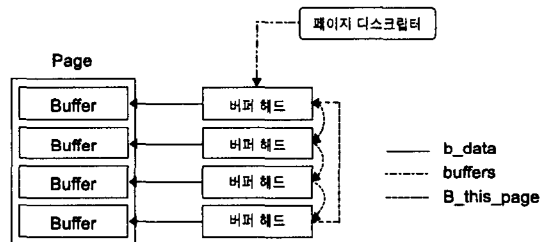


〈그림 2〉 리눅스 페이지 캐시

리눅스에서 각각의 파일은 VFS inode 자료구조에 의해서 식별된다. 각각의 VFS inode는 유일하고 오직 하나의 파일을 식별하도록 해준다. 페이지 테이블에서 인덱스(index)는 파일의 VFS inode와 그 파일에서의 오프셋(offset)으로부터 유도된다.

페이지를 메모리 매핑된 파일에서 읽을 때, 페이지 캐시를 통해 읽게 된다. 페이지가 캐시에 있으면, 그 페이지를 나타내는 mem_map_t 자료구조에 대한 포인터가 페이지 폴트 처리 코드로 되돌려진다. 캐시에 없다면 이미지를 갖고 있는 파일 시스템으로부터 페이지를 메모리로 가져와야 한다. 리눅스는 물리적 페이지를 할당하고 디스크상의 파일로부터 페이지를 읽어 들인다. 시간이 흘러 이미지를 읽고 실행함에 따라 페이지 캐시가 증가하게 된다. 더 이상 페이지가 필요 없게 되면, 캐시로부터 제거된다. 리눅스가 메모리를 사용해 나감에 따라 물리적 페이지가 부족해지기 시작하고, 이때 리눅스는 페이지 캐시의 크기를 줄인다.

리눅스에서 파일 접근은 페이지 단위라 할지라도 실제 블록장치와의 연산은 적절한 블록장치 핸들러로 한번에 한 블록씩 데이터를 전송한다. 바꾸어 말하면 페이지 입출력 연산은 페이지를 포함하는 페이지 프레임을 버퍼 그룹으로 본다. <그림 3>은 페이지 프레임과 버퍼와의 관계를 보여준다.



〈그림 3〉 버퍼 캐시와 페이지 캐시

버퍼 그룹에 들어있는 버퍼의 수는 사용된 블록 크기에 따라 달라진다. <그림 2>처럼 4KB의 페이지 프레임이 있는 경우 블록 크기가 1024라면 1KB 버퍼 네 개를 포함하고, 블록 크기가 4096이라면 4KB 버퍼 하나를 포함한다. 이 페이지 프레임을 사용하기 위해서는 구성되는 여러 버퍼의 연산이 모두 완료가 되어야 한다.

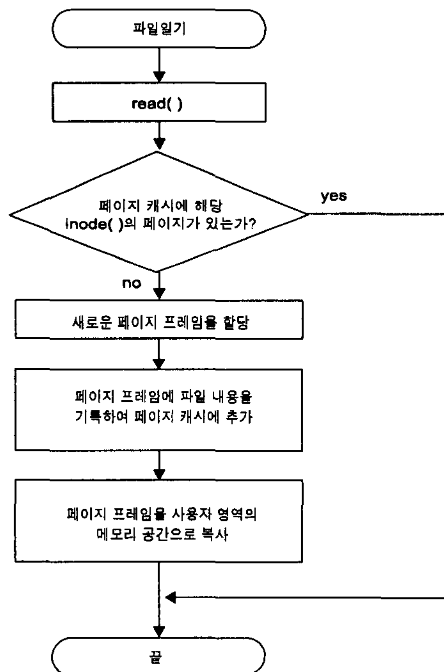
이와 같이, 하드디스크와 같은 저속의 저장장치를 사용하는 시스템에서 리눅스는 성능향상을 위해 디스크 캐시를 지원한다. 임베디드 시스템은 대부분 저장장치로 플래시 메모리를 사용하는데, 플래시 메모리의 read 성능은 램과 유사하다[1]. 플래시 메모리를 파일시스템으로 사용

하는 임베디드 시스템에서 기존의 캐싱 방법을 사용하는 것은 플래시 메모리의 특징을 반영하지 못한 것이다. 본 논문은 플래시 메모리의 특징을 반영한 정규 파일 접근 알고리즘을 제안한다. 제안된 정규 파일 접근 알고리즘은 캐싱 기능을 축약하여 빠른 파일 접근과 축약된 캐시 메모리를 가용 공간으로 활용하는 방법을 제시한다.

