1. 개요

본 과제는 xv6 파일시스템에 스냅샷과 롤백을 추가하여, 데이터를 COW로 보호/복구하도록 구현한 것이다. 핵심 로직은 스냅샷 생성 시 데이터 블록은 복제하지 않고 포인터를 공유하고, 이후 쓰기 시 참조수가 1 초과인 블록만 새 블록으로 분기시키는 방식이다. 이를 위해 /snapshot/.refmap으로 블록 참조수를 관리하고, /snapshot/<id>는 읽기 전용 트리, 디렉터리 데이터 블록은 COW 예외, T_DEV와 /snapshot 자체는 캡처 제외로 설계하였다. 롤백은 지정 스냅샷의 트리를 새 inode로 재구성하되 블록 내용은 스냅샷 시점을 가리키게 하고, 삭제는 해당 스냅샷이 마지막 참조자일 때에만 실제 해제되도록 bfree 경로에서 게이트를 둔다. 동작 검증은 print_addr로 직/간접 블록 주소를 고정 포맷으로 관찰하고, mk_test_file-append-snap_test로 생성->수정(COW)->롤백->삭제(해제 타이밍) 을 순차 확인한다. 이 구현을 통해 블록 매핑과 간접 블록, 참조수 기반 COW의 결합 구조를 실제 쓰기 경로에서 체득할 수 있고, 저널링 범위와 락 순서가 메타데이터 일관성에 주는 영향을 확인할 수 있다. 또한 포인터 공유로 얻는 공간 효율과 쓰기 시 COW 분기에 따른 비용의 트레이드오프를 정량적으로 파악하고, 스냅샷 보존/롤백/삭제 각 단계에서 참조수 변화가 실제 해제로 이어지는 정확한 시점을 분명히 구분할 수 있다.

2. 상세설계

2-1. A : block COW

2-1-1. 개요

스냅샷을 생성할 때에는 데이터 블록을 복제하지 않고 기존 블록의 포인터만 공유하며, 해당 시점에 참조수를 /snapshot/.refmap으로 기록한다. 이후 파일에 쓰기 요청이 들어오면 writei 경로에서 대상 논리 블록이 가리키는 실제 블록의 참조수를 확인하고, 공유 상태(ref>1)일 때에만 새 블록을 할당한 뒤 원본 내용을 복사하여 inode 매핑을 새로운 블록으로 교체한다. 이렇게 하면 스냅샷이 붙잡고 있는 과거 데이터는 불변으로 보존되고, 현재 파일만 변경된 블록을 사용하게 된다. 디렉터리의 데이터 블록은 명세에 따라 COW 대상에서 제외되며, /snapshot 하위는 사용자 공간에서의 쓰기/생성/변경을 차단하여 읽기 전용 트리로 유지된다. 블록 해제는 unlink/itrunc 등에서 즉시 진행하지 않고 bfree에서 참조수를 감소시키는 것으로만 처리하며, 참조수가 0이 되는 시점에 한해 비트맵에서 실제로 해제되도록 게이트를 둔다. 이 흐름을 통해 스냅샷 생성은 포인터 공유만으로 빠르고 공간 효율적으로 끝나고, 수정 시에는 필요한 블록만 지연 복사되며, 삭제 시에는 마지막 참조가 사라질 때에만 안전하게 자원이 회수된다.

2-1-2. snapshot.c

2-1-2-1. rc_idx(uint b, uint *idx_out)

```
static inline int rc_idx(uint b, uint *idx_out){
  if(b < rc_data_start) return 0; // not a data block
  uint idx = b - rc_data_start;   // data block index
  if(idx >= rc_nblks) return 0;   // out of range
  *idx_out = idx;
  return 1;
}
```

절대 블록 번호 b를 참조수 테이블 인덱스로 사상하는 내부 유틸리티이다. 데이터 블록 시작점(rc_data_start)보다 작거나 데이터 영역 크기(rc_nblks)를 넘어서는 블록은 추적 대상이 아니므로 0을 반환하고, 유효한 데이터 블록이면 *idx_out에 (b - rc_data_start)를 기록하고 1을 반환한다. 이 함수로 메타데이터/로그/비트맵 영역을 참조수 관리에서 배제함으로써, COW/해제가 순수 데이터 블록에만 적용되도록 보장한다.

