컴퓨터학부 20221867 이채은

**1. 개요**

-이 과제는 xv6 운영체제의 메모리 관리와 파일 시스템을 확장하는 것을 목표로 한다.

- (3-1)에서는 ssusbrk() 시스템 콜을 구현한다. 이 시스템 콜은 지연 메모리 할당 및 해제 기능을 제공하며, 가상 메모리는 즉시 할당하지만 물리 메모리는 실제 접근이 발생할 때 할당되도록 설계되었다. 또한 memstat() 시스템 콜을 통해 현재 할당된 가상 메모리와 물리 메모리의 정보를 확인할 수 있다.

- (3-2)에서는 간단한 소프트웨어 MMU(Memory Management Unit)를 구현한다. 이는 가상 주소를 물리 주소로 변환하는 기능을 담당하며, 페이지 테이블 구조를 이해하고 관리하는 것이 핵심이다. 구현해야 할 주요 함수로는 페이지 테이블 할당을 위한 alloc\_page\_table()과 주소 변환을 위한 mmu\_address\_translation()이 있다.

-(3-3)은 다단계 파일 시스템과 페이지 캐시를 구현하는 것이다. 기존 xv6의 파일 시스템을 확장하여 더 큰 파일을 지원할 수 있도록 3단계 간접 블록 구조를 도입하며, 레드블랙트리를 이용한 페이지 캐시를 구현하여 디스크 접근을 최적화한다. lseek(), rb\_print(), rb\_count() 등의 시스템 콜을 구현하여 파일 시스템의 상태를 확인하고 제어할 수 있다.

**2. 상세 설계 & 3. 실행 결과 & 4. 주석 달린 소스코드**

**- 3-1. Multi-level 파일시스템 구현**

- (정의) 3-level 파일시스템 구현

- (기능 1) 6개(addrs[0] ~ addrs[5])의 direct 블록, 4개(addrs[6] ~ addrs[9])의 indirect 블록, 2개(addrs[10] ~ addrs[11])의 2-level indirect 블록, 1개(addrs[12])의 3-level indirect 블록을 가지는 파일시스템을 구현