3. 임베디드 시스템에서 정규파일 접근 방법

3.1 리눅스의 정규파일 접근

리눅스의 정규파일 읽기는 페이지 기반이다. 커널은 언제나 전체 페이지 데이터를 한번에 전송한다. 프로세스가 몇 바이트를 얻기 위해 read() 시스템 콜을 호출하였고, read()는 generic_file_read()를 호출한다. generic_file_read()는 do_generic_file_read()를 호출하여 원하는 inode와 index를 사용하여 해당 파일의 페이지 캐시가 있는지 검사한다. 그 데이터가 램에 없으면 커널은 새로운 페이지 프레임을 할당하고, readpage()에서 할당된 페이지를 정규 파일의 적당한 부분으로 채운 다음 이 페이지를 페이지 캐시에 추가한다. 마지막으로 __copy_to_user()를 호출하여 요청한 바이트의 데이터를 프로세스 주소 공간으로 복사한다.



〈그림 4〉 리눅스의 파일 READ

<그림 4>는 리눅스의 정규파일 READ 과정을 나타낸 것이다.

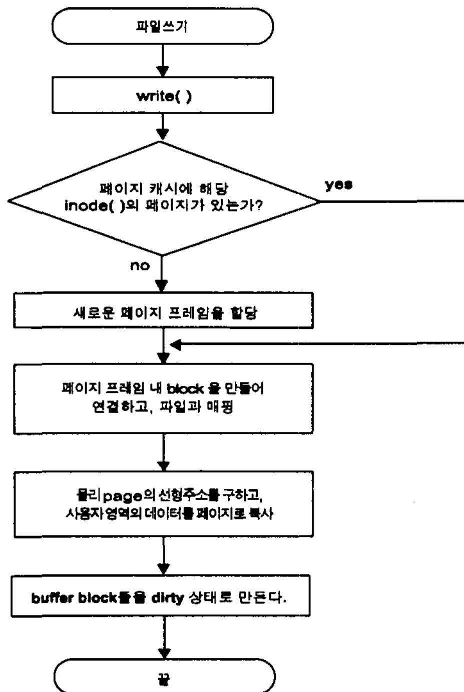
반대로 파일 WRITE 연산은 write() 시스템 콜이 데이터를 호출하는 프로세스 사용자 모드의 주소공간에서 커널 자료구조로 이동하고, 다시 디스크로 이동한다. 파일 객체의 write()는 각 파일 시스템 유형에 따라 특수한 WRITE 연산을 정의하도록 허용한다. 리눅스 커널 2.4에서는 generic_file_write()를 호출하여 리눅스 커널 2.2에서와는 달리 페이지 단위로 WRITE를 한다[4].

generic_file_write()는 __grab_cache_page()를 호출하고, 이 함수는 파일과 매핑된 페이지가 있는지 hash_table에서 찾아보고 없으면 새로운 페이지를 만들어 반환한다. prepare_write()는 페이지 내 buffer block을 만들어 연결하고, 각 버퍼를 get_block()에 의해 파일과 매핑시킨다. 물리페이지에 대한 커널 선형 주소를 구하고 파일에 써넣을 사용자 영역 버퍼에서 데이터를 읽어 들여 커널 공간에 복사한다. commit_write()는 buffer block들을 dirty 상태로 만든다. 향후 매핑된 버퍼는 bdflush 쓰레드에 의해 디스크에 쓰여진다.

<그림 5>는 리눅스의 정규파일 WRITE 과정이다.

리눅스는 파일에 대한 작업(읽기, 쓰기)을 할

때는 버퍼 캐시처럼 블록단위의 캐시가 아닌 페이지 단위의 캐시를 사용한다. 하지만 파일을 읽어 들일 때 페이지 캐시도 페이지를 블록으로 나눈 후 1블록씩 읽어 들인 후, 페이지 단위로 페이지 캐시에 보관되어진다. 이러한 디스크 캐시는 디스크 기반 파일시스템에서 좋은 성능을 보여준다.