2-1-2-2. ensure_snapshot_layout(void)

```
static void ensure_snapshot_layout(void){
  struct inode *root = namei("/");
  if(root == 0) panic("ensure_snapshot_layout: no root");

  // /snapshot
  ilock(root);
  struct inode *snap = dirlookup(root, SNAPDIR, 0); // look for /snapshot
  if(snap == 0){
    snap = ialloc(root->dev, T_DIR);
    if(!snap) panic("mk SNAPDIR: ialloc");
    ilock(snap);
    snap->nlink = 1; iupdate(snap); // link count for "."
    // create . and .. entries
    if(dirlink(snap, ".",  snap->inum) < 0) panic("mk SNAPDIR: .");
    if(dirlink(snap, "..", root->inum) < 0) panic("mk SNAPDIR: ..");
    if(dirlink(root, SNAPDIR, snap->inum) < 0) panic("mk SNAPDIR: link");
    root->nlink++; iupdate(root);
    iunlock(snap);
  }
  iunlock(root);

  // /snapshot/.refmap
  ilock(snap);
  struct inode *ip = dirlookup(snap, RC_FNAME, 0);
  if(ip == 0){
    ip = ialloc(snap->dev, T_FILE);
    if(!ip) panic("mk .refmap: ialloc");
    ilock(ip);
    ip->nlink = 1; iupdate(ip); // link count
    // link into /snapshot
    if(dirlink(snap, RC_FNAME, ip->inum) < 0) panic("mk .refmap: link");
    iunlock(ip);
  }
  iunlock(snap);

  rc_ip = ip; // hold rc_ip reference
  iput(snap);
  iput(root);
}
```

/snapshot 디렉터리와 그 안의 .refmap 파일을 지연 생성(lazy) 하는 루틴이다. 루트에서 snapshot 엔트리를 조회해 없으면

T_DIR로 생성하고, ./.. 링크와 부모 링크를 연결하여 디렉터리 정합성을 맞춘다. 이어 /snapshot/.refmap을 T_FILE로 확보하고 링크를 추가하며, 그 inode 포인터를 전역 rc_ip에 보관한다. 이로써 스냅샷 메타데이터 파일의 존재를 보장하고, 이후 메모리 참조수 테이블의 디스크 영속화가 가능해진다.

2-1-2-3. static void save_refmap(void)

```
static void save_refmap(void){
  if(!rc_ip || !rc_tab) panic("save_refmap: uninit");
  if(rc_saving) return;
  rc_saving = 1;

  begin_op();
  ilock(rc_ip);

  uint need = rc_nblks * sizeof(ushort);
  if(rc_ip->size != need){
    uint off = 0;
    char z[BSIZE];
    memset(z, 0, BSIZE);
    while(off < need){
      uint m = (need - off > BSIZE) ? BSIZE : (need - off);
      if(writei(rc_ip, z, off, m) != m) panic("save_refmap: grow");
      off += m;
    }
  }
  if(writei(rc_ip, (char*)rc_tab, 0, need) != need)
    panic("save_refmap: write");

  iunlock(rc_ip);
  end_op();
  rc_saving = 0;
}
```

현재 메모리 상의 참조수 테이블(rc_tab)을 /snapshot/.refmap inode(rc_ip)에 동기화하는 루틴이다. 재진입 방지를 위한 rc_saving 가드를 두고, 트랜잭션 경계(begin_op/end_op) 안에서 파일 크기를 필요 크기(rc_nblks*sizeof(ushort))만큼 확보한 뒤, 테이블 전체를 오프셋 0부터 writei로 기록한다. 실패 가능 구간에서 패닉 처리로 정합성을 보장한다. 이 함수는 크기 미스매치 시 초기 확장, 정기 플러시, 종료 전 동기화에 사용되어 참조수의 내구성을 확보한다.