- 주석을 달아 상세하게 설명

|  |
| --- |
| fs.c에서 수정한 내용: bmap()과itrunc() 부분 |
| static uint  bmap(struct inode \*ip, uint bn)  {  uint addr, \*a, \*b, \*c;  struct buf \*bp, \*bp2, \*bp3;  // 직접 블록 처리  if(bn < NDIRECT){ // 직접 블록 (Direct blocks)  if((addr = ip->addrs[bn]) == 0) // 해당 블록이 아직 할당되지 않았으면  ip->addrs[bn] = addr = balloc(ip->dev); // 블록을 할당  return addr; // 블록 주소 반환  }  bn -= NDIRECT;  // 단일 간접 블록 처리  if(bn < NINDIRECT \* 4){ // 단일 간접 블록 (Indirect blocks)  uint indirect\_index = NDIRECT + (bn / NINDIRECT); // 간접 블록 테이블의 인덱스 계산  bn = bn % NINDIRECT; // 해당 블록의 오프셋 계산    if((addr = ip->addrs[indirect\_index]) == 0) // 간접 블록 테이블이 할당되지 않았으면  ip->addrs[indirect\_index] = addr = balloc(ip->dev); // 테이블 블록 할당    bp = bread(ip->dev, addr); // 블록을 읽어옴  a = (uint\*)bp->data; // 블록 데이터를 배열로 취급  if((addr = a[bn]) == 0){ // 오프셋 위치의 블록이 아직 할당되지 않았으면  a[bn] = addr = balloc(ip->dev); // 새로운 블록 할당  log\_write(bp); // 로그에 기록  }  brelse(bp); // 버퍼 해제  return addr; // 블록 주소 반환  }  bn -= NINDIRECT \* 4;  // 이중 간접 블록 처리  if(bn < NDINDIRECT \* 2){ // 이중 간접 블록 (Double indirect blocks)  uint dindex = 10 + (bn / NDINDIRECT); // 이중 간접 블록 테이블 인덱스 계산  bn = bn % NDINDIRECT; // 오프셋 계산    if((addr = ip->addrs[dindex]) == 0) // 이중 간접 블록 테이블이 할당되지 않았으면  ip->addrs[dindex] = addr = balloc(ip->dev); // 테이블 블록 할당    bp = bread(ip->dev, addr); // 테이블 블록 읽어옴  a = (uint\*)bp->data; // 데이터를 배열로 취급  uint index1 = bn / NINDIRECT; // 첫 번째 간접 블록 인덱스 계산  uint index2 = bn % NINDIRECT; // 두 번째 간접 블록 인덱스 계산    if((addr = a[index1]) == 0){ // 첫 번째 간접 블록이 아직 할당되지 않았으면  a[index1] = addr = balloc(ip->dev); // 할당  log\_write(bp); // 로그에 기록  }  bp2 = bread(ip->dev, addr); // 첫 번째 간접 블록 읽어옴  b = (uint\*)bp2->data; // 데이터를 배열로 취급    if((addr = b[index2]) == 0){ // 두 번째 간접 블록이 아직 할당되지 않았으면  b[index2] = addr = balloc(ip->dev); // 할당  log\_write(bp2); // 로그에 기록  }    brelse(bp2); // 버퍼 해제  brelse(bp); // 버퍼 해제  return addr; // 블록 주소 반환  }  bn -= NDINDIRECT \* 2;  // 삼중 간접 블록 처리  if(bn < NTINDIRECT){ // 삼중 간접 블록 (Triple indirect block)  if((addr = ip->addrs[12]) == 0) // 삼중 간접 블록 테이블이 아직 할당되지 않았으면  ip->addrs[12] = addr = balloc(ip->dev); // 테이블 블록 할당    bp = bread(ip->dev, addr); // 삼중 간접 블록 테이블 읽어옴  a = (uint\*)bp->data; // 데이터를 배열로 취급  uint index1 = bn / (NINDIRECT \* NINDIRECT); // 첫 번째 인덱스 계산  uint index2 = (bn % (NINDIRECT \* NINDIRECT)) / NINDIRECT; // 두 번째 인덱스 계산  uint index3 = bn % NINDIRECT; // 세 번째 인덱스 계산    if((addr = a[index1]) == 0){ // 첫 번째 블록이 할당되지 않았으면  a[index1] = addr = balloc(ip->dev); // 할당  log\_write(bp); // 로그에 기록  }  bp2 = bread(ip->dev, addr); // 첫 번째 블록 읽어옴  b = (uint\*)bp2->data; // 데이터를 배열로 취급    if((addr = b[index2]) == 0){ // 두 번째 블록이 할당되지 않았으면  b[index2] = addr = balloc(ip->dev); // 할당  log\_write(bp2); // 로그에 기록  }  bp3 = bread(ip->dev, addr); // 두 번째 블록 읽어옴  c = (uint\*)bp3->data; // 데이터를 배열로 취급    if((addr = c[index3]) == 0){ // 세 번째 블록이 할당되지 않았으면  c[index3] = addr = balloc(ip->dev); // 할당  log\_write(bp3); // 로그에 기록  }    brelse(bp3); // 버퍼 해제  brelse(bp2); // 버퍼 해제  brelse(bp); // 버퍼 해제  return addr; // 블록 주소 반환  }  panic("bmap: out of range"); // 범위를 초과한 경우 오류 발생  }  static void  itrunc(struct inode \*ip)  {  int i, j, k;  struct buf \*bp, \*bp2, \*bp3;  uint \*a, \*b, \*c;  // 직접 블록 해제  for(i = 0; i < NDIRECT; i++){ // 직접 블록 (Direct blocks)  if(ip->addrs[i]){ // 블록이 할당되어 있으면  bfree(ip->dev, ip->addrs[i]); // 블록 해제  ip->addrs[i] = 0; // 블록 주소 초기화  }  }  // 단일 간접 블록 해제  for(i = 0; i < 4; i++){ // 간접 블록 테이블 4개 처리  if(ip->addrs[NDIRECT + i]){ // 간접 블록 테이블이 존재하면  bp = bread(ip->dev, ip->addrs[NDIRECT + i]); // 테이블 블록 읽어옴  a = (uint\*)bp->data; // 데이터를 배열로 취급  for(j = 0; j < NINDIRECT; j++){ // 간접 블록 내 모든 블록 순회  if(a[j]) // 블록이 할당되어 있으면  bfree(ip->dev, a[j]); // 블록 해제  }  brelse(bp); // 버퍼 해제  bfree(ip->dev, ip->addrs[NDIRECT + i]); // 간접 블록 테이블 자체 해제  ip->addrs[NDIRECT + i] = 0; // 주소 초기화  }  }  // 이중 간접 블록 해제  for(i = 0; i < 2; i++){ // 이중 간접 블록 테이블 2개 처리  if(ip->addrs[10 + i]){ // 이중 간접 블록 테이블이 존재하면  bp = bread(ip->dev, ip->addrs[10 + i]); // 테이블 블록 읽어옴  a = (uint\*)bp->data; // 데이터를 배열로 취급  for(j = 0; j < NINDIRECT; j++){ // 첫 번째 간접 블록 내 모든 블록 순회  if(a[j]){ // 첫 번째 간접 블록이 존재하면  bp2 = bread(ip->dev, a[j]); // 첫 번째 간접 블록 읽어옴  b = (uint\*)bp2->data; // 데이터를 배열로 취급  for(k = 0; k < NINDIRECT; k++){ // 두 번째 간접 블록 내 모든 블록 순회  if(b[k]) // 블록이 할당되어 있으면  bfree(ip->dev, b[k]); // 블록 해제  }  brelse(bp2); // 첫 번째 간접 블록 버퍼 해제  bfree(ip->dev, a[j]); // 첫 번째 간접 블록 해제  }  }  brelse(bp); // 이중 간접 블록 테이블 버퍼 해제  bfree(ip->dev, ip->addrs[10 + i]); // 이중 간접 블록 테이블 해제  ip->addrs[10 + i] = 0; // 주소 초기화  }  }  // 삼중 간접 블록 해제  if(ip->addrs[12]){ // 삼중 간접 블록 테이블이 존재하면  bp = bread(ip->dev, ip->addrs[12]); // 테이블 블록 읽어옴  a = (uint\*)bp->data; // 데이터를 배열로 취급  for(i = 0; i < NINDIRECT; i++){ // 첫 번째 간접 블록 내 모든 블록 순회  if(a[i]){ // 첫 번째 간접 블록이 존재하면  bp2 = bread(ip->dev, a[i]); // 첫 번째 간접 블록 읽어옴  b = (uint\*)bp2->data; // 데이터를 배열로 취급  for(j = 0; j < NINDIRECT; j++){ // 두 번째 간접 블록 내 모든 블록 순회  if(b[j]){ // 두 번째 간접 블록이 존재하면  bp3 = bread(ip->dev, b[j]); // 두 번째 간접 블록 읽어옴  c = (uint\*)bp3->data; // 데이터를 배열로 취급  for(k = 0; k < NINDIRECT; k++){ // 세 번째 간접 블록 내 모든 블록 순회  if(c[k]) // 블록이 할당되어 있으면  bfree(ip->dev, c[k]); // 블록 해제  }  brelse(bp3); // 세 번째 간접 블록 버퍼 해제  bfree(ip->dev, b[j]); // 두 번째 간접 블록 해제  }  }  brelse(bp2); // 첫 번째 간접 블록 버퍼 해제  bfree(ip->dev, a[i]); // 첫 번째 간접 블록 해제  }  }  brelse(bp); // 삼중 간접 블록 테이블 버퍼 해제  bfree(ip->dev, ip->addrs[12]); // 삼중 간접 블록 테이블 해제  ip->addrs[12] = 0; // 주소 초기화  }  ip->size = 0; // 아이노드 크기 초기화  iupdate(ip); // 아이노드 업데이트  } |