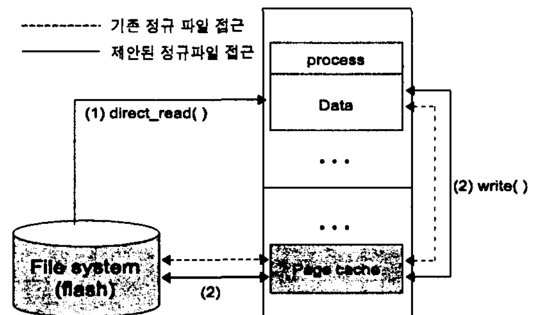


〈그림 5〉 리눅스의 정규파일 WRITE

3.2 임베디드 시스템에서 파일접근 알고리즘

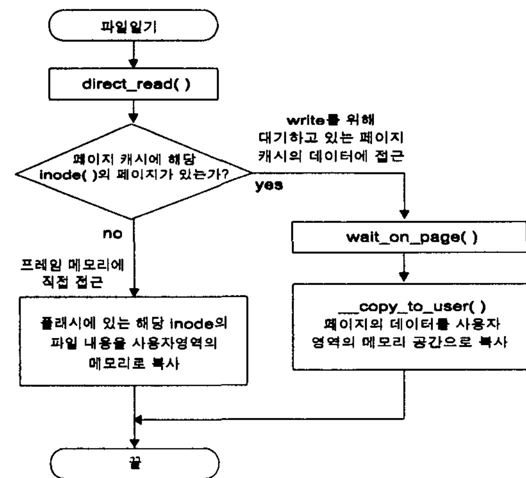
<그림 6>에서 보듯이, 리눅스는 프로세스가 파일에 접근할 때, 항상 페이지 캐시에서 READ하거나 페이지 캐시에 WRITE 한다. 하지만 제안하는 알고리즘에서는 READ는 플래시 메모리에서 직접 읽어가고, WRITE는 기존과 동일하게 페이지 캐시에 기록한다.

리눅스의 파일을 READ할 때, <그림 6>의 점선부분 흐름에서처럼 해당 inode에 대한 페이지가 없을 경우, 플래시에서 페이지 캐시로 파일의 내용을 복사하고 다시 프로세스의 메모리 공간으로 페이지를 복사한다. 하지만 제안하는 파일접근 알고리즘은 <그림 6>의 (1)과 같이 플래시 메모리로부터 직접 사용자 프로세스 메모리 공간으로 복사함으로써, 적어도 페이지 캐싱 시간만큼 데이터 접근 시간을 단축시킬 수 있다. 이는 플래시 메모리의 READ 성능이 램의 READ 성능과 유사하므로 재사용하는 페이지를 플래시 메모리에서 읽는 시간과 메모리의 페이지 캐시에서 읽는 시간이 유사한 결과를 보여주기 때문이다.



〈그림 6〉 정규 파일접근 구조

<그림 7>은 제안된 파일접근의 READ에 대한 흐름도이다.



〈그림 7〉 제안된 파일접근의 READ

<그림 7>에서 보듯이 기존의 read()를 호출하는 대신 direct_read() 시스템콜을 등록하여 정규파일을 READ할 때 이 함수를 호출한다. 해당 inode의 페이지가 페이지 캐시에 있으면, 해당 페이지가 WRITE 연산 중이면 연산이 끝날 때가지 대기한다. WRITE 연산이 끝나거나 해당 페이지를 읽을 수 있으면 기존의 file_read_actor() 함수를 호출하여 해당 페이지의 내용을 사용자 공간으로 복사한다. 반면에, 해당 inode의 페이지가 페이지 캐시에 없으면 플래시에 직접 그 파일을 block 단위로 사용자 공간에 읽어 들인다. 기존에 readpage() 함수는 새로 할당된 페이지에 해당 inode의 내용을 READ하는 함수이다. 제안된 알고리즘에서는 readpage() 함수 대신 플래시에서 직접 사용자 영역으로 데이터를 블록단위로 READ하는 direct_get_block()를 사용하여 파일을 READ하는데 기존의 _copy_to_user() 만큼의 시간을 단축시켰다.