2-1-2-4. static void load_refmap(void)

```
static void load_refmap(void){
  uint need = rc_nblks * sizeof(ushort);
  if(need > PGSIZE) panic("load_refmap: table too big");

  rc_tab = (ushort*)kalloc();
  if(!rc_tab) panic("load_refmap: kalloc");
  memset(rc_tab, 0, PGSIZE);

  ilock(rc_ip);
  uint have = rc_ip->size;
  if(have >= need){
    if(readi(rc_ip, (char*)rc_tab, 0, need) != need)
      panic("load_refmap: read");
    iunlock(rc_ip);
  }else{
    iunlock(rc_ip);
    save_refmap();
  }
}
```

디스크의 .refmap을 메모리로 적재한다. 테이블 크기가 PGSIZE 한계를 넘지 않는지 확인한 뒤 kalloc으로 페이지를 확보하고 0으로 초기화한다. .refmap의 현재 크기가 필요 크기 이상이면 readi로 전량을 읽어 들이고, 부족하면 save_refmap()을 호출해 제로 초기화된 테이블을 디스크에 먼저 만들어 정합성을 맞춘다. 이 함수는 초기 마운트 시점의 단 한 번 호출되어 메모리/디스크 상태를 동기화한다.

2-1-2-5. void rc_init(void)

```
void rc_init(void){
  initlock(&rc_lock, "rc");
  // set parameters from superblock
  rc_nblks       = sb.nblocks;
  rc_data_start = sb.bmapstart + ((sb.size + BPB - 1)/BPB);

  // allocate in-memory table
  uint need = rc_nblks * sizeof(ushort);
  if(need > PGSIZE) panic("rc_init: table too big");

  // allocate zeroed table
  rc_tab = (ushort*)kalloc();
  if(!rc_tab) panic("rc_init: kalloc");
  memset(rc_tab, 0, PGSIZE);

  // mark uninitialized
  rc_ip = 0;
  rc_ready = 0;
  rc_saving = 0;

  cprintf("rc_init(minimal): data_start=%u data_blocks=%u bytes=%u\n",
          rc_data_start, rc_nblks, rc_nblks * (int)sizeof(ushort));
}
```

부팅 시점의 경량 초기화 루틴이다. 스핀락을 초기화하고, 슈퍼블록을 기준으로 데이터 영역의 첫 블록(rc_data_start = sb.bmapstart + ceil(sb.size/BPB))과 데이터 블록 수(rc_nblks = sb.nblocks)를 산출한다. PGSIZE 내에 들어가는 크기의 메모리 테이블을 kalloc으로 할당해 0으로 초기화하고, 디스크 측 핸들(rc_ip)은 아직 미지정 상태로 둔다. 이 시점에는 디스크 I/O를 전혀 수행하지 않으며, 이후 필요 시 rc_ensure_mounted()가 레이아웃과 디스크 테이블을 준비한다.

2-1-2-6. void rc_ensure_mounted(void)

```
void rc_ensure_mounted(void){
  if(rc_ready) return;

  ensure_snapshot_layout();  // /snapshot, .refmap

  uint need = rc_nblks * sizeof(ushort);

  if(rc_tab == 0){
    // load from disk
    load_refmap();
  } else {
    ilock(rc_ip);
    uint have = rc_ip->size; // existing size on disk
    iunlock(rc_ip);
    if(have < need) save_refmap(); // grow on disk
  }
  rc_ready = 1;
}
```

참조수 서브시스템을 최초 사용 시 활성화하는 진입점이다. 이미 준비(rc_ready)되어 있으면 즉시 복귀한다. 준비되지 않았다면 ensure_snapshot_layout()으로 /snapshot/.refmap 존재를 보장한 뒤, 메모리 테이블이 없다면 load_refmap()을 호출해 디스크 내용을 메모리로 가져오고, 메모리 테이블이 이미 있다면 디스크 파일 크기를 점검해 부족할 경우 save_refmap()으로 확장한다. 마지막에 rc_ready=1로 표기하여 중복 초기화를 차단한다. 이 함수로 부팅 경로의 I/O를 제거하면서도 스냅샷 경로에서 필요한 순간에만 비용을 지불하도록 설계된다.

2-1-2-7. void rc_get(uint b)

```
int rc_get(uint b){
  uint idx; if(!rc_idx(b, &idx)) return 0;
  int v; acquire(&rc_lock); v = rc_tab[idx]; release(&rc_lock); return v;
}
```

절대 블록 b에 대한 현재 참조수를 읽기 전용으로 반환한다. rc_idx로 데이터 블록 여부를 확인하고, 유효하면 스핀락(rc_lock)을 잡은 상태에서 테이블 값을 읽어 반환한다. 데이터 영역이 아니면 0을 반환한다. COW 분기 여부 판단, 디버깅, 로그에 활용된다.

