컴퓨터학부, 20190511, 배준형

1. 과제 개요

xv6 파일시스템은 기본적으로 direct 연결 12 개와 Indirect 형태로 포인터로 연결하여 (128+12)*512(=PageSize) 140*512 = 약 70KB 를 지원하는 파일시스템으로 구성되어 있다. 즉 파일 크기가 70KB 를 넘어갈 수 없기 때문에 동영상이나 고용량 프로그램 등에서 불리한 면을 보여주고 있다. 이를 보완하기 위해 multi-level 파일시스템을 구현한 파일시스템 구조를 통해 파일 크기를 확장해볼 예정이다.

새로 고안된 multi-level 파일 시스템은 기존에 Indirect 가 1 개였던 파일시스템을 벗어나서 Direct 6 개 Indirect 4 개 2-Level Indirect 2 개 3-Level Indirect 1 개를 배치하여 파일 크기를 (6+128*4 +128*128*2 + 128*128*128) * 512 인 약 1GB 크기의 파일까지도 지원하는 파일시스템을 구현하는 것이다. 이를 구현하기 위해 xv6 의 파일 시스템인 fs.c 와 fs.h 를 중심으로 수정하여 구현할 예정이다.

그리고 해당 파일시스템에서 ssualloc 이라는 시스템 콜을 이용하여 초기에 프로세스 가상메모리 페이지테이블의 인덱스에 할당 표시만 해두고 나중에 접근할 때 실제로 물리메모리로 접근되는 Lazy_Allocation 시스템을 구현하는 시스템콜을 지원할 예정이고 getvp 와 getpp 를 이용하여 현 프로세스에서 사용되는 가상메모리와 물리메모리 크기를 보여주는 시스템 콜을 지원할 예정이다. 이를 위하여 sysproc.c 와 trap.c 를 적절히 수정하여 Lazy Allocation 을 할당가능한 OS 를 제작할 예정이다.

2. 상세설계

[1. 가상메모리 할당을 위한 ssualloc() 시스템 콜 구현]

- 해당 시스템콜의 주요 원리는 다음과 같다.

> ssualloc 를 요청하면 가상메모리를 할당하되 물리메모리는 현 시점에서 할당하지 않음.
(vm.c 의 allocuvm 함수에서 물리메모리할당을 빼서 구현)

> 실제로 해당 메모리에 접근하였을 때 접근한 페이지에서만 물리메모리 할당 (trap.c 에서 page Fault 처리)

실제로 구현 전 사전정보로 알고있는 xv6 의 가상메모리 공간의 형태는 mmu.h 를 참고하여 알 수 있었다.
(10: 페이지디렉토리, 10:페이지테이블, 12:페이지크기) 형태로 비트가 분배되어있었음을 알 수 있었다.
즉 PDE 매크로는 가상메모리의 상위 10 비트를 구하는 매크로이고, PTX 는 페이지 테이블을 (이후 10 비트) 를 구하는 매크로로 사용되고 있었다.

(가상메모리에만 메모리 할당방법 구현)

ssualloc 의 기본조건은 **양수이면서 페이지크기의 배수(512 * N) 이기 때문에** 해당조건을 만족하지 못한다면 -1 을 리턴하여 에러처리하도록 설계하였다.

그리고 ssualloc 에서 가상메모리의 페이지를 할당해주기 위해서 vm.c 의 allocuvm 에서 사용하는 기법을 응용하여 현 프로세스의 가상메모리(사용자영역) 크기를 알려주는 myproc()->sz 를 이용하여 해당 위치에 새로운 가상메모리 페이지를 할당하겠다는 의미로 PGROUNDUP(myproc()->sz) 를 통해 가상페이지를 할당하였다. 그리고 해당 위치의 페이지 테이블을 접근하기 위해 마치 Intel 의 **cr3 레지스터 역할을하는 myproc()->pgdir** 변수를 이용하여 페이지디렉토리 주소를 얻었다. 이후 PDX, PTX 를 이용하여 새롭게할당한 페이지테이블에다 시스템콜에서 직접 페이지를 할당하는 방식으로 구현하였다. **이 과정에서 trap.c 에서 ssualloc() 시스템 콜을 이용하여 페이지테이블에 할당은 되었지만 PageFault 가 발생하였음을 알려주기 위하여 mmu.h 내부에다 (#define PTE_LAZY 0x008) 매크로를 추가하여 페이지테이블 엔트리에 해당 내용을 기록하였다.**

(실제 메모리 접근 시 trap.c 에서 페이지 Fault 처리)

실제로 xv6 의 trap.c 의 내부구현에서는 페이지폴트가 발생할 시 trapno 로 처리하고 있지 않으며 단지 Trap 인덱스 (매크로) 넘버만 띄워주고 종료하도록 되어있었다. 해당 과정에서 **중요한 힌트로 rcr2() 라는 함수** 를 호출하고 있었는데 해당함수에서는 페이지폴트가 발생한 **가상주소를 리턴하고있었다** . 이 점을 활용하여 Lazy Allocation 을 구현할 수 있었다. 우선 **PageFault TRAP** 가 발생한다면 (T_PGFLT) rcr2 로 가상주소를 받아온 후 해당 가상주소의 PDE 와 PTE 를 구하기 위해서 가상주소의 하위 12 비트(페이지크기비트) 를 날리는 매크로 **PGROUNDDOWN** 을 통해서 PDE 와 PTE 를 PDX, PTX 매크로를 이용하여 알아냈고 myproc()->pgdir 를 이용하여 실제 페이지 테이블에 접근해본 뒤 앞선 **검사방식인 PTE_LAZY** 플래그가 설정되어 있는지 확인해보고 PTE_LAZY 가 설정되어있다면 그때서야 **kalloc() 를 이용하여 물리메모리를 할당받아 페이지테이블에 연결하기위해 mappages() 를 이용하였다. (해당과정에서 PTE_LAZY 대신 PTE_P | PTE_R | PTE_W 비트를 설정하였다.)**

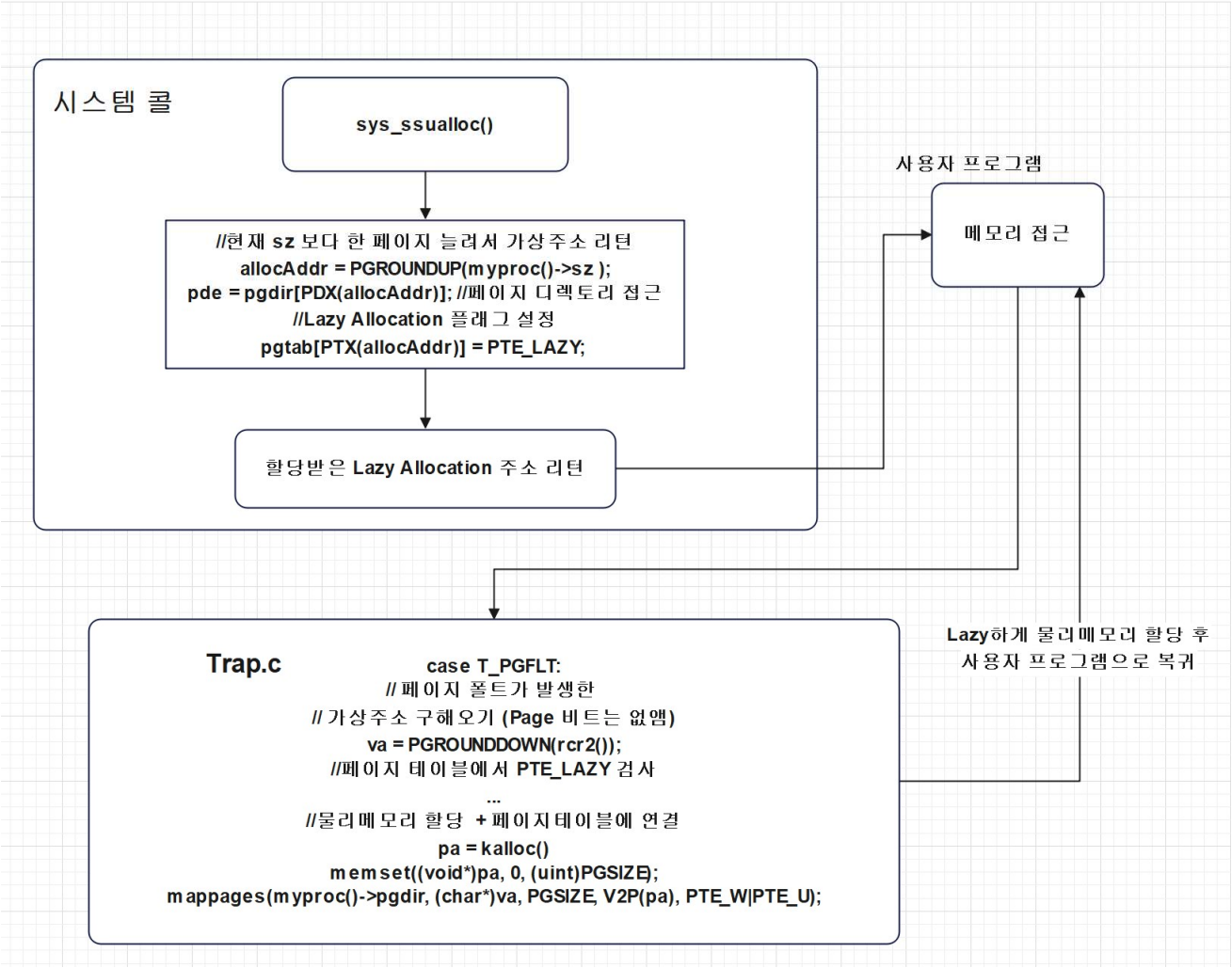
- **중요한 점 : vm.c 의 mappages() 를 사용하기 위해 vm.c 의 static int mappage 구현부를 int mappage 로 변경하였습니다.** (+defs.h 에 static int mappages(pde_t* pgdir, void* va, uint size, uint pa, int perm); 추가)

이를 통해 ssualloc 는 가상메모리에만 임시적으로 할당한 후 접근 시 물리메모리를 할당하는 시스템콜을 구현할 수 있었다.

[2. 가상메모리 할당 크기를 알 수 있는 getvp, getpp 구현]

getvp 와 getpp 시스템콜을 구현하기 위해 사전지식을 이용하여 xv6 에서 사용하는 ELF 포멧에서는 **0x8000000 부터는 kernel Stack 임을** 인지하여 0x00000000 ~ 0x7FFFFFFF 페이지 테이블을 검사하여 PTE_P (페이지테이블이 유효한가) 를 체크하여 유효한 페이지 수를 세서 할당된 페이지 테이블 수를 체크하였다.

getvp 에서는 위의 과정과 유사하지만 PageTable 에서 PTE_P 뿐 아니라 PTE_LAZY 가 설정된 페이지의 개수를 세어 실제로 물리메모리에 할당된 페이지 수를 체크하였다.



[1 번 Lazy Allocation 호출과정과 실제 물리메모리 할당과정]

[3. xv6 의 파일 시스템을 계층적 (Cascade) Multi-Level 파일시스템을 지원하여 파일 시스템 공간 크기 확장]

xv6 의 원래 파일시스템 변경의 핵심은 fs.c 의 bmap 함수와 itrunc 함수의 변형이 핵심이다.

