Directory, inode 및 file을 디스크의 인접한 공간에 저장하는 파일 시스템

조준우[0] 황주영 김경호 임승호 박규호
한국 과학기술원 전자전산학과 컴퓨터 공학 연구실
jwc@core.kaist.ac.kr

File system which store directory, inode, file adjacently in disk

Cho, Joon-Woo[0] Hwang, Joo-Young Kim, Kyung-Ho Lim, Seung-Ho Park, Kyu Ho
KAIST EECS(Electrical Engineering and Computer Science) CORE Lab.

요 약

현재의 디스크 시스템 성능의 병목이 되는 부분은 disk 헤드의 느린 이동 속도이다. 일단 원하는 데이터가 있는 곳으로 헤드가 이동하고 나면, data는 초당 수십 MB의 속도로 memory에 전송될 수 있다. 만약에 작은 크기의 파일들이 디스크의 물리적인 block에 산재해 있다면, 이 파일들의 위치로 헤드가 이동하는 데에 많은 시간이 걸릴 것이다. 일반적으로 **한 디렉토리 안에 들어있는 파일**들은 비슷한 시간에 읽혀질 가능성이 크므로 **이 파일들을 디스크의 인접한 block에 할당**해 준다면 파일에 헤드가 접근하는 데 걸리는 시간을 크게 줄일 수 있을 것이다. 또 UNIX 계열의 OS가 파일을 관리하기 위해 사용하는 **inode도 한 디렉토리 안의 파일을 가리키는 것들을 모두 인접하게 위치**시킨다면 이 inode들을 찾기 위해 disk의 헤드가 움직이는 횟수를 줄일 수 있을 것이다. 이 두 가지 방법을 **Linux OS를 platform**으로 하여 구현하였다. 실험 결과 이러한 방법을 사용한 파일 시스템은 이전의 파일 시스템에 비해서 최고 44%까지 더 높은 성능을 보임을 알 수 있었다.

1. Introduction

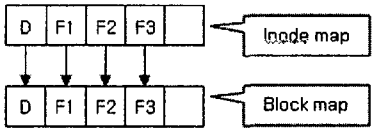
현재의 디스크 시스템에서 병목이 되는 부분은 disk access time이다. 이것은 특히 디스크에 작은 파일들이 많이 있을 경우 문제가 있다. 이 작은 파일들이 디스크의 여러 부분에 흩어져 있다면 파일들이 위치한 disk block을 찾아가는 데에 많은 시간이 걸리게 되는 것이다. 이러한 단점을 극복하기 위해서 현대 disk system들은 디스크 캐쉬를 두어서 인접한 파일들을 한꺼번에 읽어온다. 만약 이렇게 읽어오는 주변 file들이 인접해 있다면 disk performance는 더욱 증가될 수 있을 것이다. 이런 인접성을 높이고자 아래의 두 방법을 사용하려고 한다.

1.1. Block merge

한 디렉토리 안에 들어 있는 file들은 일반적으로 비슷한 시간에 불려질 확률이 높다. 따라서 이 file들을 인접한 디스크 block에 저장해 놓아서 전체 file들을 한번에 읽고 쓴다면 디스크 위에서 각각의 file을 찾아 움직이는 불필요하고 오래 걸리는 disk seek를 줄일 수 있을 것이다.

1.2. Inode merge

inode 역시 한 디렉토리 안의 파일을 가리키는 inode들을 인접하게 위치시킴으로써 disk 위에서 inode를 찾아 움직이는 불필요한 disk access time 을 줄일 수 있을 것이다.



위의 두 알고리즘을 적용한 후에 disk layout 모습

2. Implementation

참고한 논문(참고문헌 ①)에서는 exokernel이라는 OS를 platform으로 하여 구현하였다. 그러나 이 OS는 내부 구성이 매우 간단하고 일반적으로 사용되는 OS가 아니다. 따라서 실제 컴퓨터 환경에서 구현했을 때의 결과는 논문에서의 결과와 다소 다를 수가 있다. 본 논문에서는 일반적으로 사용되고 있는 **Linux OS를 platform으로** 하여 구현하고 실험하였다. 또한 구현의 간결함을 위해 Linux에서 지원하는 가장 간단한 형태의 파일 시스템인 MINIX 파일 시스템의 code를 수정했다.

다음 절에서 구현 과정을 구체적으로 제시한다.

논문에 나오는 모든 파일 경로는 중복을 피하기 위해 앞 부분에 /usr/src/linux/이라는 경로명을 생략한 것이다.