다음은 direct_read()의 알고리즘이다.

```
// system call : direct_file_read
ssize_t sys_direct_file_read(fd, buf, count)
        // count : 페이지 크기의 배수
{
// 파일의 inode 및 파일에 대한 정보를 읽어옴
  file_p = fget(fd) ;
  inode = file_p->f_dentry->d_inode ;

  if(page = find_page(inode)) {
    //해당 페이지가 페이지 캐시에 있는지 검사
    wait_on_page(page) ;
    //해당 페이지를 읽을 수 있을 때까지 대기

    //기존 함수 호출 : __copy_to_user()
    ret = file_read_actor(buf, page, 0, count) ;
    return ret ;
  }

  // Not in page_cache
  offset = 0 ;
  do {
    // direct read I/O 수행.
    ret = direct_get_block(file_p,
                      buf + offset, block_size) ;
    if (ret == ERROR)
      break;
    offset += block_size ;
    read_count += ret ;
  } while(read_count != count) ;
  return ret ;
}
```

플래시 메모리는 READ 속도는 램과 비슷하지만, WRITE는 지움정책으로 인해 상대적으로 느리다. 따라서 파일 WRITE는 리눅스에 제공하는 generic_file_write()를 따른다. 단, 최신 데이터에 대한 동기를 맞추기 위해 해당 파일에 대한 WRITE 연산이 진행되는 동안 READ 연산은 READ 알고리즘에 의해 lock된다.

4. 성능 평가

제안한 정규파일 접근에서의 READ는 플래시 메모리에서 데이터를 직접 읽음으로서 페이지 캐시의 메모리 공간의 일부를 가용 메모리 공간으로 확보할 수 있다. 또한 페이지가 캐시에 없을 경우 필요한 페이지 공간을 할당받아 처리하는 시간과 페이지의 내용을 사용자 공간으로 복사하는 시간이 절약된다.

디스크 블록 장치의 경우 읽기 성능의 향상을 위해 인접한 블록을 미리 읽는 알고리즘이 사용된다. 이는 디스크 기반의 블록 장치의 구조가 회전식 디스크에 기록된 블록을 읽는 것으로 최초 접근 시간과, 디스크 헤드가 움직이는 횟수를 줄이기 위한 것이다. 그러나 명확하게 요청되지 않은 블록을 미리 읽는 것은 복잡한 알고리즘이 요구된다. 또한 파일시스템에 임의 접근의 경우에는 아무런 도움이 되지도 않고, 캐시

에 쓸모없는 정보를 저장하여 공간을 낭비하기 때문에 장단점이 있다. 제안하는 파일 접근 알고리즘은 플래시 메모리를 기반으로 하기 때문에 충분한 READ 성능을 가지므로 미리 읽기를 하지 않고도 그에 부합되는 성능을 가지므로 미리 읽기에 사용되는 캐시 공간과 연산 시간을 다른 프로세스가 사용할 수 있다.

제안된 파일 접근 알고리즘이 적용할 임베디드 시스템의 구조는 플래시 메모리와 램의 속도가 유사한 성능을 가진다. 기존의 알고리즘은 미리읽기를 하지 않는다고 가정하였고, 리눅스의 페이지 캐시는 동적으로 할당받지만 계산의 편의성을 위해 일정한 크기로 고정하였다.

식 (1)은 기존 방법의 알고리즘을 적용할 때, 페이지 캐시에 없는 데이터를 읽는 처리 시간이고 식 (2)는 페이지 캐시에 데이터가 있을 때의 처리시간이다.

$$T = T_{flash_access} * page_size + T_{ram_access} * page_size + __copy_to_user(page) \quad (1)$$

$$T = T_{ram_access} * page_size + T_{ram_access} * page_size \quad (2)$$

다음은 제안된 메모리 관리 기법의 알고리즘을 적용할 때의 처리 시간이다.

$$T = T_{flash_access} * page_size + T_{ram_access} * page_size \quad (3)$$

식 (1)의 시간을 분석하면, 플래시 파일시스템의 파일을 읽기 위해 플래시 메모리에서 페이지 단위로 READ를 수행한다. 읽은 페이지를 램의 페이지 캐시에 기록하고, 다시 __copy_to_user() 함수를 이용하여 커널 영역에서 사용자 영역으로 복사를 수행한다. 따라서 플래시 메모리를 READ 하기 위해 한번 접근하고, 페이지 캐시에 WRITE하고 다시 READ하여 사용자 영역으로 복사(WRITE) 함으로써 램을 3번 접근한다. 같은 방법으로 식 (2)는 페이지 캐시에서 READ하여 사용자 영역으로 WRITE하는 것이므로 램에 2번 접근한다. 식 (3)은 플래시 메모리의 파일을 READ하여 사용자 영역에 기록한다. 그러므로 식 (1), 식 (2), 식 (3)은 각각