2-1-2-8. void rc_inc(uint b)

```
int rc_inc(uint b){
  // get index
  uint idx; if(!rc_idx(b, &idx)) return 0;
  int v; acquire(&rc_lock);
  // prevent overflow
  if(rc_tab[idx] < 0xFFFF) rc_tab[idx]++;
  // return new value
  v = rc_tab[idx]; release(&rc_lock); return v;
}
```

절대 블록 b의 참조수를 증가한다. 유효한 데이터 블록일 때 스핀락 하에서 0xFFFF 상한까지 1 증가시키고, 증가 후 값을 반환한다. 스냅샷 생성 시 공유 포인터 수 증가, 파일 캡처, 링크 증가 등 참조 추가 이벤트에서 호출되어 COW 결정의 근거가 된다.

2-1-2-9. void rc_dec(uint b)

```
int rc_dec(uint b){
  // get index
  uint idx; if(!rc_idx(b, &idx)) return 0;
  int v; acquire(&rc_lock);
  if(rc_tab[idx] > 0) rc_tab[idx]--;  // prevent underflow
  // return new value
  v = rc_tab[idx]; release(&rc_lock); return v;
}
```

절대 블록 b의 참조수를 감소한다. 유효한 데이터 블록일 때 스핀락 하에서 0 미만으로 내려가지 않도록 1 감소시키고, 감소 후 값을 반환한다. 파일 삭제/트리 정리 등 참조 제거 이벤트에서 호출되며, 실제 블록 해제는 별도의 해제 경로(bfree)에서 rc_get(b)==0일 때에만 수행되도록 설계 흐름이 맞춰진다.

2-1-2-10. void rc_mark_alloc(void)

```
void rc_mark_alloc(uint b){
  // get index
  uint idx; if(!rc_idx(b, &idx)) return;
  acquire(&rc_lock);
  // mark as allocated
  if(rc_tab[idx] == 0) rc_tab[idx] = 1;
  release(&rc_lock);
}
```

새로 할당된 데이터 블록을 참조수 테이블에 등록한다. 유효 블록이고 현재 참조수가 0인 경우 1로 설정하여 "최소 한 곳에서 참조 중"임을 보장한다. 이는 balloc() 직후나 COW로 새 블록을 만든 직후에 호출되어, 신규 블록이 즉시 추적 체계에 편입되도록 한다.

2-1-2-11. void rc_flush(void)

```
void rc_flush(void){
  if(!rc_ready) return;   // not mounted yet → nothing to do
  if(rc_saving) return;   // reentrancy guard
  save_refmap();
}
```

메모리 참조수 테이블을 조건부로 디스크에 동기화한다. 아직 마운트 준비가 안 되었거나(rc_ready==0), 저장 중(rc_saving==1)이면 아무 것도 하지 않고 복귀한다. 그 외에는 save_refmap()을 호출해 .refmap을 갱신한다. 스냅샷 생성/삭제/대량 변경 같은 구간 후에 영속 상태를 수시로 밀어두기 위한 가벼운 훅으로 쓰인다

2-1-3. fs.c

2-1-3-1. 개요

본 파일에서의 변경 목적은 "쓰기 시에만 복제(COW)"와 "마지막 참조에서만 해제"를 커널의 파일 입출력 경로에 일관되게 녹여 넣는 데 있다. 이를 위해 쓰기 경로인 writei()에 공유 여부 판단→새 블록 할당→내용 복사→매핑 교체→원본 참조수 감소의 COW 분기를 추가하였고, 해제 경로인 bfree()에는 참조수 게이트를 두어 참조수가 0이 되는 순간에만 실제 비트맵을 해제하도록 했다. 또한 매핑 교체의 안전성을 위해 cow_set_block()을 별도 함수로 도입하여 직접/간접 블록을 동일한 인터페이스로 교체하도록 캡슐화하였다. 한편 새로 할당되는 모든 블록은 balloc() 말미의 훅에서 초기 참조수 1로 등록되어, 이후 bfree()에서 참조수 기반의 해제 판정이 정확히 동작하도록 했다. 초기화 단계에서는 iinit()에서 rc_init()을 호출해 참조수 테이블의 부트타임 경량 초기화를 마친 뒤, 실제 디스크 상 .refmap은 스냅샷 경로에서 지연 마운트/동기화한다. 이 일련의 흐름으로 A 명세의 핵심 요구인 "스냅샷 불변, 수정 시 COW, 참조 0에서만 해제"가 fs.c 경로에서 체계적으로 보장된다.