|  |
| --- |
| file.h에서 수정한 내용 |
| // in-memory copy of an inode  struct inode {  uint dev;  uint inum;  int ref;  struct sleeplock lock;  int valid;  short type;  short major;  short minor;  short nlink;  uint size;  uint addrs[13]; // NDIRECT+1에서 13으로 변경  }; |

|  |
| --- |
| fs.h에서 수정하고 추가한 내용 |
| #define NDIRECT 6  #define NINDIRECT (BSIZE / sizeof(uint))  #define NDINDIRECT (NINDIRECT \* NINDIRECT)  #define NTINDIRECT (NINDIRECT \* NINDIRECT \* NINDIRECT)  #define MAXFILE (NDIRECT + NINDIRECT \* 4 + NDINDIRECT \* 2 + NTINDIRECT)  struct dinode {  short type; // File type  short major; // Major device number (T\_DEV only)  short minor; // Minor device number (T\_DEV only)  short nlink; // Number of links to inode in file system  uint size; // Size of file (bytes)  uint addrs[13]; // Data block addresses  }; |

+ 3-1 테스트를 위한 임시 기능 및 사전 작업

|  |
| --- |
| defs.h에 추가한 내용 |
| int argfd(int n, int \*pfd, struct file \*\*pf); |

|  |
| --- |
| sysfile.c에서 수정한 부분 |
| int // static 제거  argfd(int n, int \*pfd, struct file \*\*pf)  {  **. . .** 기존 코드와 동일  } |

|  |
| --- |
| usys.S |
| **. . .** 기존 코드와 동일  SYSCALL(rb\_count)  SYSCALL(rb\_print)  SYSCALL(lseek) |

|  |
| --- |
| user.h |
| **. . .** 기존 코드와 동일  int rb\_count(int fd);  int rb\_print(int fd);  int lseek(int fd, int offset, int whence); |

|  |
| --- |
| syscall.h |
| **. . .** 기존 코드와 동일  #define SYS\_rb\_count 22  #define SYS\_rb\_print 23  #define SYS\_lseek 24 |

|  |
| --- |
| syscall.c |
| **. . .** 기존 코드와 동일  extern int sys\_rb\_count(void);  extern int sys\_rb\_print(void);  extern int sys\_lseek(void);  **. . .**  static int (\*syscalls[])(void) = {  **. . .** 기존 코드와 동일  [SYS\_rb\_count] sys\_rb\_count,  [SYS\_rb\_print] sys\_rb\_print,  [SYS\_lseek] sys\_lseek,  }; |

|  |
| --- |
| sysproc.c에 우선 임시로 시스템 콜 추가, 특별한 기능을 하지는 않음 |
| #include "types.h"  #include "x86.h"  #include "defs.h"  #include "date.h"  #include "param.h"  #include "memlayout.h"  #include "mmu.h"  #include "proc.h"  #include "spinlock.h" // sleeplock을 위해 필요  #include "sleeplock.h" // sleeplock을 위해 필요  #include "fs.h"  #include "file.h"  #include "fcntl.h" // O\_RDONLY 등의 플래그를 위해  **. . .** 기존 코드와 동일  int sys\_rb\_count(void)  {  struct file \*f;  if(argfd(0, 0, &f) < 0)  return -1;  return 0; // 임시 반환값  }  int sys\_rb\_print(void)  {  struct file \*f;  if(argfd(0, 0, &f) < 0)  return -1;  return 0; // 임시 반환값  }  int sys\_lseek(void)  {  struct file \*f;  int n, whence;    if(argfd(0, 0, &f) < 0 || argint(1, &n) < 0 || argint(2, &whence) < 0)  return -1;  return 0; // 임시 반환값  } |