(원래 xv6 의 파일시스템)

read 나 write 내부에서 동일한 방식으로 bmap 을 호출한다. bmap 함수의 가장큰 역할은 DIRECT 12 개까지는 해당 프로세스의 inode 내부의 address 에 직접 할당 (balloc())한다. (만약 존재한다면 해당 직접 주소를 바로 리턴). 이후 bmap 인자로 넘겨받은 블록번호가 12 개를 넘어간다면 INDIRECT 모드로 바뀌게되며 이때는 bmap 내부에서 디렉토리 페이지 (블록 번호)를 inode 에 할당한 후 4 바이트씩 총 128 개씩 페이지에 차례대로 페이지 블록번호(정수크기)를 할당하도록 설정하였다.

이후 itrunc 함수 역시 동일한 방법으로 DIRECT 12 개를 free 한 후 INDIREC 의 페이지디렉토리를 free 한 후 할당된 나머지 페이지들이 존재하면 할당해제 하도록 설정되어있다.

(새롭게 구현한 xv6 멀티레벨 파일시스템)

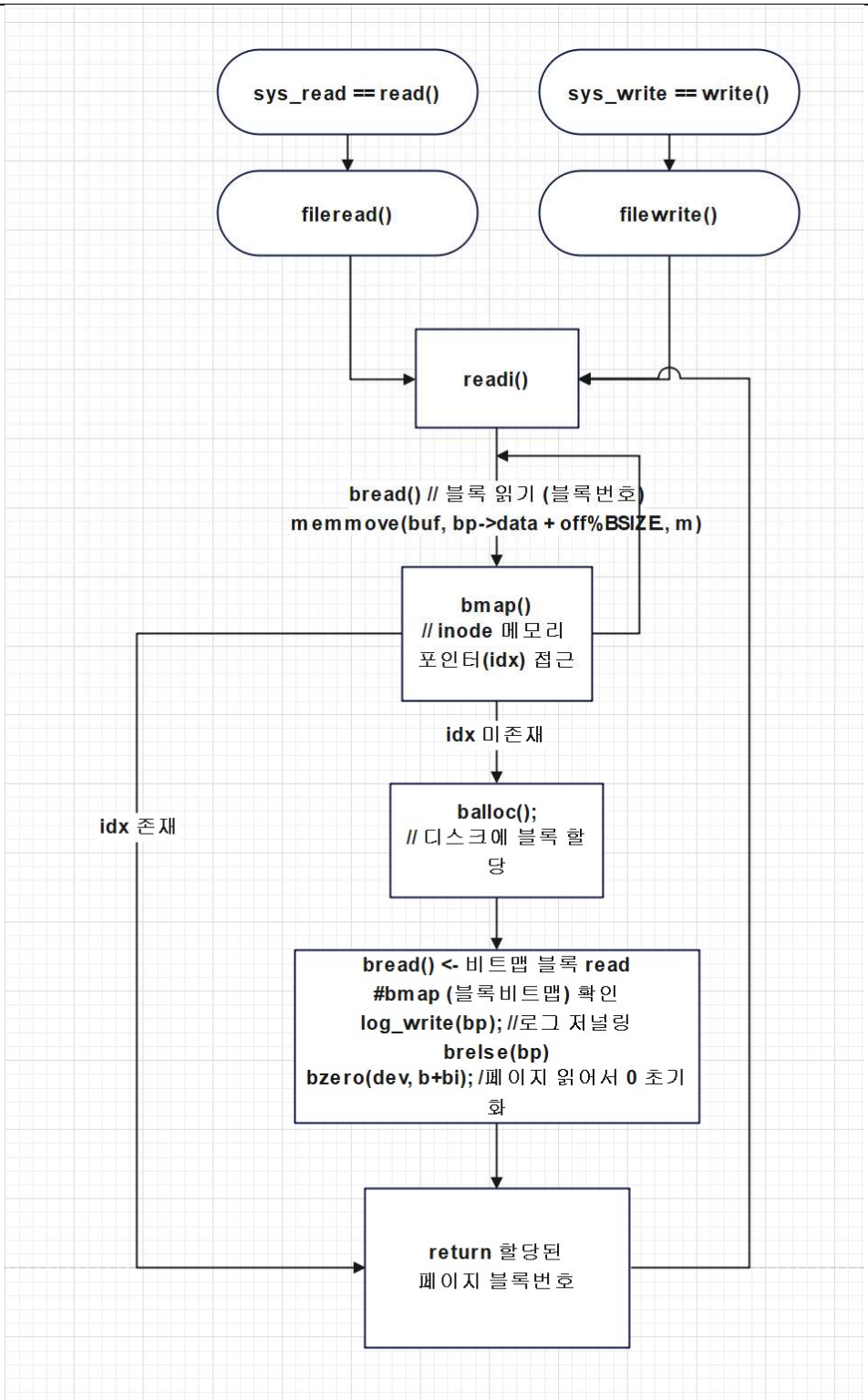
fs.h 의 파라미터에서 레벨별 파일크기를 새롭게 정의하였다.

```
#define NINDIRECT (BSIZE / sizeof(uint)) //주소개수를 넣을 수 있는 개수
#define LEVEL1      NINDIRECT*4                  // 6,7,8,9
#define LEVEL2      NINDIRECT*NINDIRECT*2        // 10,11
#define LEVEL3      NINDIRECT*NINDIRECT*NINDIRECT   // 12
#define MAXFILE (NDIRECT + LEVEL1 + LEVEL2 + LEVEL3)
```

Incore Inode 의 파일시스템 역시 uint addrs[NDIRECT+7]; 형태로 변경하였다.

앞에서 설명한 fs.c 의 bmap 을 직접매핑 6 개와 4 개의 INDIRECT 구조는 앞선 xv6 의 원래형태와 비슷하게 구현해두었고, 3-LEVEL 4-LEVEL 파일시스템은 bread, bwrite 함수 내부에서 사용된 테크닉을 활용하여 for 문을 이용하여 **for (idx_lvl2 = 6+4 ; bn >= NINDIRECT*NINDIRECT ; idx_lvl2++, bn-=NINDIRECT*NINDIRECT); 와 같이 배치하여 2-level 매핑의 가장 앞단 디렉토리의 인덱스를 찾아 할당되지 않았다면 balloc()으로 디렉토리를 받아온다.** 또한 해당 첫 번째 디렉토리 블록을 따라가 다시 한 번 INDIRECT 를 구현하기 위해 다시 for 문을 이용하여 **for (idx_lvl1=0 ; bn >= NINDIRECT ; idx_lvl1++, bn -=NINDIRECT);** 를 이용하여 2 번째 디렉토리 포인터를 찾는다. (없으면 balloc 으로 할당) 이후 마지막으로 2 번째 디렉토리 페이지에서 **실제로 할당된 페이지 블록을 찾아 해당 블록을 리턴하도록 3-Level Mapping 시스템을 구현하였다.** 이후 마지막 4 단계 포인터 매핑 구조를 구한하기 위해 bn 을 LEVEL2 크기까지만큼 감소시킨 후 LEVEL3 를 구현하였다. LEVEL-3 은 1 개의 인덱스 밖에 없으므로 LEVEL-3 의 인덱스를 찾을 필요는 없다. (addr[12] 이기 때문) 이후 해당 인덱스에 블록이 없다면 4 단계 매핑의 첫 번째 디렉토리를 매핑한 후 앞선 3 단계 매핑 파일시스템을 다시 한 번 구현하도록 설계하였다.

이후 파일 할당해제를 위한 itrunc 함수를 inode 의 addr 인덱스에 맞게 수정해주었다.
이 역시 bmap 과 비슷한 구조로 1 단계(DIRCET) 와 2 단계(INDIRECT) 파일시스템은 앞서 원래의 xv6 파일시스템과 비슷하게 할당해제를 해주고 있다. 마지막으로 3 단계 매핑시스템에 대응하는 10,11 번 inode idx 의 경우에는 첫 번째 디렉토리가 존재한다면 할당해제(bfree)한 후 두 번째 디렉토리에서 최대 128 개까지의 할당된 블록번호를 확인 후 존재한다면 다시 해당 블록으로 이동 후 128 개의 디렉토리를 확인 후 페이지 할당해제 (bfree)를 해준다. 그리고 마지막으로 4 단계 매핑 시스템에 대응하는 12 번 inode idx 의 경우에는 첫 번째 디렉토리, 2 번째디렉토리, 3 번째 디렉토리를 통해 Indirecting 과정의 디렉토리들에 접근하여 할당된 페이지들을 할당해제 (bfree) 해준 뒤 계층적으로 상위계층적으로 4 단계 말단 -> 3 단계 디렉토리 -> 2 단계디렉토리 -> 가장 부모 매핑 디렉토리 페이지를 할당해제 (bfree) 해준다.
그리고 bfree 를 해주기 전에 buffer cache 에서 페이지를 빼주기위해 bfree 를 해준다.



[xv6 가 file read/write 를 할 때 진행되는 함수 call Graph]

[xv6 부팅 시작점에 실행되는 mkfs.c]

추가적으로 xv6 에선 기존 프로그램에서는 xv6 를 부팅하기 전 mkfs.c 파일을 실행시켜 에뮬레이터 시키는 모습을 볼 수 있다.

make qemu 중 아래 내용 .. (./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _ssufs_test _ssualloc_test)

즉, 파일 시스템 초기에 mkfs.c 에서 fs.h 를 바탕으로 미리 xv6 에 파일들을 에뮬레이터를 해주고있음을 볼 수 있다. 이를 지원해주기 위해 기존 mkfs.c (DIRECT 12 개 INDIRECT 1 개) 에서 발전시켜 멀티레벨 파일시스템을

지원하도록 수정하였다. mkfs.c 의 주요 수정점은 **iappnd 함수 (void iappend(uint inum, void* xp, int n)) 와 balloc 함수 (void balloc (int used)) 이다.**

balloc 함수는 used 블록 수 만큼 **데이터블록 비트맵을 1 로 변경시켜주는 함수이다. 해당 함수는 원래 초기 부팅 시 최대 1 블록만 사용된다고 가정되고 짜여져있기에 가변적으로 for 문을 이용해서 used 가 사용되는 블록수 만큼 for 문을 반복하여 bitmap 을 초기화시켜주도록 설계하였다.**

iappend 함수는 inode 번호에 해당하는 블록개수를 xint() 함수로 LITTLE ENDIAN 형태로 변환해서 **파일 크기만큼 xv6 의 inode 에 따라가는 데이터블록을 미리 할당해주는 함수이다.** 함수내에서 사용된 기초 함수로는 아래와 같다.

<mkfs.c 내부 주요 함수들>

void balloc (int used); //used 만큼 사용된 블록 수의 데이터블록 비트맵을 1 로 초기화해줌

void iappend (uint inum, void* xp, int n); //inumber 에 해당하는 데이터블록을 n 개만큼 데이터블록을 에뮬레이터에 올려주는 함수.

void rinode(int inum, struct dinode *ip); // inum 에 해당하는 ondisk inode 를 읽어온다.

uint xint (uint x); //x 주소를 LITTLE ENDIAN 으로 변경한 후 해당 주소 리턴.

void rsect (uint sec, void* buf); // 블록번호 (sec) 로부터 buf 로 read

void wsect (uint sec, void* buf); // 블록번호 (sec) 에 buf 내용 쓰기

void bcopy (void *src, void* dest, size_t n); //src -> dest 로 n 만큼 복사.