2-1-3-2. cow_set_block() 추가

```
static void
cow_set_block(struct inode *ip, uint bn, uint newb)
{
  if(bn < NDIRECT){
    ip->addrs[bn] = newb;
    iupdate(ip);
    return;
  }
  bn -= NDIRECT;

  // indirect block
  if(bn >= NINDIRECT)
    panic("cow_set_block: bn out");

  // read indirect block
  uint a = ip->addrs[NDIRECT];
  if(a == 0)
    panic("cow_set_block: no indirect");

  // read indirect block
  uint a = ip->addrs[NDIRECT];
  if(a == 0)
    panic("cow_set_block: no indirect");

  // update indirect block
  struct buf *bp = bread(ip->dev, a);
  uint *addr = (uint*)bp->data;
  if(addr[bn] == 0)
    panic("cow_set_block: indir slot empty");

  // set new block
  addr[bn] = newb;
  log_write(bp);
  brelse(bp);

  iupdate(ip);
}
```

이 함수는 논리 블록 번호 bn이 직접/간접 어디에 속하든 매핑을 안전하게 교체하는 보조 함수이다. bn<NDIRECT이면 ip->addrs[bn]=newb로 직접 엔트리를 갱신하고 iupdate(ip)로 메타데이터를 반영한다. 간접 영역이면 bn에서 NDIRECT를 뺀 인덱스로 간접 포인터 블록을 읽어 해당 엔트리를 newb로 바꾸고 log_write()로 저널링한다. 간접 포인터 블록 자체는 데이터 블록이 아니므로 참조수 관리 대상이 아니며, 여기서는 메타데이터 정합성과 원자성만 확보하면 된다. 이 함수의 도입으로 writei()는 COW 시 매핑 교체를 한 줄 호출로 위임할 수 있어, 간접 인덱스 계산 실수나 저널링 누락을 구조적으로 방지한다.

2-1-3-3. writei() 수정

```
for(tot=0; tot<n; tot+=m, off+=m, src+=m){
  // COW: if block is shared, do block-level COW
  uint bn = off/BSIZE;
  uint b = bmap(ip, bn);            // may allocate if needed

  // If existing block is shared, do block-level COW.
  if(cow_enabled && b != 0){
    if(rc_get(b) > 1){
      uint nb = balloc(ip->dev);            // allocate new block
      struct buf *ob = bread(ip->dev, b);   // old block
      struct buf *nbp = bread(ip->dev, nb); // new block
      memmove(nbp->data, ob->data, BSIZE);  // copy data
      log_write(nbp);                       // write new block
      brelse(nbp);                          // release new block
      brelse(ob);                           // release old block
      cow_set_block(ip, bn, nb);            // switch mapping
      rc_dec(b);                            // drop our ref to old
      b = nb;
    }
  }
  bp = bread(ip->dev, b); // read the block, no leaked buffer
```

writei()는 블록 단위로 데이터를 기록하는 핵심 경로로, 본 구현은 루프 내부에 COW 분기를 추가하였다. 각 반복에서 bn=off/BSIZE로 논리 블록을 계산하고, b=bmap(ip,bn)으로 현재 물리 블록을 얻는다(필요 시 bmap이 새 블록을 할당). 파일 타입이 T_FILE인 경우에만 COW를 적용하며, 현재 블록의 참조수 rc_get(b)가 1을 초과하면 공유 상태로 판단한다. 이때 balloc()으로 새 데이터 블록 nb를 할당하고, 원본 b의 내용을 nb로 전체 복사한 뒤 cow_set_block(ip,bn,nb) 호출로 inode의 매핑을 nb로 교체한다. 그 다음 원본 b에 대해 rc_dec(b)로 참조수만 감소시키고, 이후의 실제 데이터 쓰기와 저널링은 교체된 nb를 대상으로 진행한다. 이 흐름으로 현재 파일만 새 블록을 사용하게 되어 스냅샷이 가리키는 과거 데이터는 불변으로 보존된다. 코드 상의 cow_enabled=(ip->type==T_FILE) 가드로 디렉터리 데이터 블록은 COW 대상에서 제외되어 명세를 준수한다. 또한 ip->size 갱신 및 iupdate() 호출은 xv6의 기존 규칙을 그대로 따른다.