✓ 원활한 과제 진행을 위해 ssufs\_test 프로그램 제공

- 실행하기 위해 Makefile 수정 및 3-1에 맞게 수정하여 테스트 진행

|  |
| --- |
| Makefile |
| UPROGS=\  . . . 기존 코드와 동일  \_ssufs\_test\  EXTRA=\  . . . 기존 코드와 동일  ssufs\_test.c\ |

|  |
| --- |
| ssufs\_test.c |
| **. . .** 기존 코드와 동일  int main(int argc, char \*\*argv)  {  for (int i = 0 ; i < BSIZE; i++) {  buf[i] = BSIZE % 10;  }  // test(1, 5, MODE\_COUNT | MODE\_PRINT);  // test(2, 500, MODE\_COUNT | MODE\_PRINT);  // test(3, 5000, MODE\_COUNT | MODE\_PRINT);  // test(4, 50000, MODE\_COUNT | MODE\_PRINT);    test(1, 5, MODE\_DEFALUT);  test(2, 500, MODE\_DEFALUT);  test(3, 5000, MODE\_DEFALUT);  test(4, 50000, MODE\_DEFALUT);  exit();  } |

- 3-1의 테스트를 위해 위와 같이 ssufs\_test.c 코드 변경

- (기능 2) 대용량 파일 생성을 위해 param.h의 FSSIZE를 변경해야함 (1000 -> 2500000)

|  |
| --- |
| param.h |
| #define FSSIZE 2500000 // size of file system in blocks |

✓ 대용량 multi-level 파일시스템을 구현한 수, ssufs\_teset의 실행 결과

- 출력 부분이 과제 예시와 조금 다른데, 교수님께서 제공해주신 ssufs\_test.c 를 열어보면 printf(1, "close file descriptor...\n"); 가 들어있어서 open and read file… 이후 close file descriptor…가 출력, 그렇기에 구현상의 문제가 아니라, ssufs\_test.c가 잘못되었거나 명세서의 출력결과에 close file descriptor...가 빠진 것으로 생각

- 실행 결과는 사진으로 첨부

|  |
| --- |
| 3-1. ssufs\_test 실행 결과 (mode: MODE\_DEFALUT) |
|  |

**- 3-2. 레드블랙트리로 이루어진 페이지 캐시 구현**

- (정의) 레드블랙트리를 이용해 최근 참조 블록이 디스크 접근을 최소화하는 페이지 캐시 구현

✓ file.h 내부의 struct inode에 레드블랙트리에 해당하는 포인터 선언

|  |
| --- |
| file.h에서 수정한 부분 |
| #*define* RB\_RED 0  #*define* RB\_BLACK 1  *struct* rb\_node {  *uint* key;// *블록 번호*  *uint* value;// *블록 주소*  *uint* color;// *노드 색상 (RED 또는 BLACK)*  *struct* rb\_node *\**parent;  *struct* rb\_node *\**left;  *struct* rb\_node *\**right;  };  *struct* rb\_tree {  *struct* rb\_node *\**root;  *int* node\_count;// *현재 노드 개수*  *void* *\**mem\_pool;// *4KB 메모리 풀의 시작 주소*  };  *extern* *struct* rb\_tree rbtree;  // *in-memory copy of an inode*  *struct* inode {  *uint* dev;  *uint* inum;  *int* ref;  *struct* sleeplock lock;  *int* valid;  *short* type;  *short* major;  *short* minor;  *short* nlink;  *uint* size;  *uint* addrs[13];// *NDIRECT+1에서 13으로 변경*  *struct* rb\_tree *\**rbtree;  }; |

🡪 3-2는 구현하지 않았다. 빨간 글씨로 표시했다.