2.1. Block allocation

한 디렉토리 안의 파일들을 모두 인접한 block들에 넣

어주어야 한다. minix file system에서는 새로운 block이 필요한 모든 경우에 **minix_new_block 함수** (/fs/minix/bitmap.c)를 호출한다. 원래 이 함수는 block map의 앞에서부터 탐색해가면서 비어 있는 block이 있다면 바로 그 block을 필요한 곳에 할당한다. 본 논문에서 제안된 알고리즘을 구현하기 위해 각각의 경우에 따라 아래와 같은 방법을 사용하여 함수를 수정하였다.

2.1.1. 자주 사용하는 함수, 추가되는 자료 구조

* 첫 번째로 bit가 0인 memory 위치를 알려주는 함수

int find_first_zero_bit (void * addr, unsigned size);

addr 주소 이후 몇 번째 bit에서 0이 처음 나왔는지를 알려준다.

* kernel memory allocation 함수

void * kmalloc (size_t size, int flags);

* 디렉토리 크기가 커질 때를 위한 자료구조

디렉토리가 처음에 할당된 block 혹은 inode의 개수보다 많은 양의 block, inode를 필요로 하는 경우 그 디렉토리를 위해 새 block, inode group을 할당해 주고 그 정보를 가지고 있어야 한다. 이러한 정보는 가변적으로 생겨나는 것이므로 Circular한 **linked list** 구조를 이용한다.

```
struct empty_block_node{
 unsigned int block;   // 새로 할당된 block group의 첫 block
 struct empty_block_node *next;
}
```

2.1.2. directory용 block 할당

directory를 새로 생성할 때 128개(필요한 경우 변경할 수 있다.)의 block을 하나의 단위로 해서 그 block에 할당해 준다.

```
for( k=0; k<1024; k=k+16){
 if( (j=minix_find_first_zero_bit( (bh->b_data)+k, 1) ) < 1 ){
  start = kmalloc( sizeof( struct empty_block_node),
          GFP_KERNEL);
  // block linked list의 처음을 위해 메모리를 할당
  tmp_inode->start->next = tmp_inode->start;
  // circular구조의 시작(처음은 다시 처음을 가리킴.)
  start->block = 찾은 block group의 첫 block 번호
  break; // 새 block group 찾으면 loop 탈출
 }
}
```

(bh->b_data)+k에서 k를 16씩 증가시키므로 minix_find_first_block 함수에서 128bit(k는 8bit)씩 빈 block을 찾아 나가고 따라서 하나의 block group은 128개의 block을 가지게 된다.

block을 찾으면 start->block에 block 숫자를 저장한다.

2.1.3. 파일용 block 할당

2.2.3.1. 이미 할당된 block group으로 충분한 경우

디렉토리 생성 시 할당된 128개의 block안에 빈자리가 있을 때 빈 자리 찾아서 사용한다.

```
k=파일이 저장되는 위치의 상위 directory의 block 숫자
t=tmp_inode->start;  // list의 시작점을 t node에 할당
do{
 if ((j=minix_find_first_zero_bit((bh->b_data)+k,128)) < 128 )
  break;      // 새 block 찾으면 loop 탈출
  t=t->next; // 이 block 단위에는 만족되는 빈 block이 없다.
             // 따라서 리스트의 다음 노드로 넘어간다.
}while(t!=tmp_inode->start); //리스트의 맨 처음까지 loop.
```

2.2.3.2 이미 할당된 block group의 block을 다 쓴 경우

디렉토리 생성 시 할당된 128개의 block을 다 썼을 때는 새로운 block group을 할당 받아서 그 데이터를 list에 저장한다.

```
for( k=0; k<1024; k=k+16){
 if ((j = minix_find_first_zero_bit( (bh->b_data)+k, 1)) < 1){
 // 여기서 받은 새 block 단위를 list 자료구조에 추가한다
        t = tmp_inode->start;
 while( (t->next)!=tmp_inode->start )  t = t->next;
 // t->next가 start될 때까지(list의 끝까지) list 검색해간다
 new_node=kmalloc(sizeof(struct          empty_block_node),
          GFP_KERNEL);
 // 새 node를 memory에 생성
 t->next = new_node; // 마지막 node의 다음이 새 node
 new_node->next = tmp_inode->start;
 new_node->block = 찾은 block group의 첫 block 번호;
 break; // 새 block 찾으면 loop 탈출
 }
}
```

2.2. Inode allocation

새로운 inode가 필요할 때 minix file system은 **minix_new_inode** 함수(/fs/minix/bitmap.c)를 호출한다. minix_new_block 함수를 수정한 것과 동일한 방법을 사용해서 minix_new_inode 함수를 고친다. 즉, 새로운 directory가 생성되면 그 directory에서 사용할 일정량의 inode group을 할당해주고 그 directory 아래에서 새로운 inode를 요구할 때(새로운 file생성)에 그 group 안의 inode를 사용한다.