2-1-3-4. balloc()에 훅 추가

```
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);

        rc_mark_alloc(b + bi); // read cache
```

balloc(uint dev) 말미에 rc_mark_alloc(b + bi); 훅을 추가하여, 새로 할당된 모든 블록을 참조수 1로 즉시 등록한다. 이 정책은 "새 블록은 사용 시작 시점에 최소 한 곳에서 참조 중"이라는 의미를 시스템 전역에서 일관되게 보장하며, 이후 bfree()에서 참조수 감소→0 도달→실제 해제라는 단선의 흐름을 가능하게 한다. 특히 COW 경로에서 writei()가 balloc()을 호출한 직후 별도의 등록을 반복하지 않는 이유가 여기에 있다. 이미 balloc()이 참조수 1을 기록했으므로, 같은 블록에 대해 다시 rc_mark_alloc()을 호출하면 이중 증가가 된다. 따라서 본 구현은 "등록 책임을 balloc()에 통일"하여 중복/누락 오류를 방지한다.

2-1-3-5. bfree()에 훅 추가

```
// Free a disk block.
static void
bfree(int dev, uint b)
{
  int after = rc_dec(b); // read cache
  if(after > 0) return;  // still referenced
  struct buf *bp;
  int bi, m;
```

bfree(int dev, uint b)는 해제 직전에 int after=rc_dec(b);로 참조수를 먼저 감소시키고, 결과 값이 0일 때에만 비트맵을 클리어한다. 참조수가 남아 있으면 아무 것도 해제하지 않고 반환한다. 이 게이트로 인해 동일 블록을 스냅샷과 현재 파일이 공유하는 동안에는 물리 블록이 결코 조기 해제되지 않고, 스냅샷 삭제나 파일 삭제 등으로 마지막 참조가 사라지는 순간에만 실제 해제가 일어난다. 과거에 관찰되던 "freeing free block"과 같은 오류는 참조수 게이트를 통과하지 못해 구조적으로 제거된다. 또한 스냅샷 트리 삭제, 파일 unlink/itrunc 등 다양한 경로가 모두 bfree()로 수렴하므로, 해제 판단이 단일 지점에서 일관되게 이뤄진다.

2-1-3-6. iinit() 수정

```
void
iinit(int dev)
{
  int i = 0;

  initlock(&icache.lock, "icache");
  for(i = 0; i < NINODE; i++) {
    initsleeplock(&icache.inode[i].lock, "inode");
  }

  readsb(dev, &sb);
  rc_init(); // read cache
```

iinit(int dev)는 슈퍼블록을 읽은 직후 rc_init()을 호출하여 부팅 시 경량 초기화를 수행한다. 여기서 참조수 테이블의 크기와 데이터 영역 시작 블록을 계산하고, 커널 메모리에 테이블 페이지를 0으로 준비한다. 중요한 점은 이 단계에서 디스크 I/O를 수반하는 .refmap 동기화는 하지 않는다는 것이다. .refmap의 생성/로드/저장은 스냅샷 API 경로에서 rc_ensure_mounted() → ensure_snapshot_layout() → load_refmap/save_refmap 순으로 지연 수행된다. 따라서 iinit()의 rc_init()은 COW와 해제 경로에서 사용하는 rc_get/inc/dec이 항상 안전하게 접근할 수 있는 메모리 테이블을 보장하는 역할을 하며, 이후의 디스크 영속화는 스냅샷 기능을 처음 사용할 때 비용을 지불하는 구조이다. 이 배치는 부팅 경로를 가볍게 유지하면서도 COW/해제 로직이 초기 단계부터 안정적으로 동작하도록 만든다.

2-1-4. defs.h

2-1-4-1. 개요

이 파일에서는 스냅샷/블록 참조수 서브시스템의 공개 인터페이스를 선언해, 커널 전역에서 COW/해제/스냅샷 초기화 로직이 일관된 API로 연결되도록 하였다. fs.c는 이 인터페이스를 통해 쓰기 시 COW 판정과 해제 게이트를 수행하고, 스냅샷 경로(시스템콜 측)는 동일 인터페이스로 블록 참조수를 조정하며, 초기화 경로는 지연 마운트/동기화를 트리거한다. 결과적으로 모듈 간 결합을 낮추면서도 "쓰기 시 복제, 마지막 참조에서만 해제"라는 정책이 한 곳(인터페이스)으로 수렴되도록 한다.