- (기능 1) 레드블랙트리는 최대 100개의 노드를 가짐

- (기능 2) 하나의 레드블랙트리가 사용할 수 있는 메모리는 최대 4KB

✓ 한번의 kalloc으로 레드블랙트리와 모든 노드가 구현 되어야함

✓ 레드블랙트리의 각 노드들을 할당받는 코드는 직접 하나의 페이지를 분리하는 코드를 작성하여 구현

- (기능 3) 레드블랙트리 구조 페이지 캐시의 cache hit 처리

✓ 찾고자 하는 블록 번호가 레드블랙트리에 이미 존재한다면 cache hit이며 바로 레드블랙트리에서 그에 해당

하는 주소를 가져옴

✓ 기존에 존재했던 키를 찾은 경우 해당 키가 가장 최근에 접근한 키가 됨

- (기능 4) 레드블랙트리 구조 페이지 캐시의 cache miss 처리

✓ 찾고자 하는 블록 번호가 레드블랙트리에 없다면 새로운 (블록 번호, 주소) 쌍을 기존의 bmap에서 구하는

방식과 동일하게 계산한 후 레드블랙트리에 추가

✓ 레드블랙트리의 최대 크기를 넘는 경우, 레드블랙트리에서 가장 오래된 노드 삭제 후 (블록 번호, 주소)쌍 추가

✓ 결과적으로 bmap으로 들어온 (블록 번호, 주소)쌍이 cache hit, cache miss 경우 모두 가장 최근에 접근한 키가 됨

**- 3-3. int lseek(int fd, int offset, int whence); 구현**

- (정의) 테스트 프로그램의 파일 오프셋 임의 접근을 위한 lseek 시스템콜 구현

✓ sys\_lseek 코드는 제공, 나머지 코드 및 매크로는 직접 추가 🡪 (3-1)에서 미리 추가했었다.

|  |
| --- |
| sysproc.c |
| *int*  sys\_lseek(*void*)  {  *struct* file *\**f;  *int* n, whence;  *uint* off;    *if*(argfd(0, 0, *&*f) < 0 || argint(1, *&*n) || argint(2, *&*whence))  *return* *-*1;  *switch*(whence) {  *case* SEEK\_SET:  off = 0;  *break*;  *case* SEEK\_CUR:  off = f->off;  *break*;  *case* SEEK\_END:  off = f->ip->size;  *break*;  *default*:  return -1;  break;  }    off += n;  // 범위 체크  if(off < 0 || off > f->ip->size)  return -1;  f->off = off;  return off;  } |

**- 3-4. int rb\_print(int fd); 구현**

- (정의) 현재 페이지 캐시의 레드블랙트리 노드 상태를 in-order 방식으로 출력하는 시스템콜 구현

- (기능 1) 첫째 줄부터 노드 개수만큼 in-order 방식으로 레드블랙트리를 탐색하여 key, val, depth, color, parent key 출력

✓ root node의 depth는 1, parent key는 -1

✓ rb\_print system call 이후 메시지 참고

|  |
| --- |
| sysproc.c |
| *int*  sys\_rb\_print(*void*)  {  *struct* file *\**f;  *if*(argfd(0, 0, *&*f) < 0)  *return* *-*1;  *if*(rbtree.root)  print\_rb\_tree\_inorder(rbtree.root, 1, *-*1); // fs.c에 pre\_order로 구현, 초기 명세 이름이므로 이름은 무시    *return* 0;  } |

-print\_rb\_tree\_inorder, 이름은 inorder이지만, preorder로 동작하는 함수임

|  |
| --- |
| fs.c |
| *void* print\_rb\_tree\_inorder(*struct* rb\_node *\*node*, *int* *depth*, *int* *parent\_key*)  {  *if*(node == 0)  *return*;  // *현재 노드 출력*  cprintf("*key : %d, value : %d, depth : %d, color : %c, parent key : %d\n*",  node->key,  node->value,  depth,  node->color == RB\_RED ? '*R*' : '*B*',  parent\_key);  // *왼쪽 서브트리 출력*  print\_rb\_tree\_inorder(node->left, depth *+* 1, node->key);  // *오른쪽 서브트리 출력*  print\_rb\_tree\_inorder(node->right, depth *+* 1, node->key);  } |

**- 3-5. int rb\_count(int fd); 구현**

- (정의) 현재 페이지 캐시의 정보를 출력하는 시스템콜 구현

- (기능 1) bmap count, cache hit count, disk acess count 출력

✓ bmap 함수의 호출 횟수를 나타내는 bmap count, cache hit 횟수를 나타내는 cache hit count, disk 접근 횟수를 나타내는 disk access count 출력