해당 위의 함수들을 이용해서 iappend 가 호출될 때마다 addrs 의 index 와 각 디렉토리 페이지의 블록번호를 수식적으로 계산하여 블록을 초기화해서 집어넣었다.

실제 수식은 4 단계 멀티페이지 디렉토리를 사용하는 inode 12 번을 참조하면 아래와 같이 작성했다.

<idx1 : addrs 번호, tmp : 1 번째 멀티페이지에서의 번호, tmp2 ; 2 번째 멀티페이지에서의 번호, ...>

```
idx1 = 12;
tmp = ((fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) / (NINDIRECT*NINDIRECT));
tmp2 = ((fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) / (NINDIRECT)) % NINDIRECT;
tmp3 = (fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) % NINDIRECT; //3 단계 디렉토리 주소
```

실제로 테스트 해보기 위해 (과제제출시에는 포함이안되어있음) Makefile 의 fs.img: 부분에 대용량 파일을 넣어서 테스트 해보았다.

test1 > 대략 1.7 GB 크기의 동영상을 test.txt 로 변환하여 fs.img 에 추가

test2 > 대략 766MB 크기의 동영상을 test.txt 로 변환하여 fs.img 에 추가

결과사진의 idx1 은 inode 데이터블록 포인터 블록의 번호이다.

tmp 는 1 차 간접 디렉토리 포인터 블록 번호

tmp2 는 2 차 간접 디렉토리 포인터 블롭 번호

tmp3 는 마지막 페이지의 위치번호를 의미한다.

즉 ip->addrs[idx1] 중 tmp 번째 블록 포인터 -> 중 tmp2 번째 블록포인터 -> 중 tmp3 번째 블록포인터를 따라갈 것이다.

mkfs.c 수정 한 부분 중 iappend 내용 일부 .. (데이터블록을 옮길 때 마다 몇번째 addr idx, 몇번째 디렉토리 번호인지 출력하도록 디버깅할 수 있게 되어있음 (실제 제출시에는 해당 디버깅값은 빠져서 제출됩니다.)

```
else { //4 단계 디렉토리
    idx1 = 12; //어차피 마지막
```

tmp = ((fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) / (NINDIRECT*NINDIRECT)); // 1 단계 디렉토리 idx -> 얘 1 개뿐이라 / 안해줘도됨

tmp2 = ((fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) / (NINDIRECT)) % NINDIRECT; // 2 단계 디렉토리 idx

tmp3 = (fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) % NINDIRECT; //3 단계 디렉토리 주소

#if JH

printf("3 LEVEL INDIRECT : %d₩n", fbn);

printf("3 LEVEL INDIRECT : idx1 = %d, tmp = %d, tmp2 = %d, tmp3 = %d₩n", idx1, tmp, tmp2, tmp3);

#endif

…

<test1 > 1.7 GB 동영상을 qemu 에뮬레이터로 test.txt 로 변환하여 넣어주었을 때>

```
3 LEVEL INDIRECT : idx1 = 12, tmp = 127, tmp2 = 127, tmp3 = 124
3 LEVEL INDIRECT : 2130435
3 LEVEL INDIRECT : idx1 = 12, tmp = 127, tmp2 = 127, tmp3 = 125
3 LEVEL INDIRECT : 2130436
3 LEVEL INDIRECT : idx1 = 12, tmp = 127, tmp2 = 127, tmp3 = 126
3 LEVEL INDIRECT : 2130437
3 LEVEL INDIRECT : idx1 = 12, tmp = 127, tmp2 = 127, tmp3 = 127
mkfs: mkfs.c:284: iappend: Assertion `fbn < MAXFILE' failed.
make: *** [Makefile:188: fs.img] Aborted
make: *** Deleting file 'fs.img'
```

(마지막 블록까지 채워지고 한계를 넘어가서 에러처리를 하도록 되어있음)

- **변경된 Multi file system 은 대략 1GB 의 파일시스템을 지원하기 때문.**

<test2 > 760 MB 동영상을 test.txt 로 넣어주었을 때 : mkfs.c 의 balloc 최댓값을 설정해주었다.>

➔ 정상적으로 돌아감을 알 수 있다.

```
Block Count : 1569691
3 LEVEL INDIRECT : 1569691
3 LEVEL INDIRECT : idx1 = 12, tmp = 93, tmp2 = 99, tmp3 = 21
new file : _cat
```

[추가적인 구현]

부팅 전 xv6 에 필요한 user 파일들을 미리 할당해두고 올려주는 파일시스템에 대응하는 mkfs.c 역시 inode 의 계층적 구조에 맞게 수정하였다.

#P1, P2, P3 과 동일하게 시스템콜을 추가하였다.

usys.S 어셈블리어 파일을 통해 user 모드에서 시스템 콜에 연결해주기 위해 SYSCALL() 시스템 콜 어셈블리어 연결을 해주고 있다.

syscall.h 의 시스템콜 매크로(index) 를 등록해주고

시스템 콜 syscall.c 에서 syscall 함수배열에 새로운 시스템 콜 함 를 추가해준다. 그리고 extern 을 선언해서 해당 시스템콜이 sysfile.c 내지 sysproc.c 파일에 존재함을 알려준다.

실제구현은 sysproc.c 에서 구현한다.

그리고 구현이 마친다면 user.h 에 해당 시스템콜을 user 프로그래밍에서 호출할 수 있도록 함수를 미리 선언해준다.

[xv6 에서 파일 역할과 구현]

stat.h	open 할 때 flag 에 대한 내용을 담음.
	#define T_DIR 1 // Directory
	#define T_FILE 2 // File
	#define T_DEV 3 // Device
	
	struct stat {
	short type; // Type of file
	int dev; // File system's disk device
	uint ino; // Inode number
	short nlink; // Number of links to file
	uint size; // Size of file in bytes
	};

defs.h	각 파일 함수들 어디있는지 찾을 때 유용함. (알고보니 defs.h 를 해주면 만능으로 함수를 사용할 수 있음 Makefile 에서 obj 연결을 해줌)
mkfs.c	파일 시스템을 만들어주는 파일 • 디스크에 i-node 를 할당하고, i-map 등을 할당하는 함수들이 포진함. (파일 시작전 실행)
ide.c	디스크 드라이버 • OS 에게 인터럽트를 발생시키고 + 디스크에 read, write 를 받아오는 역할.

file.h	struct file {
	enum { FD_NONE, FD_PIPE, FD_INODE } type;
	int ref; // reference count
	char readable;
	char writable;
	struct pipe *pipe;
	struct inode *ip;
	uint off;
	};
	
	// in-memory copy of an inode
	struct inode {
	uint dev; // Device number

	uint inum; // Inode number
	int ref; // Reference count
	struct sleeplock lock; // protects everything below here
	int valid; // inode has been read from disk?
	
	short type; // copy of disk inode
	short major;
	short minor;
	short nlink;
	uint size; // 현재 파일의 크기를 저장함.
	uint addrs[NDIRECT+1];
	};
	
	// table mapping major device number to
	// device functions
	struct devsw {
	int (*read)(struct inode*, char*, int);
	int (*write)(struct inode*, char*, int);
	};
	
	extern struct devsw devsw[];
	
	#define CONSOLE 1

file.c	file 데이터 할당 및 복사에 관한 파일.
fs.h	파일 inode 의 헤더파일 NDIRECT 개수 및 INDIRECT 개수 등 지정
fs.c	파일 시스템 파일 : i-map, d-map, i-node table 등... 접근해서 --> 메인메모리로 데이터를 받아옴. • i-node 를 할당하고 • 기본적으로 log 저널링을 사용하고 있음.
param.h	xv6 의 초기 부팅 시 데이터블록 개수, cpu 개수 등을 지정해놓는 파일

fs.c

기본 선언	struct devsw devsw[NDEV]; struct {

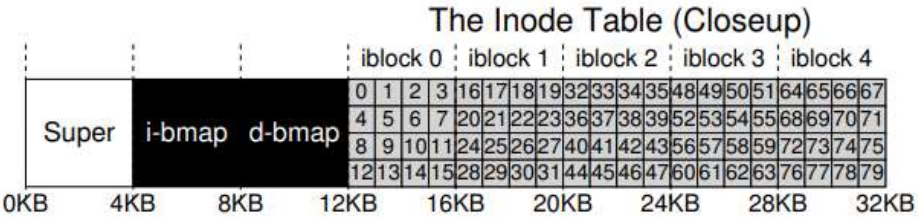
	struct spinlock lock; struct file file[NFILE]; } ftable; • open 된 파일 구조체 관리 • 모든 함수는 acquire --> release 방식으로 ftable 에 락을 걸 수 있음.
struct file* filealloc(void);	현재 존재하는 open 된 file 구조체 중 아직 할당되어있지 않은 file 구조체 반환 (최대 100 개 open 가능)
void fileclose(struct file*);	close() 호출할 때 호출됨 • ref 가 0 이면 file->type 에 FD_NONE 설정하고 return
struct file* filedup(struct file*);	파일에 link 를 걸 경우 file->ref 를 하나 증가.
void fileinit(void);	ftable 에 락 초기화 (세팅)
int fileread(struct file*, char*, int n);	file 에서 read 를 할 때 호출되는 함수. • readi() 를 호출하여 f->off 에 읽은 바이트만큼 반영 • f->off += readi() 을 하여 offset 반영 (readi 리턴값은 offset 에서 n 만큼 읽고, n 리턴)
int filestat(struct file* f, struct stat* st);	파일의 미타데이터 stat 을 구해주는 함수. 해당 파일 f 의 정보를 st 구조체로 담아줌.
int filewrite(struct file*, char*, int n);	file 에서 write 를 할 때 호출되는 함수 • 신기한 점은 log 트렌젝션의 최대 크기 (3 블록) 을 벗어나지 않도록 log 를 분할하여 write 함.