2-1-4-2. 수정내용

```
// snapshot.c
void  rc_init(void);                // Initialize the read cache
void  rc_ensure_mounted(void);      // Ensure the snapshot filesystem is mounted
int   rc_get(uint blkno);           // Get a block from the read cache
int   rc_inc(uint blkno);           // Increment the reference count for a block in the read
int   rc_dec(uint blkno);           // Decrement the reference count for a block in the read
void  rc_mark_alloc(uint blkno);    // Mark a block as allocated in the read cache
void  rc_flush(void);               // Flush the read cache to disk
```

스냅샷 참조수 테이블의 생성/보장/조회/증감/등록/동기화에 대한 프로토타입을 추가하였다. rc_init은 부팅 직후의 경량 준비, rc_ensure_mounted는 최초 스냅샷 경로에서의 지연 마운트/동기화 보장, rc_get/inc/dec은 데이터 블록 단위의 참조수 조회 및 증감, rc_mark_alloc은 새로 할당된 블록의 초기 참조 등록, rc_flush는 메모리 테이블을 디스크의 .refmap으로 내리는 동기화 지점에 사용된다. 이 선언 집합을 통해 writei는 공유(ref>1) 판정으로 COW를, bfree는 참조수 0에서만 실제 해제를, 스냅샷 생성/삭제 루틴은 참조수 조정을 각각 안정적으로 수행한다.

2-1-5. 실행흐름

COW는 xv6의 일반 파일 쓰기 경로 안에 자연스럽게 녹아 작동하도록 설계되어 있다. 사용자 공간에서 write()가 호출되면 커널은 파일 디스크립터를 통해 대상 inode를 얻어 잠근 뒤, writei()로 블록 단위 쓰기를 수행한다. 이때 각 반복에서 논리 블록 번호를 계산하고 bmap()으로 현재 물리 블록을 구하는데, 이 시점이 COW 여부를 판정하는 지점이다. COW는 파일 타입(T_FILE)에 대해서만 고려하며 디렉터리 데이터 블록은 명세에 따라 제외한다.

현재 물리 블록의 참조수는 참조수 테이블에서 즉시 조회된다. 참조수가 1이면 해당 블록을 단독으로 보유 중이므로 곧바로 그 블록에 쓰기를 진행한다. 반대로 참조수가 2 이상이면 스냅샷 등 다른 참조자가 존재하는 공유 상태이므로, balloc()으로 새 블록을 할당하고 원본 블록의 전체 내용을 새 블록으로 복사한다. 그 다음 cow_set_block()을 호출하여 inode의 매핑을 새 블록으로 교체하고, 원본 블록은 참조수만 감소시켜 스냅샷이 보존하는 과거 내용을 훼손하지 않는다. 교체가 끝나면 실제 사용자 데이터는 새 블록에 반영되고 저널에 기록된다. 새로 할당된 블록은 할당 시점에 참조수 1로 등록되도록 balloc() 말미의 훅에서 처리되어, 이후 해제 경로가 참조수 기반으로 일관되게 동작한다.

파일 삭제나 트리 정리에서는 반대 흐름이 작동한다. unlink()/itrunc() 등에서 데이터 블록을 회수하려 할 때 bfree()가 먼저 참조수를 감소시키고, 감소 결과가 0인 경우에만 비트맵을 실제로 해제한다. 공유 중인 블록은 마지막 참조자가 사라질 때까지 물리 해제가 지연되므로, 스냅샷과 현재 파일이 동시에 같은 블록을 가리키더라도 조기 해제로 인한 불일치가 발생하지 않는다. 결과적으로 "쓰기 시에는 복제, 해제는 마지막 참조 시"라는 원칙이 파일 쓰기/해제의 핵심 경로에서 자동으로 유지된다.