|  |
| --- |
| sysproc.c |
| *int*  *sys\_rb\_count(void)*  *{*  *struct file \*f;*  *// argfd를 통해 파일 디스크립터를 파일 구조체 포인터로 변환*  *if(argfd(0, 0, &f) < 0)*  *return -1;*    *// 파일의 오프셋이 0인 경우(파일의 시작점) 모든 카운터를 0으로 초기화*  *if(f->off == 0) { // 파일 위치가 시작점일 때는 초기화*  *counters.bmap\_count = 0; // bmap 함수 호출 횟수 초기화*  *counters.cache\_hit = 0; // 캐시 히트 횟수 초기화*  *counters.disk\_access = 0; // 디스크 접근 횟수 초기화*  *}*    *// 각각의 카운터 값을 저장할 지역 변수 선언*  *int bmap\_cnt, cache\_cnt, disk\_cnt;*  *// get\_counters 함수를 호출하여 현재 카운터 값들을 가져옴*  *get\_counters(&bmap\_cnt, &cache\_cnt, &disk\_cnt);*    *// 카운터 값들을 화면에 출력*  *// bmap 함수 호출 횟수, 캐시 히트 횟수, 디스크 접근 횟수를 순서대로 출력*  *cprintf("bmap access count : %d, cache hit count : %d, disk access count : %d\n",*  *bmap\_cnt, cache\_cnt, disk\_cnt);*    *return 0; // 성공적으로 수행되었음을 나타내는 0 반환*  *}* |

[최종 출력]

- 이것은 최종 출력 결과이다. 현재 실질적으로 사용되는 코드는 3-1과 일부 구현한 3-3, 4, 5의 코드로 동작하기 때문에 출력은 과제 명세와 다르다.

|  |
| --- |
| ssufs\_test 실행 결과 (test1: MODE\_PRINT | MODE\_COUNT, test2: MODE\_PRINT, test3: MODE\_COUNT, test4: 생략) |
| $ ssufs\_test  ### test1 start  create and write 5 blocks...  ok  close file descriptor...  ok  open and read file...  rb\_count system call start...  bmap access count : 0, cache hit count : 0, disk access count : 0  ok  rb\_print system call start...  ok  rb\_count system call start...  bmap access count : 5, cache hit count : 0, disk access count : 5  ok  rb\_print system call start...  ok  close file descriptor...  ok  open and read file...  rb\_count system call start...  bmap access count : 0, cache hit count : 0, disk access count : 0  ok  rb\_print system call start...  ok  rb\_count system call start...  bmap access count : 5, cache hit count : 0, disk access count : 5  ok  rb\_print system call start...  ok  close file descriptor...  ok  open and read file...  rb\_count system call start...  bmap access count : 0, cache hit count : 0, disk access count : 0  ok  rb\_print system call start...  ok  rb\_count system call start...  bmap access count : 5, cache hit count : 0, disk access count : 5  ok  rb\_print system call start...  ok  close file descriptor...  ok  unlink file1...  ok  open file1 again...  failed  ### test1 passed...  ### test2 start  create and write 500 blocks...  ok  close file descriptor...  ok  open and read file...  rb\_print system call start...  ok  rb\_print system call start...  ok  close file descriptor...  ok  open and read file...  rb\_print system call start...  ok  rb\_print system call start...  ok  close file descriptor...  ok  open and read file...  rb\_print system call start...  ok  rb\_print system call start...  ok  close file descriptor...  ok  unlink file2...  ok  open file2 again...  failed  ### test2 passed...  ### test3 start  create and write 5000 blocks...  ok  close file descriptor...  ok  open and read file...  rb\_count system call start...  bmap access count : 0, cache hit count : 0, disk access count : 0  ok  rb\_count system call start...  bmap access count : 5000, cache hit count : 0, disk access count : 5000  ok  close file descriptor...  ok  open and read file...  rb\_count system call start...  bmap access count : 0, cache hit count : 0, disk access count : 0  ok  rb\_count system call start...  bmap access count : 5000, cache hit count : 0, disk access count : 5000  ok  close file descriptor...  ok  open and read file...  rb\_count system call start...  bmap access count : 0, cache hit count : 0, disk access count : 0  ok  rb\_count system call start...  bmap access count : 5000, cache hit count : 0, disk access count : 5000  ok  close file descriptor...  ok  unlink file3...  ok  open file3 again...  failed  ### test3 passed... |