$$T = T_{flash_access} + T_{ram_access} * 3 \quad (4)$$

$$T = T_{ram_access} * 2 \quad (5)$$

$$T = T_{flash_access} + T_{ram_access} \quad (6)$$

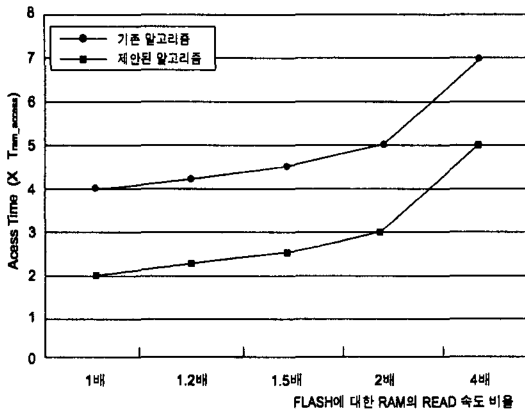
와 같다. 이 식들을 이용하여 기존의 알고리즘과 제안된 알고리즘의 성능을 3가지 상황으로 비교해 본다.

첫 번째, 프로세스가 파일을 필요로 할 때 한번만 읽을 경우, 즉 해당 파일에 대한 접근이 가끔일 때 시스템이 다른 작업을 위해 페이지 캐시에서 파일의 페이지를 해제할 것이다. 때문에 기존의 알고리즘은 파일에 접근할 때마다 페이지 캐시로 다시 READ해야 하므로 식 (4)를 적용하고, 제안된 알고리즘은 식 (6)을 적용할 수 있다. <그림 8>은 플래시에 대한 램의 READ 속도 비율에 따른 전체 소요 시간을 나타낸 것으로, 세로축은 소요 시간으로 램의 READ 속도인 T_{ram_access}를 곱한 시간이다. 만일 램과 플래시 메모리의 READ 시간이 유사한 경우라면 제안된 알고리즘이 2배 정도 빠른 성능을 보여준다. 플래시 메모리가 램에 비해 속도가 느려도 기존 알고리즘의 페이지 캐시에서 사용자 공간으로 데이터를 복사하는 시간($2T_{ram_access}$) 만큼 제안된 알고리즘이 데이터에 더 빨리 접근함을 알 수 있다.

〈표 2〉 페이지 캐시에 없는 파일을 READ할 때 성능 비교

(단위 : Taccess)

플래시에 대한 램의 속도	1배	1.2배	1.5배	2배	4배
기존 알고리즘	4	4.2	4.5	5	7
제안된 알고리즘	2	2.2	2.5	3	5



〈그림 8〉 페이지 캐시에 없는 파일을 READ할 경우

두 번째, 한번 접근된 파일을 계속하여 접근하는 경우이다. 빈번하게 접근되는 파일은 페이지 캐시에 유지되고 기존의 알고리즘은 페이지 캐시에서 데이터를 READ한다. 제안된 알고리즘은 파일 접근 연산을 할 때마다 플래시 메모리에서 데이터를 READ한다. 기존 알고리즘의 경우 처음에 플래시 파일시스템으로부터 읽게 되므로 식 (4)를 적용하고, 이후의 접근에 대해 식 (5)를 적용한다. <표 3>는 플래시 메모리의 READ 성능과 램의 READ 성능을 다양하게 하여 동일한 파일의 접근 횟수에 따른 성능 비교를 나타낸 것이다.

<그림 9>는 <표 3>에서 두 번, 다섯 번 READ할 경우를 그래프로 나타낸 것이다. 한 번 READ할 때는 <그림 8>에서 보여지는 것과 같다.

<그림 9>에서 알 수 있듯이, 같은 데이터를 여러 번 접근할 경우라도 플래시 메모리와 램의 READ 속도가 비슷하다면 제안 알고리즘의 성능이 더 우수함을 알 수 있다. 하지만, 플래시 메모리가 램보다 READ 속도가 1.2배 이상 느릴 경우 페이지 캐시에 해당 데이터를 유지하고 있는 기존 알고리즘이 데이터에 더 빨리 접근한다. 임베디드 시스템에 사용 되는 플래시 메모리와 램이 하나의 칩(29F3204C3)으로 구성되는 경우가 있는데, 이러한 경우에 같은 속도의 READ 성능으로 제안된 알고리즘을 사용할 경우에 성능 향상이 기대된다. 일부 고속으로 동작하는 임베디드 시스템의 경우에 SDRAM을 사용하는데, SDRAM의 READ 성능은 플래시 메모리의 READ 성능에 비해 2배 정도의 속도를 가지므로 본 알고리즘을 적용하기에 적절하지 않다. 그러나 비교적 저속으로 동작하는 임베디드 시스템의 경우에는 램의 READ 속도가 플래시 메모리의 READ 속도와 유사하므로 본 알고리즘을 적용하였을 때 성능 향상이 기대된다.