bio.c

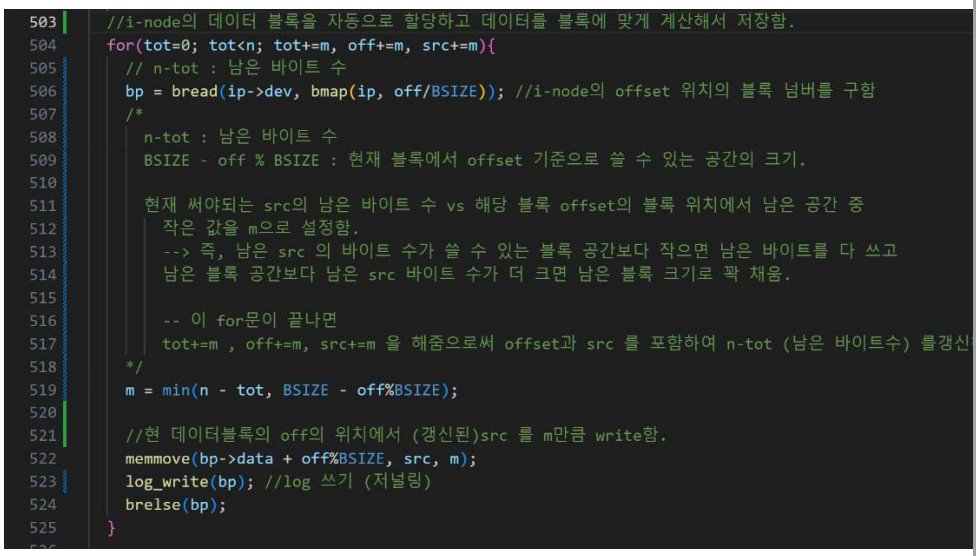
struct buf* bread (uint dev, uint blockno)	장치에서, block 번호에 해당하는 버퍼가 있으면 해당 버퍼를 받아서 return
`void bwrite (struct buf *b)`	버퍼 b 를 디스크로 내용을 내려보냄 ide_rw 호출
static struct buf* bget (uint dev, uint blockno)	block 캐시에서 dev 의 block number 에 데이터가 있으면 해당 버퍼를 리턴하고

	buf 에 없으면 최근에 참조되지 않았으면서 수정되지 않은 버퍼로 갈아끼우고 return

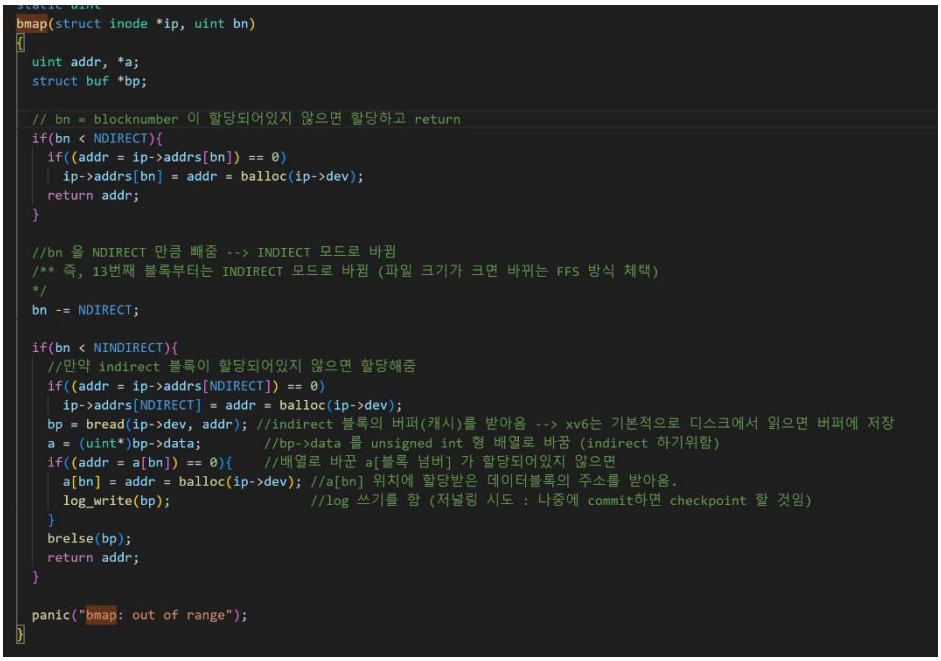
fs.c

void readsb (int dev, struct superblock *sb);	슈퍼 블록을 읽는 함수 (물론 xv6 는 1 블록당 512 Byte 로 되어있음) • 슈퍼블록은 디스크의 1 번 블록에 저장됨. • superblock *sb 라는 구조체에 해당 슈퍼블록데이터를 저장함. • **슈퍼블록**에는 b-map 위치, i-node table 위치, 데이터블록 시작 위치 등 여러 메타데이터 정보가 있음! 
int dirlink(struct inode*, char*, uint);	디렉토리 전용함수 디렉토리에 해당 name 을 연결하는 함수 return 성공하면 0, 에러시 -1
struct inode* dirlookup (struct inode*, char*, uint*);	디렉토리 전용 함수 해당 디렉토리 i-node 를 바탕으로 해당 디렉토리의 데이터블록을 접근하여 해당 디렉토리의 요소 (dirent) 안에 찾는 name 있는지 검사해서 있다면 해당 name 의 i-node 를 리턴.
i-node 계열 함수들	struct { struct spinlock lock; struct inode inode[NINODE]; } icache;
struct inode* ialloc (uint dev, short type);	`i-node` 를 새로 할당함 (type 으로 <- O_RDONLY 이런거) inum 을 1 부터 차례대로 증가시키며 i-node number 중 아직 할당되지 않은 i-node 를 찾아서 해당 i-node 를 i-node table 위치에 할당함.
struct inode* idup (struct inode*);	하드링크를 생성할 때 i-node 의 ref 값을 증가시켜주는 함수

void iinit (int dev);	inode 내부의 lock 을 초기화하고 슈퍼블록을 읽어서 (readsb) 슈퍼블록의 정보들을 출력함. 커널 부팅하면 sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
void ilock(struct inode*);	i-node 를 Lock 해줌 + **필요하면 i-node 의 정보를 디스크(버퍼캐시) 로부터 읽어옴**
void iput (struct inode*);	i-node 의 ref 를 하나씩 줄이는 함수, 만약에 i-node->ref 을 줄이면 0 이되면 ip->valid=0 을 만들어 i-node 를 유효 X 상태로 만듦 삭제할 때 사용할 듯.
void iunlock (struct inode*);	i-node 락을 해제함
void iunlockput (struct inode*);	i-node 락을 풀고 iput 호출. (아마 삭제할때 iput 이전에 호출하는듯)
void iupdate (struct inode* ip);	inode 의 번호 inode->inum 을 읽어서 슈퍼블록에 있는 i-node 의 정보를 받아서 i-node 를 update 해줌.
int namecmp (const char*, const char*);	디렉토리 전용함수 directory 검사할 때 이름 같은지 검사하는 함수. strncmp (s, t, DIRSIZ); 호출함.
struct inode* namei(char*)	디렉토리 전용함수 디렉토리 의 namex 를 호출하여 디렉토리 i-node 리턴
struct inode* nameiparent(char* path, char* name);	디렉토리 전용함수 디렉토리의 namex 를 호출하여 (iparentnumber 를 1 로 리턴) 하여 path 의 부모 i-node 를 리턴 를 리턴.
`int readi (struct inode *ip, char`	i-node 의 파일의 off 위치에서 n 만큼 det 로 데이터 읽기. • writei 와 거의 동일하게 동작함. (단지 mmove 의 1,2 번 인자만 바뀌었을 뿐)

*dst, uint off, uint n)	return -> 4 번째 인자 n 리턴 (n 의 값이 수정될 수 있음)
void stati(struct inode*, struct stat*);	inode 로부터 stat 구조체를 갱신하는 함수 갱신하는 내용은 5 개 • st->dev (장치 번호) • st->ino (inode 번호) • st->type (i-node 타입) • st->nlink (연결된링크개수) • st->size (파일크기)
int writei (struct inode *ip, char *src, uint off, uint n)	off 위치에서 n 바이트 만큼 src 데이터를 ip->inum 데이터 블록에 써야되는 함수. return -> 4 번째 인자 n 을 리턴하거나 에러시 -1 해당 함수에서 for 문 블록을 잘 볼 필요가 있다. i-node 의 데이터 블록을 자동으로 할당할지 사용할지 판단함 (bmap) 그리고 for 문을 돌면서 데이터블록 크기만큼 src 를 자르고 계속 데이터를 집어넣음. 
<아래부터는 defs.h 에 정의되지 않은 함수들>	
static uint bmap(struct inode *ip, uint bn)	i-node ip 의 bn (블록 데이터 number) 를 새로 할당해주는 함수. i-node 의 addr 주소에서 직접포인터, 간접포인터를 파악해서 블록의 주소를 return 해주는 함수. 12 번째 블록까지는 직접포인터로 계산하고, 13 번째블록부터는 INDIRECT 방식으로 mapping 해주고 있다. <원본 xv6 코드 해석>

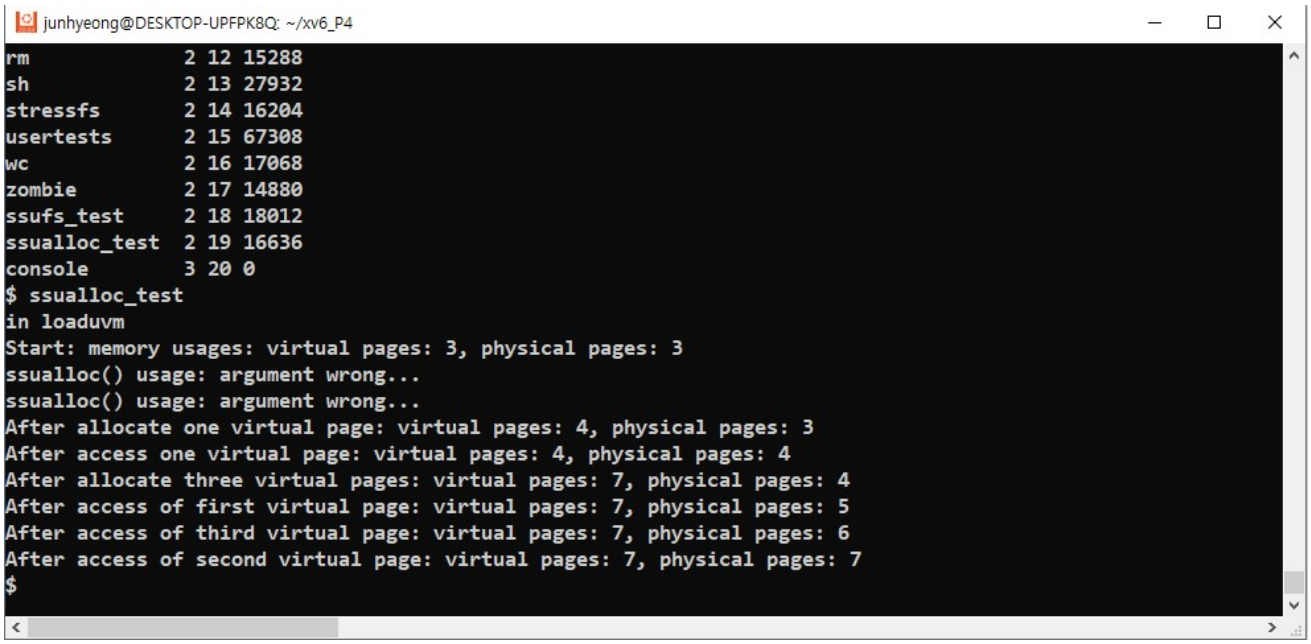
```
503  //i-node의 데이터 블록을 자동으로 할당하고 데이터를 블록에 맞게 계산해서 저장함.
504  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
505    // n-tot : 남은 바이트 수
506    bp = bread(ip->dev, bmap(ip, off/BSIZE)); //i-node의 offset 위치의 블록 넘버를 구함
507    /*
508      n-tot : 남은 바이트 수
509      BSIZE - off % BSIZE : 현재 블록에서 offset 기준으로 쓸 수 있는 공간의 크기.
510
511      현재 써야되는 src의 남은 바이트 수 vs 해당 블록 offset의 블록 위치에서 남은 공간 중
512        작은 값을 m으로 설정함.
513        --> 즉, 남은 src 의 바이트 수가 쓸 수 있는 블록 공간보다 작으면 남은 바이트를 다 쓰고
514        남은 블록 공간보다 남은 src 바이트 수가 더 크면 남은 블록 크기로 꽉 채움.
515
516        -- 이 for문이 끝나면
517        tot+=m , off+=m, src+=m 을 해줌으로써 offset과 src 를 포함하여 n-tot (남은 바이트수) 를갱신
518    */
519    m = min(n - tot, BSIZE - off%BSIZE);
520
521    //현 데이터블록의 off의 위치에서 (갱신된)src 를 m만큼 write함.
522    memmove(bp->data + off%BSIZE, src, m);
523    log_write(bp); //log 쓰기 (저널링)
524    brelse(bp);
525  }
```

