Project 2 : DGFS

파일 시스템 구현

김 영빈

기초학부

대구경북과학기술원

대구광역시 한국

요약

DGFS 라는 파일을 만들기 위해서 File Deduplication Exercise, Implement Interface of DGFS, Add Deduplication Feature, Support Large File 의 네 가지 과정을 수행하였고, 생성된 DGFS 의 파일 크기를 줄이기 위하여 추가로 알고리즘을 짰다. 파일 시스템은 super block, inode bitmap, data bitmap, inode region, hash table region, data region 으로 구성이 되어있다. 이는 create, add, remove, extract, ls 의 기능을 구현한 함수를 통해 사용할 수 있다.

1 Step 1: File Deduplication Exercise

File deduplication 을 구현하기 위해서는, 각각의 데이터가 같은지를 확인해야 한다. 이 데이터가 같은지를 구현하려면 데이터를 비교할 수단이 필요하다. 이를 각각 byte 비교 방식과 hash 비교방식으로 구현할 수 있다.

먼저, byte 비교 방식은 파일을 불러와서, 각각의 파일에 해당하는 크기의 값을 lseek(fd, 0, SEEK_END)로 알아내어 변수에 저장한다. 이 두 파일의 크기가 다르면 다르다고 하고(구현한 파일에서는 diff 라는 변수를 0 에서 1 로 해줌)같으면 내부의 값을 검사한다. for 문으로, 파일의 제일 마지막 값까지 비교하고, 같으면 remove(파일 2)와 link(파일 1, 파일 2)를 통해 linking 해준다. Diff 가 1 일경우, 해당 포멧에 맞게 출력해준다.

Hash 값을 비교하는 방식은 먼저 크기를 비교하고(틀리면 다르다 출력하고 함수 종료), 값이 같으면 openssl/md5.h 에 구현되어있는 함수들로 hash 값을 얻는다. 이 해쉬값을 얻기 위해서 각각 파일들을 문자열로 만들어주는데, malloc(fsize + 1)로 파일 사이즈보다 1 크게 만들어 준 뒤, 제일 마지막 바이트를 null 로 해주고 문자열 시작 주소를 해쉬함수에 넣어주면 된다. 이 값의 비교를 통해 포멧에 맞게 출력해준다.

2 Step 2,3,4: DFGS

사용한 함수는 store, DGFS_create, DGFS_add, GFS_ls, DGFS_remove 로 구성되어있다,

-1 int store(FILE *dfp, int *addr, int *dedup, void*data, int storeing)

이 함수는 파일포인터를 입력받아 데이터를 저장하고 포인터로 받은 변수의 주소에 원하는 값을 할당하는 함수이다. 대략적인 코드는 다음과 같다. 데이터 비트맵을 불러온 뒤, 비트맵의 값이 1 인(데이터 block 이 존재하는) 부분에 대해서 해쉬 값을 조사한다. 해쉬 값이 같으면, 저장된 데이터와 저장할 데이터를 비교하고, 같으면 해쉬 테이블영역에 link 값을 +1 해주고, dedup 값을 +1 해주고, addr 값을 그 비트맵에 대응되는 데이터리전의 block 주소값으로 넣고, return(0)를 해준다. 만약 모든 데이터에 대해 해쉬값이 같지 않다면, 가장 앞쪽의 빈 데이터 공간에 데이터를 저장하고(4kb, 빈공간이 남을 경우 null 값으로 채워짐), 데이터 비트맵의 비트를 1 로 바꾸어 주고, 해쉬 영역에 link = 1 된 값과 해쉬를 계산하여 20 바이트의 크기로 저장한다. 만약 storeing 이 1 이라면 해쉬값을 읽지 않고 무조건 데이터 비트맵의 빈공간에 저장하는 것으로 한다.(포인터를 저장하는 방법)

-2 int DGFS_create(char* file)

이 함수는 DGFS 파일을 만들어 주는 것이다. 먼저, 같은 이름의 파일이 있다면, 있다고 출력한 뒤 return(1)을 해준다.

파일을 w 로 오픈한 뒤, 길이에 맞게 파일 이름을 저장하고 데이터영역 직전 바이트까지 0 으로 초기화 시켜준다.

-3 int DGFS_add(char* filename)

먼저 파일을 r+로 열어준다. inode 비트맵 비트가 1 인 inode 들을 조사하여, 같은 이름일 경우, 같은 이름이 있다고 출력한 뒤 return(1)을 해준다. 이후, inode 비트맵의 비트값을 1 로 바꾸어 주고, 비어있는 가장 앞쪽의 inode 에 값들을 저장한다. 값들은 파일명을 128 바이트씩, 파일 크기를 8 바이트씩, 그리고 파일 포인터를 저장한다. 파일 포인터 저장 방법은 파일 크기에 따라 달라진다. i 번째 데이터(크기는 4096 바이트)를 저장한다. 저장은 store 함수를 통해 저장되고, 저장되거나 linked 된 주소는 addr 변수에 저장시킨다. i 가 10 미만일때는 pointer 에 저장시키고, i 가 10 이상 1034 미만일 때에는 indirect pointer[i – 10]에, 1034 이상 1034+1024^2 미만일 때에는 double indirect point[(i-10-1024)/1024][(i-10-1024)%1024]에 저장시킨다. 그리고 indirect pointer 를 store 함수를 통해서 저장시키고(이 이하 내용은 dedup 을 증가시키지 않는다)주소값을 pointer[10]에 저장시켜주고, double indirect pointer[index]는 저장시켜 그 주소를 dobpointaddr[index]라는 만들어준 배열에 저장해주고, dobpointaddr[]도 저장시켜 그 주소를 pointer[11]에 저장시켜준다. 저장시킬지 말지는 파일의 크기에 따라 결정되고, 그 순서는 pointer(0~9), indirect pointer, double indirect pointer 가 우선순위가 된다.