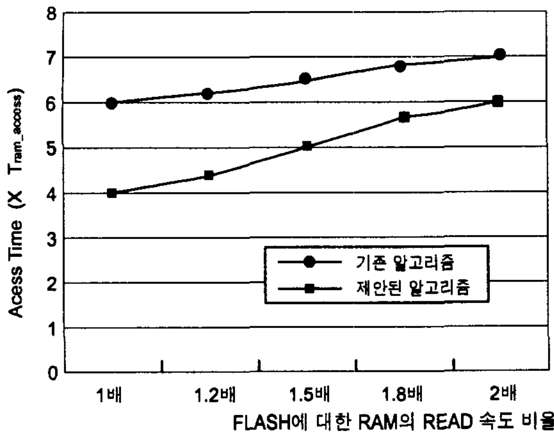
세 번째, 프로세스가 여러 개의 파일을 연속적으로 요구하는 경우이다. 사용된 파일의 수는 5개이며, 각각의 크기는 100k, 200k, 300k, 400k, 500k 이고, 1번에서 5번까지 연속된 번호를 부여했다. 캐시의 크기는 1000k로 제한하였고, 접근한 파일의 용량이 1000k가 넘을 경우 LRU 정책에 따라 오래된 파일의 페이지 캐시부터 삭제된다. 접근한 파일의 순서가 다음과 같다.

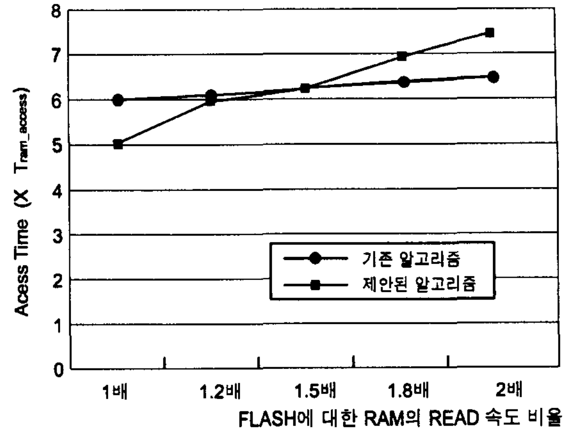
1, 3, 2, 1, 4, 3, 2, 5, 2, 5, 2, 1, 4, 4, 1, 5, 1, 4, 1, 1

페이지 캐시의 상태를 보면 1번, 3번, 2번의 파일을 접근하게 되면 식 (4)에 의해 플래시 파

〈표 3〉 동일한 파일을 접근 횟수에 따른 성능 비교 (× $T_{ram\ access}$)

플래시에 대한 램의 속도	1배		1.2배		1.5배		1.8배		2배	
	기존	제안	기존	제안	기존	제안	기존	제안	기존	제안
1번 READ	4	2	4.2	2.2	4.5	2.5	4.8	2.8	5	3
2번 READ	6	4	6.2	4.4	6.5	5	6.8	5.6	7	6
5번 READ	12	10	12.2	12	12.5	12.5	12.8	14.0	13	15





<그림 9> 동일한 파일을 READ할 경우 (a) 2번 READ. (b) 5번 READ

일시스템으로부터 파일을 읽어 파일을 페이지 캐시에 유지하고, 프로세스로 넘겨준다. 다음의 1번 파일의 접근은 페이지 캐시에 유지된 내용을 프로세스에게 넘겨주며, 다음의 4번 파일은 플래시 파일시스템으로부터 읽게 된다. 8번째 5번 파일의 접근을 수행할 때, 페이지 캐시는 1번에서 4번 파일의 용량만큼 1000k를 모두 사용 중이다. 5번 파일을 페이지 캐시에 넣기 위해 오래된 순서인 1번과 4번 파일의 페이지를 삭제한다. 같은 방법으로 나머지 부분을 수행할 경우 기존 알고리즘의 페이지 캐시의 이용은 다음과 같다.