2.3. directory가 지워지는 경우

해당 디렉토리에 할당된 block, inode linked list의 모든 node(memory에 할당되어 있는)를 지워(kfree()함수 사용)주어야 한다. 디렉토리는 minix_rmdir 함수(/fs/minix/namei.c)에서 지워지므로 여기에 list의 node를 지워주는 코드를 추가했다.

3. Experiment

OS: Linux Redhat 6.2

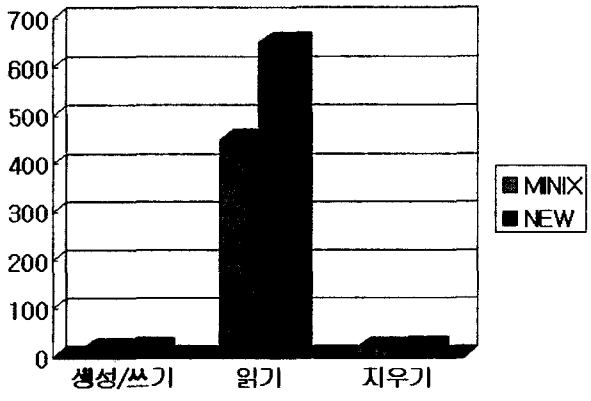
computer system: 400MHz Pentium II, 128MB memory, 1.3GB Quantum HDD

3.1. 방법

1KB짜리 file 5000개를 생성한 후 생성된 파일에 읽기, 지우기의 동작을 수행해 보았다. 또한 캐쉬가 실험에 영향을 주지 않게 하기 위해 각각의 동작 사이에 system을 rebooting시켜주었다.

본 논문에서 구현한 disk layout(디렉토리 단위로 파일 block과 inode들이 인접해 있는)과의 비교 실험을 위해 다음의 방법으로 통상 사용하는 파일 시스템에서 보이는 disk layout과 비슷한 모양의 disk layout을 생성하였다. 디렉토리 안의 파일을 반은 disk의 인접한 block에 할당하고 나머지 반은 랜덤하게 인접하지 않은 block에 할당한다. 이렇게 만든 disk layout의 파일들은 일반적으로 사용되는 file server에서보다 같은 디렉토리 안의 파일들이 더 인접한 공간에 저장된다.

3.2. 결과



단위 file/s

예상대로 새 파일 시스템이 여러 파일 오퍼레이션에서 MINIX 파일 시스템보다는 빠른 수행시간을 보인다. 파일 생성/쓰기/지우기에서는 14%의 성능 향상을 보이고, 읽기에서는 성능을 44% 향상시킬 수 있었다. 읽기는 매우 빠른 데에 비해 파일을 생성해서 기록하는 것과 파일을 지우는 것은 상대적으로 크게 느린 이유는 다음과 같다. 읽기의 경우에는 하나의 file을 읽을 때 인접한 file까지 같이 읽어서 memory에 넣어 놓으므로 각각의 file에 대한 disk access 횟수가 쓰기/지우기보다 크게 적다. 또한 실험을 할 때에 파일을 실제로 디스크에 기록하기 위해 sync()라는 시스템 콜을 사용하기 때문이다. sync()함수는 파일 뿐만 아니라 시스템 모든 영역에 걸쳐 캐쉬와 각종 구성요소를 동기화 시킨다. 이로 인한 오버헤드에 의해 이런 동작이 없는 read보다 크게 느린 수행시간을 보인다.

4. Conclusion

file에 대한 disk의 access 속도가 느리기 때문에 작은 파일들에 대한 접근이 주로 이루어지는 응용 프로그램의 경우 보통의 file system에서는 다소의 성능 감소가 나타날 수 있다. 이런 응용 프로그램에서 좋은 성능을 낼 수 있도록 한 디렉토리 안의 모든 파일들을 인접한 block, 인접한 inode에 할당해 주었다. 이런 방법을 사용한 결과 읽기에서는 최대 44%.의 성능 향상을 낼 수 있었다. 따라서 작은 파일들에 대한 빈번한 접근이 일어나는 리눅스 어플리케이션은 이 file system을 기반으로 함으로써 좀 더 좋은 성능을 보일 수 있을 것이고 한 디렉토리 안에 연관성 있는 파일들이 모여있는 리눅스 어플리케이션에서는 더욱 큰 성능 향상을 보일 수 있을 것이다.

5. References

① Gregory R. Ganger and M. Frans Kaashoek, Embedded Inodes and Explicit Grouping, USENIX, 1997 Annual Technical Conference, 1997

② Michael Beck, Harald Bohme, ... LINUX KERNEL INTERNALS, 2nd ed, Addison-Wesley, P. 148-185, 1998

③ Andrew S. Tanenbaum, Albert S.Woodhull Operating Systems: Design and Implementation, 2nd ed.,

④ 다니엘 보베이, 마르코 체사티, 리눅스 커널의 이해, 한빛 미디어, 2001