	
static uint balloc (uint dev)	data-bitmap 을 reading 해서 비어있는 비트맵 (== 할당되지 않은 데이터블록) 을 찾아서 해당 블록을 모두 0 으로 초기화 한 후 (bzero()호출) 해당 블록 번호 리턴. (한 비트맵 블록 당 8*512 Bits 를 가짐 : 블록크기(Byte) -> Bits) 해당 함수에서 bi/8 를 하는걸 볼 수 있는데 이는 바이트->비트 때문에 한 것. ex) bi = 9 면 사실상 9 비트로 봐야하는데 C 언어는 비트단위가 없어서 bi/8 == 1 이 됨. 그래서 xxxx xx0x 라고 했을 때 0 의 위치에 1 를 설정하기 위해 m = 1 << (bi%8) 를 한 것.
static void bzero (int dev, int bno)	dev 장치(번호) 의 block number 위치에 값을 모두 0 로 만들어버리는 함수. • 즉, 해당 디스크 블록 넘버를 데이터블록으로 사용한다고 치고, 해당 블록을 초기화하는 함수. • 이전에 balloc() 에서 data bitmap 에 체킹하고 --> bzero() 호출하는식.
static void bfree (int dev, uint b)	block 을 할당 해제하는 함수 data-bitmap 에도 bp->data[bi/8] &= ~m 을 하여 할당해제함. • m 위치에 비트를 없에겠다. (m 위치의 비트를 뒤집어서 and 하니까 ㅋㅋ)

	static char* skipelem (char *path, char *name)	디렉토리 전용함수 /를 기준으로 이름을 분리해주는 함수. • //a -> a • ///a -> a • /a/b/ -> a • "" -> 0 **(중요) :** **return 값으로는 구분자를 처리하고 남은 문자열** **name 에는 구분자로 처리한 토큰을 넣는다.** ex) ///a/b/c 라고 하면 --> return 값은 b/c/, name 에는 a 가 들어있다.
	static struct inode* namex(char *path, int nameiparent, char *name)	디렉토리 전용함수 path 가 / 이면 (inode 1 번은 루트 디렉토리) 루트 디렉토리 i-node 를 받아옴. 이외에는 현재 경로의 i-node 를 받아옴.

3. 결과

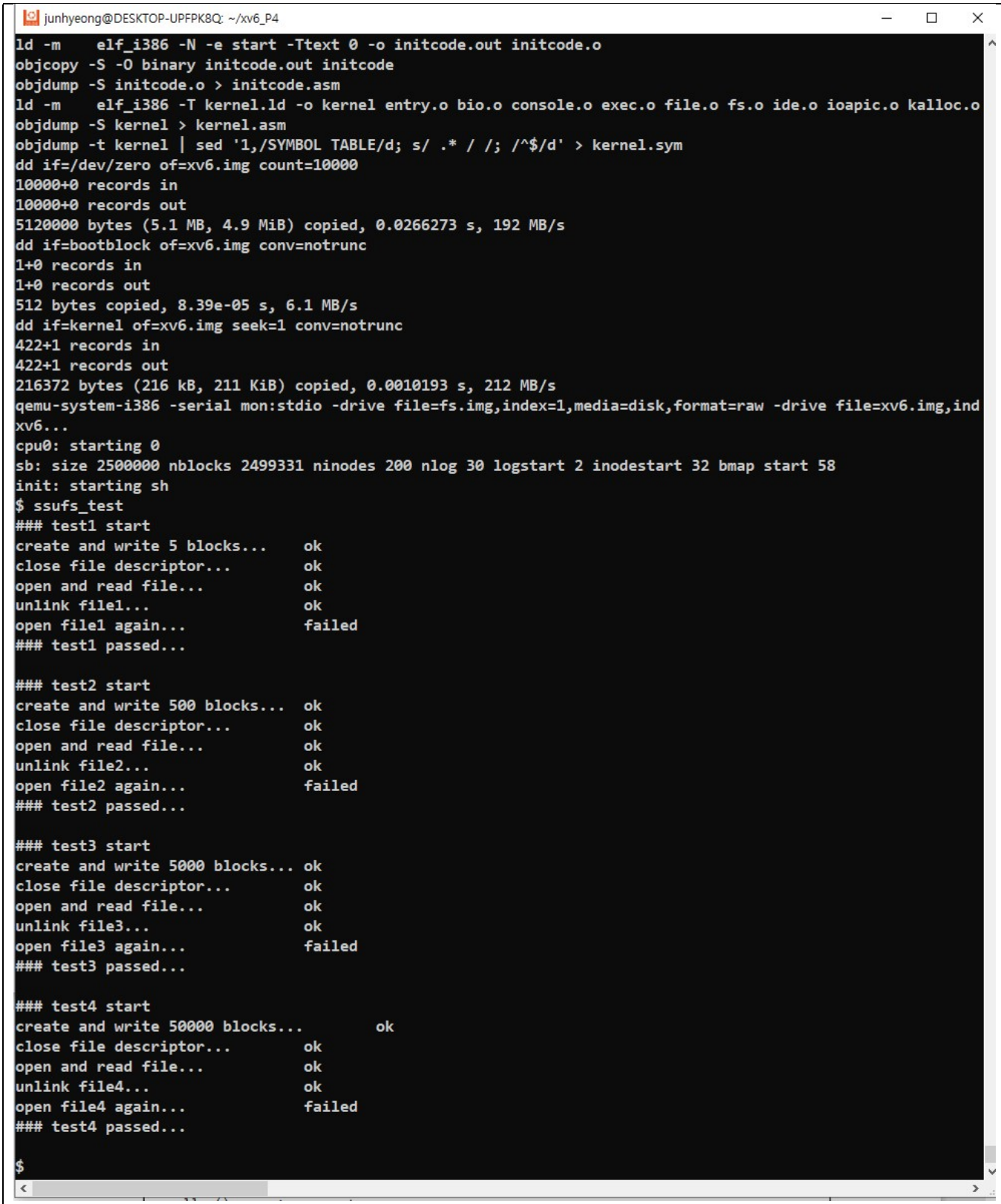
➢ **ssualloc_test.c 결과**



- 해당 ssualloc_test 프로세스에서 getvp() 에서 실제로 12KB 가 초기에 할당됨을 볼 수 있음
- 초기에 ssualloc(-1234) 를 할당하여 음수의 페이지를 할당하였으므로 할당실패함

- ssualloc(1234) 는 페이지크기 4096 의 배수가 아니여서 할당실패를 리턴
- ssualloc(4096) 는 페이지크기 4096 1 개를 가상메모리에만 할당하였으므로 가상페이지가 4, 물리페이지 개수가 3 개임을 볼 수 있음. 그리고 해당 주소에 'I' 를 수정하고 있으므로 trap 으로 PageFault 가 발생했고 해당 PageFault 로 실제 물리메모리 페이지가 할당되어 페이지 개수가 늘어난 점을 볼 수 있음
- ssualloc(12288) 는 페이지 크기 4096 3 개를 가상메모리에만 할당하였으므로 가상페이지가 7 개, 물리페이지는 3 개 적은 4 개로 할당된 모습을 볼 수 있음. 그리고 addr 의 각 페이지를 0, 10000, 8000 번 접근하였으므로 새롭게 할당된 페이지의 (1 번째, 3 번째, 2 번째) 페이지를 할당받고 접근할 때마다 할당된 물리페이지가 늘어난 모습을 볼 수 있음.

➢ **ssufs_test.c 결과**



- ssu_test 에서 테스트 하는 내용은 **파일 오픈 여부, 파일이 buf 개수만큼 써질 수 있는가?, 파일을 닫을 때 버퍼가 정상적으로 저장되는가?, 파일을 read 할 때 문제가 없는가? 파일을 지웠을 때 다시열었을 때 무난하게 failed 가 정상적으로 뜨는가? 체크이다.**

- ssu_test 의 페이지개수의 배수 (512*5, 512*500, 512*5000, 512*50000) 를 통해
DIRECTING 매핑, 2 단계-레벨 매핑, 3 단계-레벨매핑, 4 단계-레벨매핑 등을 테스트하고 있음을 알 수 있음.
실제로 50000 까지 무난하게 할당(bmap) 과 삭제(itrunc) 가 정상적으로 이루어지고있음을 볼 수 있음.

4. 소스코드

[가상메모리 ssualloc 관련]

mmu.h (PTE_LAZY 추가 *LAZY allocation 추가)

```
....
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_LAZY        0x008   // 페이지 테이블에 할당은 됐는데 trap 호출 시 할당하는 것으로 변경
#define PTE_PS          0x080   // Page Size
...
```