1, 3, 2, 1, 4, 3, 2,
① 5, 2, 5, 2, ② 1,
③ 4, 4, 1, ④ 5, 1, 4, 1, 1

① 페이지 캐시에서 파일 1, 4를 삭제
② 페이지 캐시에서 파일 3을 삭제
③ 페이지 캐시에서 파일 5를 삭제
④ 페이지 캐시에서 파일 2를 삭제

위의 결과에서 밑줄 친 파일들은 플래시 파일시스템으로부터 읽은 파일들로 식 (4)를 적용한 것이고, 나머지는 페이지 캐시에서 읽은 파일들로 식 (5)를 적용하였다. 따라서 기존의 알고리즘의 결과는 다음과 같다.

처음에 플래시의 데이터를 READ할 때는 식 (4)를 적용하여,

$$T = 2500k * (T_{flash_access} + 3 * T_{ram_access})$$
$$= 2500k * T_{flash_access} + 7500k * T_{ram_access}$$

이 되고, 이미 READ된 데이터를 페이지 캐시에서 읽을 때는 [식 5]를 적용하여,

$$T = 2700k *(2 * T_{ram_access}) = 5400k * T_{ram_access}$$

이 된다. 그러므로 이 예제에서 사용되는 파일을 위의 순서대로 읽을 때는

$$T = 2500k * T_{flash_access} + 12900k * T_{ram_access}$$

이다. 제안된 알고리즘은 페이지 캐시를 사용하지 않고 모두 플래시 파일시스템으로부터 읽으므로 [식 6]을 적용하면

$$T = 5200k * T_{flash_access} + 5200k * T_{ram_access}$$

가 된다. 플래시와 램의 속도 비율에 따른 접근 시간을 <표 4>에 나타냈다. 이 결과에 램의 접근 시간인 T_{ram_access}를 곱한 것이 접근 시간이

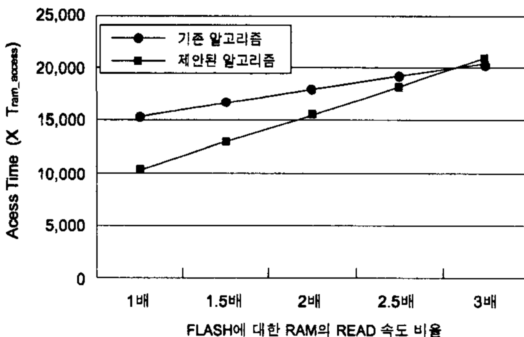
〈표 4〉 서로 다른 파일을 연속적으로 접근할 때 성능 비교

플래시에 대한 램의 속도	1배	1.5배	2배	2.5배	3배
기존 알고리즘	15,400k	16,650k	17,900k	19,150k	20,400k
제안된 알고리즘	10,400k	13,000k	15,600k	18,200k	20,800k

〈표 5〉 서로 다른 파일을 연속적으로 접근할 때 캐시 메모리 사용량(단위 : K)

파일 READ 순서	1	3	2	1	4	3	2	①	5	2	5	2	②	...
기존 알고리즘	100	400	600	600	1000	1000	1000	500	1000	1000	1000	1000	700	...
제안 알고리즘	0	0	0	0	0	0	0		0	0	0	0	0	...

된다. <그림 10>은 <표 4>를 그래프로 나타낸 것이다.사용된 예제에서는 플래시 메모리의 READ 속도가 램의 READ 속도의 약 2.5배 정도 되는 시점에서 기존 알고리즘과 제안된 알고리즘의 성능이 비슷해진다. 새로운 데이터를 연속으로 READ할수록 제안된 알고리즘의 데이터 접근 시간이 더욱 향상되고, 플래시 메모리의 READ 속도가 램의 READ 속도와 비슷할 때 우수한 성능을 보인다. 캐시에 있는 데이터를 더 많이 접근할수록 기존 알고리즘이 우수함은 두 번째 예제에서 보였다.



〈그림 10〉 서로 다른 파일을 연속적으로 접근할 경우

<표 5>는 WRITE 연산이 없다고 가정하였을 때, 위의 예제에서의 캐시 사용량을 나타낸다. 제안된 알고리즘은 데이터 READ 시 직접 플래시 메모리를 접근하므로 페이지 캐싱를 하지 않기 때문에 캐시 사용량이 0k가 된다. 반면, 기존 알고리즘은 서로 다른 파일을 연속적으로 READ할 경우 1000k의 캐싱 공간이 모두 사용되고 있으면, LRU 정책에 따라 오래된 데이터를 캐시에서 해제하고 요구되는 데이터를 캐시로 READ한다. 이와 같이, 제안된 알고리즘은 캐싱를 하지 않는 만큼 메모리 공간을 절약할 수 있고, 기존 것처럼 캐시된 데이터에 대한 추가적 접근이 없으므로 램의 접근 횟수도 감소한다.