-3 int DGFS_ls()

ls 는 DGFS 파일의 저장된 상태를 출력하는 함수이다. inode 비트맵을 불러온 뒤, 1 인 모든 inode 를 조사한다. 각각 이름, 크기를 read 하여 출력시킨다. 또한 크기는 전부 더해줘 totalsize 를 계산한다.data 비트맵을 조사하여 해당 비트가 1 일 경우,초기에 0 값이었던 blocks 변수에 blocks += 1 로 1 을 더해준다. 마지막으로 총 파일 크기와 블록 개수를 출력해준다.

-4 int DGFS_remove(char* filename)

이 함수는 해당 이름을 가진 파일을 지워주는 함수이다. 먼저, 모든 inode 를 조사하여 같은 이름을 가진 파일이 있는지 찾는다. 없으면 오류 출력 후 return(1)을 해주고, 있다면 그 파일을 지워준다.

파일을 지우는 과정은 다음과 같다. inode 비트맵 비트를 0 으로 바꾸고, link 값을 바꾸기 위해, 포인터 값들을 읽어온다. 포인터 값들은 가지고 있는 모든 데이터 주소를 읽어올 때까지 direct 연결로 불러온다. 주소값을 알면, 몇번째 블록인지 알고, 이를 통해 해쉬영역에 접근할 수 있다. 여기서 가지고 있는 모든 포인터에 해당하는 블록의 link 를 -1 씩 해준다(indirect pointer, double indirect pointer, directed indirect pointer 포함) 만약 link 값이 1 이하로 떨어 질 경우, link 의 값을 0 으로 저장하고, 데이터 비트맵의 비트를 0 으로 바꾸어준다.

-5 int DGFS_extract(char* filename, char* output_filename)

모든 inode 를 탐색해서, 같은 문자열일 경우 inode data 들을 읽어온다. 저장되어있는 주소 값을 불러와 파일사이즈와 같아질 때 까지 데이터를 불러오고 저장한다. 저장할 데이터의 크기는 temped 로 지정을 했고, 이 값은 size 를 4kb 씩 빼면서 진행하는 중 남은 값이 4kb 보다 낮을 경우 size, 아닐 경우 4kb 로 지정을 하였다.

확인해 본 결과, 40mb 크기 이하의 파일에 대해 불러오고, ls 값을 확인하고, remove 하고, extract 하는 과정에서 문제가 없음을 확인하였다. (extracted 된 파일의 경우, diff 명령을 통해 값이 값이 같은지 확인하였음)그 이상의 크기 파일에 대해서는 오래 걸려 확인해 보지는 못했지만, 충분히 가능할 것이다.

2 Step 5: improved-DFGS

압축은 파일 데이터 자체에 대한 부분에서는 하지 못하였다. 다만 같은 파일이 있는 경우와 같은 특수한 경우에 대해서 DGFS 파일의 크기를 줄이려고 하였다.

먼저 Pointer 값들을 데이터에 저장하게 되는데, 이 포인터의 데이터가 저장된 block 의 link 수는 1 이라는 점에서 착안하여 같은 파일이 있는 경우, link 수를 늘리는 방식으로 하면 포인터 데이터를 덜 저장시킬 수 있을 것이라 생각하여 한 방법이다. 방법은 간단하다. Store 에 들어가는 조건문에서 storing 조건을 빼는 것이다. 이렇게 되면, 포인터 값들을 저장하는 데에도 포인터 값들의 데이터가 이미 저장되어 있는 경우 해당 block 의 link 에 +1 을 하고 추가로 저장하지 않는 방식으로 하였다. 실행해본 결과, 기존의 코드는 같은 데이터의 이름만 다른 파일들을 입력 받을 경우, DGFS 파일의 크기가 늘어났지만 improved 된 알고리즘의 경우, DGFS 의 파일

크기도 늘어나지 않고, allocated 블록의 개수도 늘어나지 않았다.

또한 나머지 한 방법은 store 함수에서 4096 씩 저장하지 않고, 데이터의 크기를 받아 필요한 크기만큼만 저장시키는 것이다. 이렇게 할 경우, 대부분의 상황에서 약간의 메모리 이득을 볼 수 있다. 제일 끝부분의 메모리에서 4096 바이트 크기의 데이터가 아닌, 약간 더 적은 데이터(평균 2048 바이트크기의 데이터)가 저장되므로, 평균 2048 바이트 정도 크기의 데이터 이득을 볼 수 있다. 실행해본 결과, 제공된 dedup_test_1.bin 과 dedup_test_2.bin 의 데이터를 add 한 두 DGFS 파일의 크기를 비교하면 45844480bB : 45840420B(= 알고리즘없이 : 알고리즘 넣고)으로 같은 파일이 아니더라도 유의미한 데이터 크기 차이가 있는 것을 볼 수 있다. dedup_test_1.bin 와 같은 데이터를 가진 세개의 파일을 input 으로 주었을 때는 25344000B : 25315332B 으로 데이터 블록 부분의 크기만 보자면 45056B : 16388B 로 좋은 효율을 보여주는 것을 볼 수 있다.

REFERENCES

No reference