sysproc.c (sys_getpp, sys_getvp, sys_ssualloc() 추가)

```
int
sys_getpp()
{
  //memlayout.h 에서 사용자 영역 확인.
  pde_t *pgdir = myproc()->pgdir, pde;
  pde_t *pgtab;
  //사용자 영역은 0x00000000 ~ 0x80000000-1 까지임 (그 위로는 커널영역이 할당되어있음)
  //0x80000000 의 상위 10 비트 -> 0x200, 중간 10 비트 -> 0x400 까지로 계산
  int pde_point = 0x200, pte_point = 0x400, i,j;
  int retVal = 0;
  for (i = 0 ; i < pde_point ; i++) {
    if ((pde = pgdir[i]) & PTE_P) { //Page Directory 가 할당되어있는지 확인
      pgtab = (pte_t*)P2V(PTE_ADDR(pde)); //Page Table 할당
      for (j = 0 ; j < pte_point ; j++) {
        if (pgtab[j] & PTE_P) {  //PTE 가 할당되어있으면 페이지개수 증가
          retVal++;
        }
      }
    }
  }
  return retVal; //물리페이지 개수 리턴
}


/**
 * 가상메모리 할당 -> 직적 내가 페이지테이블에 할당을 박아넣을까 고민..
 * 교수님의 의도는 가상페이지 테이블에 받은 인자에 대한 페이지를 연속적으로 할당하는 것을 생각하신듯함
 * 그리고 trap.c 에서 page fault 에 대하여 그때서야 비로소 페이지테이블을 할당하라는 뜻 같음.
```

```
*/
int
sys_ssualloc()
{
  int size;
  int pagesize = PGSIZE;
  pde_t *pgdir = myproc()->pgdir, pde;
  pde_t *pgtab;
  argint(0, &size); //첫번째 인자에서 사이즈 받아오기

  //음수이거나 4 의 배수 아니면 일단 에러처리
  if (size < 0 || size % pagesize != 0)
    return -1;
  //cprintf("%d₩n",blockCount);


#ifdef ORIGIN
  //memlayout.h 에서 사용자 영역 확인.
  //사용자 영역은 0x00000000 ~ 0x40000000-1 까지임 (그 위로는 커널영역이 할당되어있음)
  //아래는 내가 임의로 페이지를 할당해서 집어넣는 과정
  int blockCount = size/pagesize; //할당받을 블록크기
  int pde_point = 1<<9, pte_point = 1<<10, i,j ,serial_count = 0;
  int save_i=-1, save_j=-1;
  for (i = 0 ; i < pde_point ; i++) {
    if ((pde = pgdir[i]) & (PTE_P)) {
      pgtab = (pte_t*)P2V(PTE_ADDR(pde)); //Page Table 할당
      for (j = 0 ; j < pte_point ; j++) {
          //LAZE 배치 뿐 아니라 P 비트가 할당안되면서 연속적인 공간을 찾아야함
        if (!(pgtab[j] & PTE_P) && !(pgtab[j] & PTE_LAZY)) { //단순히 물리메모리가 할당되었는지 판단
(PTE 검사안해도 됨)
          if (serial_count == 0) {
            save_i = i;
            save_j = j; //시작위치기록
          }
          serial_count++;
        } else {
          if (serial_count > 0) {
            serial_count = 0;
            save_i = -1;
            save_j = -1;
          }
        }
        if (serial_count >= blockCount) {
```

```
        break;
      }
    }
  }
}

//알맞은 공간을 못 찾았으면 pde 새로 하나 할당받아서 거기다 배치
if (save_i == -1 || save_j == -1) {
  for (i = 0 ; i < pde_point ; i++) {
    if (!((pde = pgdir[i]) & PTE_P)) {
      if((pgtab = (pte_t*)kalloc()) < 0)
        return -1;
      pgdir[i] = V2P(&pgtab[i]) | PTE_P | PTE_W | PTE_U; //페이지테이블 할당
      save_i = i;
      save_j = 0;
    }
  }
}

//이제 saved_i,j 에 가상페이지를 배치해보자
pde = pgdir[save_i];
pgtab = (pte_t*)P2V(PTE_ADDR(pde));
for (j = save_j, i = save_i ; j < save_j + blockCount ; j++) {
  pgtab[j] = PTE_LAZY; //pteLazy 하나 박아줌
}
myproc()->sz += size;
return PGADDR(save_i, save_j, 0);

//배치가 결정된 공간에 PTE_LAZY 설정하도록 하자
#else
//아래부터는 vm.c 의 함수들을 이용하여 myproc()->sz 를 이용하여 페이지 위치를 찾는 과정
//해당 방식이 원래 시스템과 더 잘 어울리며 더 안정적으로 돌아가도록 설계됨
int sz = myproc()->sz;
if (sz + size >= KERNBASE)
  return 0;
uint allocAddr = PGROUNDUP(sz); //현재 sz 위치에 페이지 할당 (새롭게 할당)
uint onlyReturnToUserProgramWithAllocationFirstAddress = allocAddr; //reutrn 할 주소
for (; allocAddr < sz + size ; allocAddr += PGSIZE)  { //allocuvm 참고
  pde = pgdir[PDX(allocAddr)]; //가상 페이지 확인
  //vm.c : walkpgdir 참고
  if (!(pde & PTE_P)) { //페이지 디렉토리 없으면 할당해야지?
    pgdir = &myproc()->pgdir[PDX(allocAddr)];
    if((pgdir = (pde_t*)kalloc()) <0) { //페이지가 할당되지 않으면 에러처리
```

```
        cprintf("I want to Allocate PDE. But, the system didn't help me.. bye₩n");
        return -1;
      }
      pgdir[PDX(allocAddr)] = V2P(pgdir[PDX(allocAddr)]) | PTE_P | PTE_W | PTE_U;  //pgdir 할당
    }

    //이제 saved_i,j 에 가상페이지를 배치해보자
    //vm.c : mappages() 참고
    pde = pgdir[PDX(allocAddr)]; //새롭게 할당받은 allocAddr 의 페이지디렉토리 가져오기
    pgtab = (pte_t*)P2V(PTE_ADDR(pde)); //페이지 테이블 받아오기 (해당 PDE 의)
    pgtab[PTX(allocAddr)] = PTE_LAZY; //나중에 물리메모리를 할당하겠다는 굳건한 의지를 박음
  }
  myproc()->sz = sz + size; //프로세스 사이즈 재갱신
  return onlyReturnToUserProgramWithAllocationFirstAddress;
#endif

}

int
sys_getvp()
{
  //memlayout.h 에서 사용자 영역 확인.
  pde_t *pgdir = myproc()->pgdir, pde;
  pde_t *pgtab;
  //사용자 영역은 0x00000000 ~ 0x80000000-1 까지임 (그 위로는 커널영역이 할당되어있음) ->ELF 포멧
구조때 참조
  int pde_point = 0x200, pte_point = 0x400, i,j;
  int retVal = 0;
  for (i = 0 ; i < pde_point ; i++) {
    if ((pde = pgdir[i]) & PTE_P) { //페이지 디렉토리가 할당되어있는지 확인
      pgtab = (pte_t*)P2V(PTE_ADDR(pde)); //PDE 에 할당된 Page Table 개수 확인
      for (j = 0 ; j < pte_point ; j++) {
        //PTE_P 뿐 아니라 PTE_LAZY (Lazy Allocation) 둘 다 가상메모리페이지 개수로 사용할 수 있음
        if ((pgtab[j] & PTE_P) ||(pgtab[j] & PTE_LAZY)) {
          retVal++;
        }
      }
    }
  }
  return retVal; //가상 페이지 개수 리턴
  //프로세스 페이지테이블의 할당된 물리페이지 수를 의미하는듯
}
```

trap.c

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
//int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;
uint va;

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}

//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  char* pa;
  pde_t* pdir, pde;
```

```
  pte_t* pgtab, pte;
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x₩n",
            cpuid(), tf->cs, tf->eip);
    lapiceoi();
    break;
  case T_PGFLT:
```

```
    pdir = myproc()->pgdir;
    va = rcr2();  //페이지폴트 예외에서의 가상주소를 읽어오는 명령어로 추정
    va = PGROUNDDOWN(va); //가상주소 변경
    //cprintf("Page Fault : %d₩n", va); //확인해보니까 맞음
    pde = pdir[PDX(va)];
    pgtab = (pte_t*)P2V(PTE_ADDR(pde));
    pte = pgtab[PTX(va)];
    if (pte & PTE_LAZY) {
      pa = kalloc(); //물리 메모리 할당
      memset((void*)pa, 0, (uint)PGSIZE);
      //해당 가상메모리를 PGSIZE (4096) 만큼 V2P(커널영역->사용장여역) 으로 할당하고 PTE_W, PTE_U
플래그 설정
      mappages(myproc()->pgdir, (char*)va, PGSIZE, V2P(pa), PTE_W|PTE_U);
      break;
    }
  //PAGEBREAK: 13
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)₩n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc₩n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
```

```
  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

[시스템 콜 구현 관련]

syscall.c, syscall.h, user.h, usys.S

[syscall.c]

```
...
extern int sys_getvp(void);
extern int sys_getpp(void);
extern int sys_ssualloc(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_getvp]   sys_getvp,
[SYS_getpp]   sys_getpp,
[SYS_ssualloc] sys_ssualloc,
};
...
[syscall.h]
...
#define SYS_getvp  22
```

```
#define SYS_getpp  23
#define SYS_ssualloc  24
...
[user.h]
...
//#P4 시스템 콜
int getvp();
int getpp();
int ssualloc(int);
...
[usys.S]
...
SYSCALL(getvp)
SYSCALL(getpp)
SYSCALL(ssualloc)
...
```

[파일시스템 구현 관련]

fs.h (NDIRECT, INDIRECT 매크로 변경 및 수정)

```
...
//#define NDIRECT 12
#define NDIRECT      6

#define NINDIRECT (BSIZE / sizeof(uint)) //주소개수를 넣을 수 있는 개수
#define LEVEL1       NINDIRECT*4                    // 6,7,8,9
#define LEVEL2       NINDIRECT*NINDIRECT*2          // 10,11
#define LEVEL3       NINDIRECT*NINDIRECT*NINDIRECT  // 12
#define MAXFILE (NDIRECT + LEVEL1 + LEVEL2 + LEVEL3)
....
```