예제들에서 알 수 있듯이, 제안하는 정규파일 접근 알고리즘은 플래시 메모리와 램의 READ 속도가 비슷하고, 파일 READ 연산이 비교적 많은 시스템에서 데이터의 접근 시간이나 메모리 사용면에서 높은 성능을 나타낸다. 게임에 사용되는 많은 이미지 파일, 데이터베이스의 레코드 등을 READ하는 데 제안된 정규파일 접근 알고리즘을 응용할 수 있다. 또한, 전력소모가 낮은 플래시 메모리를 직접 READ하는 것은 램에 접근 횟수가 감소하고 그만큼 에너지 소비를 하지 않는 등 시스템 성능 향상에 크게 기여한다.

5. 결론 및 추후과제

리눅스 정규파일 접근은 데이터를 읽을 때, 데이터가 페이지 캐시에 없으면 새로운 페이지를 할당하고 저장장치에서 페이지로 데이터를 읽어온 후 그 페이지를 다시 사용자 영역의 메

모리로 복사한다. 따라서 페이지 캐싱을 통해 느린 저장장치와의 데이터 전송 속도를 향상시킬 수 있다. 하지만, 이러한 기법은 플래시 메모리를 저장장치로 사용하는 임베디드 시스템에선 오히려 제한된 자원을 낭비하게 된다. 디스크와 같은 느린 저장장치에서 데이터를 직접 READ 하는 것은 시스템 성능을 악화시키지만, 플래시 메모리는 READ 연산 속도가 램과 유사하기 때문에 플래시 메모리의 데이터를 직접 READ하는 것은 기존 페이지 캐싱에 사용되던 메모리 자원을 시스템 성능향상을 위해 사용할 수 있다. 본 논문에서는 정규파일 접근을 플래시 메모리에서 직접 READ하는 알고리즘을 제안하였고, 기존 방법보다 메모리로의 접근 횟수를 많이 줄이고, 그로인해 시스템의 자원이 절약됨을 보였다. 임베디드 시스템은 주로 루트파일 시스템으로 램 디스크를 사용하는데, 램 디스크에서도 같은 방법을 사용하여 메모리 효율을 높일 수 있다. 향후 연구과제는 플래시 메모리에서 운영되는 여러 다른 파일시스템에 제안된 알고리즘을 적용하여 그 성능을 검증하고, 램(페이지)과 저장장치(블록)의 접근 단위를 동기화하여 플래시를 램처럼 활용하는 연구가 필요하다.

참 고 문 헌

[1] 김정기, 박승민, 김채규, "임베디드 플래시 파일 시스템", *정보처리*, 제9권 제1호, 2001년 1월, pp. 43-49.

[2] 박성호, 정기동, "내장형 운영체제의 파일 시스템 - Flash File System", *정보처리*, 제9권 제3호, 2002년 5월, pp. 103-108.

[3] A. Kawaguchi, S. Nishioka and H. Motoda, "A Flash-Memory Based File System," Proc. of the 1995 USENIX Technical Conference, Jan., 1995.

[4] Daniel P.Bovet, Marco Cesati, "Understanding the LINUX KERNEL", O'Reilly, January 2001.

[5] Linux Kernel 2.4.2 source.

■ 저자소개

이 은 주

현재 (주)세인정보통신에서 멀티미디어 임베디드 소프트웨어 개발자로 일하고 있고, 한밭대학교 정보통신 대학원에서 공학석사(2004)를 취득했다. 주요 관심분야는 임베디드 리눅스 시스템, 멀티미디어 데이터 처리, 플래시 메모리 카드의 인터페이스 개발, 정보가전 등이다.

박 현 주

현재 한밭대학교 정보통신컴퓨터공학부 조교수로 재직 중이다. 서울시립대학교 전산통계학사(1990), 서울대학교 계산통계학 석사(1992), 박사학위(1997)를 취득하였다. 주요 연구 분야는 파일 시스템, 임베디드 소프트웨어, 모바일 데이터베이스(Mobile Database) 등이다.

◈ 이 논문은 2003년 9월 3일 접수하여 1차 수정을 거쳐 2004년 1월 5일 게재확정되었습니다.