fs.c (bmap, itrunc 수정)

```
...
/**
 * P4 과제를 위한 수정
 * bn 은 블록번호를 가리킴.. -> 차례차례 접근하도록 설정
*/
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;
  int idx_lvl1=-1, idx_lvl2, idx_lvl3;       //level 접근을 위한 인덱스
  if(bn < NDIRECT){
```

```
  if((addr = ip->addrs[bn]) == 0)
    ip->addrs[bn] = addr = balloc(ip->dev);
  return addr;
}
bn -= NDIRECT;  //첫 번째 블록만 접근
//level 1 블록
if(bn < LEVEL1){
  /*
  int db=0;
  for (db = 0 ; db < 13 ; db++)
    cprintf("ip->addrs : [%d]%d₩n",db, ip->addrs[db]);
  */
  for (idx_lvl1 = NDIRECT ; bn >= NINDIRECT; idx_lvl1++, bn-=NINDIRECT);
  //디렉토리 할당부분
  if((addr = ip->addrs[idx_lvl1]) == 0) {
    ip->addrs[idx_lvl1] = addr = balloc(ip->dev); //디렉토리 생성에는 저널링을 안함
  }

  //디스크블록을 읽는 부분
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data; //데이터를 벡터화 (128 idx 를 가지는 주소배열)
  if((addr = a[bn]) == 0){ //막상가봤더니 없네? -> 할당 후 log 재정리
    a[bn] = addr = balloc(ip->dev);
    log_write(bp); //저널링
  }
  brelse(bp); //저널링 풀기
  //cprintf("my block : idx1:%d, bn:%d₩n", idx_lvl1, bn);
  return addr;
}

bn -= LEVEL1;       //3-LEVEL 메모리
if (bn < LEVEL2) {
  for (idx_lvl2 = 6+4 ; bn >= NINDIRECT*NINDIRECT ; idx_lvl2++, bn-=NINDIRECT*NINDIRECT);
  //idx_lvl2 할당되었는지 확인 (가장 처음 프레임)
  if((addr = ip->addrs[idx_lvl2]) == 0)
    ip->addrs[idx_lvl2] = addr = balloc(ip->dev); //디렉토리 생성에는 저널링을 안함

  for (idx_lvl1=0 ; bn >= NINDIRECT ; idx_lvl1++, bn -=NINDIRECT);

  //idx_lvl1 할당되었는지 확인
  bp = bread(ip->dev, addr); //첫 번째 디렉토리 정보를 받아옴
  a = (uint*)bp->data; //데이터를 벡터화 (128 idx 를 가지는 주소배열)
  if((addr = a[idx_lvl1]) == 0){ //막상가봤더니 없네? -> 할당 후 log 재정리
```

```
    a[idx_lvl1] = addr = balloc(ip->dev);
    log_write(bp); //저널링 -> 오류 생길 수 있음!
  }
  brelse(bp); //저널링 풀기

  //idx_lvl0 실제 데이터 블록부분 가져오기
  bp = bread(ip->dev, addr); //첫 번째 디렉토리 정보를 받아옴
  a = (uint*)bp->data; //데이터를 벡터화 (128 idx 를 가지는 주소배열)
  if((addr = a[bn]) == 0){ //막상가봤더니 없네? -> 할당 후 log 재정리
    a[bn] = addr = balloc(ip->dev);
    log_write(bp); //저널링 -> 오류 생길 수 있음!
  }
  brelse(bp); //저널링 풀기
  return addr;
}

bn -= LEVEL2;       //4-LEVEL 메모리
if (bn < LEVEL3) {
  //addr[12] 할당되었는지 확인 (가장 처음 프레임)
  idx_lvl3 = 6+4+2;
  if((addr = ip->addrs[idx_lvl3]) == 0)
    ip->addrs[idx_lvl3] = addr = balloc(ip->dev); //디렉토리 생성에는 저널링을 안함

  for (idx_lvl2=0 ; bn >= NINDIRECT*NINDIRECT ; idx_lvl2++, bn -=NINDIRECT*NINDIRECT);
  //idx_lvl2 할당되었는지 확인
  bp = bread(ip->dev, addr); //첫 번째 디렉토리 정보를 받아옴
  a = (uint*)bp->data; //데이터를 벡터화 (128 idx 를 가지는 주소배열)
  if((addr = a[idx_lvl2]) == 0){ //막상가봤더니 없네? -> 할당 후 log 재정리
    a[idx_lvl2] = addr = balloc(ip->dev);
    log_write(bp); //저널링 -> 오류 생길 수 있음!
  }
  brelse(bp); //저널링 풀기

  for (idx_lvl1=0 ; bn >= NINDIRECT ; idx_lvl1++, bn-=NINDIRECT);
  //idx_lvl1 할당되었는지 확인
  bp = bread(ip->dev, addr); //첫 번째 디렉토리 정보를 받아옴
  a = (uint*)bp->data; //데이터를 벡터화 (128 idx 를 가지는 주소배열)
  if((addr = a[idx_lvl1]) == 0){ //막상가봤더니 없네? -> 할당 후 log 재정리
    a[idx_lvl1] = addr = balloc(ip->dev);
    log_write(bp); //저널링 -> 오류 생길 수 있음!
  }
  brelse(bp); //저널링 풀기
```

```
    //idx_lvl0 실제 데이터 블록부분 가져오기
    bp = bread(ip->dev, addr); //첫 번째 디렉토리 정보를 받아옴
    a = (uint*)bp->data; //데이터를 벡터화 (128 idx 를 가지는 주소배열)
    if((addr = a[bn]) == 0){ //막상가봤더니 없네? -> 할당 후 log 재정리
      a[bn] = addr = balloc(ip->dev);
      log_write(bp); //저널링 -> 오류 생길 수 있음!
    }
    brelse(bp); //저널링 풀기

    return addr;
  }
  panic("bmap: out of range");
}
```

```
// Truncate inode (discard contents).
// Only called when the inode has no links
// to it (no directory entries referring to it)
// and has no in-memory reference to it (is
// not an open file or current directory).
static void
itrunc(struct inode *ip)
{
  int i, j, l, m;
  struct buf *bp, *bp2, *bp3;
  uint *a, *a2, *a3;
#ifdef JHS
  int db = 0;
  for (db = 0 ; db < 13 ; db++)
    cprintf("[%d] [%d]->%d₩n", ip->inum, db, ip->addrs[db]);
#endif
  //1 단계 Directing Mapping 할당해제
  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){ //0,1,2,3,4,5 idx 에대하여 할당해제
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  //6,7,8,9 에 해당하는 2-level mapping system 해제
```

```
  for (i = NDIRECT; i < NDIRECT + 4; i++) {
    if (ip->addrs[i]) {
      bp = bread(ip->dev, ip->addrs[i]); //페이지를 읽어옴
      a = (uint*)bp->data;
      //차례차례접근하며 페이지에 Indirecting 하게접근된 페이지 할당해제
      for (j = 0; j < NINDIRECT; j++) {
        if (a[j]) {
          bfree(ip->dev, a[j]);
        }
      }
      brelse(bp);
      //차례차례접근하며 페이지에 Indirecting 하게접근된 페이지 디렉토리 할당해제
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0; //다음 bmap 을 위해 0 으로 초기화
    }
  }
  //3-Level Mapping System 할당해제 () 10,11 idx
  for (i = NDIRECT+4 ; i < NDIRECT+6 ; i++) {
    if (ip->addrs[i]) {
      //3 단계의 가장 초기 부모 맵 페이지 접근
      bp = bread(ip->dev, ip->addrs[i]);
      a = (uint*)bp->data;
      // 3 단계의 2 번째의 디렉토리 페이지 접근
      for (j = 0; j < NINDIRECT; j++) {
        //디렉토리 페이지가 있다면?
        if (a[j]) {
          //디렉토리 페이지를 접근하여 INDIRECING page 할당해제
          bp2 = bread(ip->dev, a[j]);
          a2 = (uint*)bp2->data ;
          for (l = 0 ; l < NINDIRECT ; l++) {
            if (a2[l]) {
              bfree(ip->dev, a2[l]);
            }
          }
          brelse(bp2); //a[j] 접근이 끝났으므로 버퍼 캐시에서 비워주기
          bfree(ip->dev, a[j]); //2 번째 디렉토리 페이지를 할당해제
        }
      }
      brelse(bp);
      bfree(ip->dev, ip->addrs[i]); //1 번째 디렉토리 페이지를 할당해제
      ip->addrs[i] = 0; //다음 bmap 을 위해 0 으로 초기화
    }
  }
```

```
  //4-Level Mapping System 할당해제 (12 번 idx)
  i = NDIRECT+6;
  if(ip->addrs[i]){ //12 번 idx 가 할당되어있다면 4-Level Mapping 간주
    if (ip->addrs[i]) {
      bp = bread(ip->dev, ip->addrs[i]); //4 레벨의 첫번째 디렉토리 페이지 접근
      a = (uint*)bp->data;
      //하나씩 포인터로 연결해가며 4 레벨의 두번째 디렉토리로 페이지 접근
      for (j = 0; j < NINDIRECT; j++) {
        if (a[j]) { //4 레벨 디렉토리 중 2 번째 디렉토리의 각 요소 포인터로 접근
          bp2 = bread(ip->dev, a[j]);
          a2 = (uint*)bp2->data;
          for (l = 0 ; l < NINDIRECT ; l++) {
            if (a2[l]) { //3 번째 디렉토리 페이지가 존재한다면 마지막 진짜 Page 접근을 위해 접근
              bp3 = bread(ip->dev, a2[l]);
              a3 = (uint*)bp3->data;
              //3 번째 디렉토리에서 -> 실제 페이지 할당해제를 위한 Indirecting Mapping 할당해제
              for (m = 0 ; m < NINDIRECT ; m++) {
                if (a3[m]){
                  bfree(ip->dev, a3[m]); //각 실제요소 할당해제
                }
              }
              brelse(bp3); //더이상 해당 3 번째 디렉토리 페이지 접근이 끝나서 캐시에서 비워주기
              bfree(ip->dev, a2[l]); //3 번째 디렉토리 페이지를 할당해제
            }
          }
          brelse(bp2); //더이상 2 번째 디렉토리 페이지 접근이 끝나서 캐시에서 비워주기
          bfree(ip->dev, a[j]); //2 번째 디렉토리 페이지를 할당해제
        }
      }
      brelse(bp); //더이상 idx 12 접근이 없으므로 캐시에서 비워주기
      bfree(ip->dev, ip->addrs[i]); //1 번째 디렉토리 (가장 앞단) 할당해제
      ip->addrs[i] = 0; //인덱스 초기화
    }
  }
  ip->size = 0;
  iupdate(ip);
}
...
```

file.h (inode 구조체 수정)

```
...
struct inode {
  uint dev;           // Device number
```

```
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+7]; //ndirect 개수 변환.
};
...
```

param.h

```
...
#define FSSIZE       2500000  // P4 과제를 위한 Filesystem 파일크기 변경
```

mkfs.c (수정한 부분만 첨삭) > 초기 부팅 시 파일 시스템에 올릴 때 사용

```
....
#define JH  0     //디버그용 매크로
#define MAXFILE_SIZE  (6 + NINDIRECT*4 + NINDIRECT*NINDIRECT*2 +
NINDIRECT*NINDIRECT*NINDIRECT) //블록 총 개수
...
void
balloc(int used)
{
  //zero block 처리
  uchar buf[BSIZE]; //Zero Block 상한 정리
  int i, j, count = 0;

  printf("balloc: first %d blocks have been allocated₩n", used);
  assert(used < MAXFILE_SIZE);
  //count 만큼 bmapstart 앞부분에 쓰기 작업은 계속함
  for (j = 0 ; count < used; j++) {
    bzero(buf, BSIZE);
    //count < used 이면서 앞의 block bit 를 하나씩 채움
    //count 는 블록을 쓸 때마다 하나씩 증가
    //bmap 은 비트단위로 계산하기 때문에 비트연산자체는 동일하게 진행
    for (i = 0; i < BSIZE*8 && count < used; i++, count++) {
      buf[i / 8] = buf[i / 8] | (0x1 << (i % 8));
    }
    wsect(sb.bmapstart + j, buf); //bitmap 초기화 계속 설정
  }
```

```
  printf("balloc: write bitmap block at sector %d₩n", sb.bmapstart);
}

#define min(a, b) ((a) < (b) ? (a) : (b))

//inode 초기화할 때 사용하는 것으로 추측중
void
iappend(uint inum, void *xp, int n)
{
  char *p = (char*)xp;
  uint fbn, off, n1;
  struct dinode din;
  char buf[BSIZE];
  uint tmp, tmp2, tmp3;
  //uint tmp1;
  uint indirect[NINDIRECT];
  uint indirect1[NINDIRECT];
  uint indirect2[NINDIRECT];
  /*
  uint indirect3[NINDIRECT];
  */
  uint x;

  int idx1 = NDIRECT;
  //n += 30;

  rinode(inum, &din);
  off = xint(din.size);
#if JH
  printf("Block Count : %u₩n", off/BSIZE);
#endif
  while(n > 0){
    fbn = off / BSIZE;
    assert(fbn < MAXFILE);

    //직접 매핑할 때 호출되는 부분
    if(fbn < NDIRECT){
#if JH
      printf("first NDIRECT : %d₩n", fbn);
#endif
      //xint -> addr[fbn] 블록의 주소를 Little Endian 으로 바꿔주는 과정
      if(xint(din.addrs[fbn]) == 0){ // 만약에 addrs 가 비어있으면 freeblock 을 자동으로 할당해주는 부분
        din.addrs[fbn] = xint(freeblock++);
```

```
    }

    //freeblock 을 할당하고 주소를 받아옴
    x = xint(din.addrs[fbn]);
  }
  else if (fbn < (NDIRECT + LEVEL1)){ //2-level 매핑할 때 호출되는 부분
#if JH
    printf("1 LEVEL INDIRECT : %d₩n", fbn);
#endif
    idx1 = NDIRECT + (fbn-NDIRECT) / NINDIRECT; //idx1 는 실제로는 몇 번쨰 indirect 인가로 접근
    tmp = fbn - NDIRECT; //INDIRECT 블록번호를 tmp 로 구해옴 -> 물론 128 * 4 개씩있으니까 나머지로
구해옴

    if(xint(din.addrs[idx1]) == 0){
      din.addrs[idx1] = xint(freeblock++);
    }
    //Indirect 블록을 읽기위해 데이터블록을 읽어옴
    rsect(xint(din.addrs[idx1]), (char*)indirect);
    //만약에 indirect 부분이 없는 블록이었다?
    if(indirect[tmp%NINDIRECT] == 0){ //-> 해당 블록에 freeblock 을 할당해줌
      indirect[tmp%NINDIRECT] = xint(freeblock++);
      //해당 indirect 페이지에 freeblock 만큼 재갱신함 (위와달리 디스크에 포인터연결을 하니까
갱신해줘야됨)
      wsect(xint(din.addrs[idx1]), (char*)indirect);
    }
    x = xint(indirect[tmp%NINDIRECT]); //freeblock 을 받아옴
  }
  //3 단계
  else if (fbn < (NDIRECT + LEVEL1 + LEVEL2)) {
    idx1 = NDIRECT + 4 + (fbn-NDIRECT-NINDIRECT*4) / (NINDIRECT*NINDIRECT); //3-level Indirect 부분
    // tmp = ((0 ~ 2*N*N-1) / N) % N (첫 번째 INDIRECT 주소)
    // tmp2 = (0 ~ 2*N*N-1) % N;
    tmp = ((fbn-NDIRECT-NINDIRECT*4) / NINDIRECT) % NINDIRECT;
    tmp2 = (fbn-NDIRECT-NINDIRECT*4) % NINDIRECT;
#if JH
    printf("2 LEVEL INDIRECT : %d₩n", fbn);
    printf("2 LEVEL INDIRECT : idx1 = %d, tmp = %d, tmp2 = %d₩n", idx1, tmp, tmp2);
#endif

    if (xint(din.addrs[idx1]) == 0) {
      din.addrs[idx1] = xint(freeblock++);
    }
    rsect(xint(din.addrs[idx1]), (char*)indirect); //1 단계 디렉토리 확인
```

```
      if (indirect[tmp] == 0) { //없으니까 freeblock 할당하고 wsect
        indirect[tmp] = xint(freeblock++);
        //해당 indirect 페이지에 freeblock 만큼 재갱신함 (위와달리 디스크에 포인터연결을 하니까
갱신해줘야됨)
        wsect(xint(indirect[tmp]), (char*)indirect);
      }

      rsect(xint(indirect[tmp]), (char*)indirect1); //2 단계 디렉토리 확인
      if (indirect1[tmp2] == 0) { //없으니까 freeblock 할당하고 wsect
        indirect1[tmp2] = xint(freeblock++);
        //해당 indirect 페이지에 freeblock 만큼 재갱신함
        wsect(xint(indirect1[tmp2]), (char*)indirect1);
      }
      x = xint(indirect1[tmp2]);
    }
    else { //4 단계 디렉토리 (실제로 선 프로그램이 2 기가? ㅋㅋㅋ)
      idx1 = 12; //어차피 마지막
      tmp = ((fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) / (NINDIRECT*NINDIRECT)); // 1 단계
디렉토리 idx -> 얜 1 개뿐이라 / 안해줘도됨
      tmp2 = ((fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) / (NINDIRECT)) % NINDIRECT;  // 2 단계
디렉토리 idx
      tmp3 = (fbn-NDIRECT-NINDIRECT*4-NINDIRECT*NINDIRECT*2) % NINDIRECT; //3 단계 디렉토리 주소
#if JH
      printf("3 LEVEL INDIRECT : %d₩n", fbn);
      printf("3 LEVEL INDIRECT : idx1 = %d, tmp = %d, tmp2 = %d, tmp3 = %d₩n", idx1, tmp, tmp2, tmp3);
#endif

      if (xint(din.addrs[idx1]) == 0) {
        din.addrs[idx1] = xint(freeblock++);
      }
      rsect(xint(din.addrs[idx1]), (char*)indirect); //1 단계 디렉토리 확인
      if (indirect[tmp] == 0) { //없으니까 freeblock 할당하고 wsect
        indirect[tmp] = xint(freeblock++);
        //해당 indirect 페이지에 freeblock 만큼 재갱신함 (위와달리 디스크에 포인터연결을 하니까
갱신해줘야됨)
        wsect(xint(indirect[tmp]), (char*)indirect);
      }

      rsect(xint(indirect[tmp]), (char*)indirect1); //2 단계 디렉토리 확인
      if (indirect1[tmp2] == 0) { //없으니까 freeblock 할당하고 wsect
        indirect1[tmp2] = xint(freeblock++);
        //해당 indirect 페이지에 freeblock 만큼 재갱신함
        wsect(xint(indirect1[tmp2]), (char*)indirect1);
```

```
      }

      rsect(xint(indirect1[tmp2]), (char*)indirect2); //3 단계 디렉토리 확인
      if (indirect2[tmp3] == 0) { //없으니까 freeblock 할당하고 wsect
        indirect2[tmp3] = xint(freeblock++);
        //해당 indirect 페이지에 freeblock 만큼 재갱신함
        wsect(xint(indirect2[tmp3]), (char*)indirect2);
      }
      x = xint(indirect2[tmp3]);
    }


    //무조건 실행해야하는 부분 -> x buf 를 받아와서 p 를 copy 로 받아오고 옮겨놓는 과정
    n1 = min(n, (fbn + 1) * BSIZE - off);
    rsect(x, buf);
    bcopy(p, buf + off - (fbn * BSIZE), n1);
    wsect(x, buf);
    n -= n1;
    off += n1;
    p += n1;
  }
  din.size = xint(off);
  winode(inum, &din);
}
```

vm.c (trap.c 에서 mappages 를 사용하기 위하여 static 제거)

```
....
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va); //가상메모리 하위 주소 (페이지값 날리고) 구해옴
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1); //이전 가상메모리 주소 구해옴
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0) //PDE 없을 시 할당 및 내부과정(PDX,PTX) 를 통해 PTE 추출
      return -1;
    if(*pte & PTE_P) //PTE 가 이미 할당된 페이지면 panic 에러 출력
      panic("remap");
    *pte = pa | perm | PTE_P; //pte 에 PTE_P 플래그 설정
    if(a == last) //페이지가 2 개이상이면 다시 반복
      break;
```

```
    a += PGSIZE;
    pa += PGSIZE;
  }


  return 0;
}
...
```

defs.h 수정부분 (trap.c 에서 mappages() 사용을 위해 선언부 추가)

```
...
//vm.c
...
int             mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

Makefile (사용자 프로그램 : ssufs_test.c 와 ssualloc_test 추가를 위해 추가)

```
...
UPROGS=₩
....
    _ssufs_test₩
    _ssualloc_test₩
....
```

[교수님이 제작한 검사 프로그램]

ssufs_test.c

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

#define BSIZE 512

char buf[BSIZE];

void _error(const char *msg) {
        printf(1, msg);
        printf(1, "ssufs_test failed...₩n");
        exit();
}

void _success() {
        printf(1, "ok₩n");
}
```

```
void test(int ntest, int blocks) {
        char filename[16] = "file";
        int fd, i, ret = 0;

        filename[4] = (ntest % 10) + '0'; //파일 이름 설정

        printf(1, "### test%d start₩n", ntest);
        //파일 생성여부 테스트
        printf(1, "create and write %d blocks...₩t", blocks);
        fd = open(filename, O_CREATE | O_WRONLY);

        if (fd < 0)
                _error("File open error₩n");

        //파일 생성을 위해 블록수만큼 write 시도 (실질적 파일 검사) -> bmap 검사
        for (i = 0; i < blocks; i++) {
                ret = write(fd, buf, BSIZE);
                if (ret < 0) break;
        }
        if (ret < 0)
                _error("File write error₩n");
        else
                _success(); //파일 성공여부 출력

        printf(1, "close file descriptor...₩t");

        //파일 디스크립터 닫기 시도
        if (close(fd) < 0)
                _error("File close error₩n");
        else
                _success();

        //파일 열기와 읽기 시도 -> Filesystem 으로 망가진게 아닌가 검사
        printf(1, "open and read file...₩t₩t");
        fd = open(filename, O_RDONLY);

        if (fd < 0)
                _error("File open error₩n");

        //블록수만큼 파일 읽기 검사  -> 기본적으로 bmap 함수랑 동일한 메커니즘
        for (i = 0; i < blocks; i++) {
                ret = read(fd, buf, BSIZE);
                if (ret < 0) break;
```

```
        }
        if (ret < 0)
                _error("File read error₩n");

        //파일 닫기 시도
        if (close(fd) < 0)
                _error("File close error₩n");
        else
                _success();

        printf(1, "unlink %s...₩t₩t₩t", filename);
        //파일 제거 시도 (블록할당해제 : itrunc 함수 검사)
        if (unlink(filename) < 0)
                _error("File unlink error₩n");
        else
                _success();

        printf(1, "open %s again...₩t₩t", filename);
        fd = open(filename, O_RDONLY);
        //파일이 지워졌는데 남아있는지 검사 : unlink 검사
        if (fd < 0)
                printf(1, "failed₩n");
        else
                printf(1, "this statement cannot be runned₩n");

        printf(1, "### test%d passed...₩n₩n", ntest);
}

int main(int argc, char **argv)
{
        for (int i = 0 ; i < BSIZE; i++) {
                buf[i] = BSIZE % 10;
        }

        test(1, 5); //5 번 직접블록검사
        test(2, 500); //6,7,8 번 등 2-level 파일 시스템 검사
        test(3, 5000); //10,11 번 3-Level 파일 시스템 검사
        test(4, 50000);      //12 번까지 Write(시간 엄청걸림) : 4-Level 파일 시스템 검사
        exit();
}
```

ssualloc_test.c

```
#include "types.h"
#include "stat.h"
```

```
#include "user.h"
#include "fcntl.h"

int main(void)
{
	int ret;
	//가상메모리 검사 (물리메모리페이지와 가상메모리 체크)
	printf(1, "Start: memory usages: virtual pages: %d, physical pages: %d₩n", getvp(), getpp());
	//가상메모리 할당 시도 -> 음수라서 에러처리
	ret = ssualloc(-1234);

	if(ret < 0)
		printf(1, "ssualloc() usage: argument wrong...₩n");
	else
		exit();

	//가상메모리 할당 시도 -> 페이지크기 배수가 아니라서 에러처리
	ret = ssualloc(1234);

	if(ret < 0)
		printf(1, "ssualloc() usage: argument wrong...₩n");
	else
		exit();

	//가상메모리에만 페이지 1 개 추가
	ret = ssualloc(4096);

	if(ret < 0 )
		printf(1, "ssualloc(): failed...₩n");
	else {
		//가상메모리 만 1 개추가 (물리메모리는 추가되면 안됨)
		printf(1, "After allocate one virtual page: virtual pages: %d, physical pages: %d₩n", getvp(),
getpp());
		char *addr = (char *) ret;

		//해당페이지 접근 후 물리메모리가 추가되는지 검사
		addr[0] = 'I';
		printf(1, "After access one virtual page: virtual pages: %d, physical pages: %d₩n", getvp(),
getpp());
	}

	//가상메모리 할당 : 페이지 3 개 추가
	ret = ssualloc(12288);
```

```
        if(ret < 0 )
                printf(1, "ssualloc(): failed...₩n");
        else {
                //가상메모리만 3 개 추가되야함
                printf(1, "After allocate three virtual pages: virtual pages: %d, physical pages: %d₩n",
getvp(), getpp());
                char *addr = (char *) ret;

                //각 페이지마다 하나씩 추가되면서 물리메모리 개수가 1 개씩 증가하는지 확인
                addr[0] = 'a';
                printf(1, "After access of first virtual page: virtual pages: %d, physical pages: %d₩n",
getvp(), getpp());
                addr[10000] = 'b';
                printf(1, "After access of third virtual page: virtual pages: %d, physical pages: %d₩n",
getvp(), getpp());
                addr[8000] = 'c';
                printf(1, "After access of second virtual page: virtual pages: %d, physical pages: %d₩n",
getvp(), getpp());
        }

        exit();
}
```