부팅 시에는 iinit()에서 슈퍼블록을 읽은 뒤 rc_init()이 호출되어 참조수 테이블의 크기와 데이터 영역 경계를 계산하고, 커널 메모리에 테이블을 준비한다. 디스크 상의 /snapshot/.refmap 생성/동기화는 스냅샷 기능을 실제로 사용할 때 지연 수행되므로, COW/해제 경로는 부팅 직후부터 가벼운 메모리 테이블만으로도 안전하게 동작한다. 스냅샷 생성(B 단계) 시에는 트리 복제 루틴이 각 파일의 데이터 블록 참조수를 올려 공유 상태를 형성하고, 그 이후 동일 블록에 대한 쓰기에서 본 절의 COW 분기가 자동으로 발동한다. 이렇게 초기화–쓰기–해제–스냅샷이 맞물려, 스냅샷의 불변성과 공간 효율, 그리고 해제의 정확성이 xv6 전체 흐름 속에서 일관되게 보장된다.

2-2. B : snapshot_create

2-2-1. 개요

현재 파일시스템 트리를 /snapshot/ 아래에 읽기 전용으로 캡처하되, 데이터 블록은 복사하지 않고 포인터를 공유하도록 설계한 시스템콜이다. 호출 시 커널은 먼저 rc_ensure_mounted()로 /snapshot 디렉터리와 .refmap의 존재를 보장하고, 새 스냅샷 ID를 발급한 뒤 /snapshot/<id>를 스냅샷 루트로 생성한다. 그 다음 루트(/)를 시작점으로 재귀 복제를 수행하되, /snapshot 자체와 장치파일(T_DEV)은 캡처 대상에서 제외하고, 디렉터리의 데이터 블록은 COW 대상에서 제외한다. 일반 파일(T_FILE)을 만나면 파일 크기/블록 포인터 등 메타데이터를 스냅샷 트리에 복제하고, 해당 파일이 가리키는 직/간접 데이터 블록마다 refcount를 +1 하여

"현재 트리와 스냅샷 트리가 같은 데이터 블록을 공유(ref>1)"하도록 만든다. 이 상태에서 이후 현재 트리에 쓰기가 들어오면, A 단계에서 확장한 writei()가 rc_get(b)>1을 감지하여 블록 단위 COW(새 블록 할당→내용 복사→매핑 교체→원본 ref 감소) 를 자동으로 수행하므로, 스냅샷의 과거 내용은 불변으로 유지된다. 성공 시 시스템콜은 할당한 스냅샷 ID를 반환하고, 실패 시 음수 값을 반환한다. 이러한 동작으로 스냅샷 생성 비용은 주로 메타데이터 복제와 참조수 증가에 국한되며, 데이터 복제 비용은 실제 수정 시점으로 지연된다.

2-2-2. sysproc.c

2-2-2-1. 개요

이 파일에서는 snapshot_create() 구현을 위해 필요한 디렉터리/파일 생성 보조, 트리 복제, 데이터 블록 참조수 상승, 정리 루틴, 그리고 시스템콜 본체를 제공한다. 핵심 아이디어는 루트(/) 트리를 /snapshot/<id>로 형태만 복제하고, 일반 파일이 가리키는 데이터 블록의 참조수(refcount)를 증가시켜 스냅샷과 현재 트리가 동일 블록을 공유하게 만드는 것이다. 이후 현재 트리에서 쓰기가 들어오면 A 단계에서 확장한 writei()가 rc_get(b)>1을 감지하여 블록 단위 COW를 자동 수행하므로, 스냅샷 내용은 불변으로 유지된다. 또한 /snapshot 자체와 장치파일(T_DEV)은 캡처 대상에서 제외하여 명세의 읽기 전용/비캡처 규칙을 준수한다.

2-2-2-2. itoa10(int x, char *buf)

```
static int itoa10(int x, char *buf){
  int i=0, j=0; char tmp[16];
  if(x==0){ buf[0]='0'; buf[1]=0; return 1; }
  while(x>0){ tmp[i++] = '0' + (x%10); x/=10; }
  while(i>0) buf[j++] = tmp[--i];
  buf[j]=0; return j;
}
```

스냅샷 디렉터리명을 숫자 ID로 만들기 위한 간단한 10진수 변환 유틸리티이다. id를 문자열로 변환하여 /snapshot/<id> 생성에 사용한다.

2-2-2-3. streq(const char *a, const char *b)

```
static int streq(const char *a, const char *b){
  while(*a && *b){ if(*a!=*b) return 0; a++; b++; }
  return *a==0 && *b==0;
}
```

디렉터리 엔트리 이름 비교를 위한 문자열 동일성 함수이다. snapshot 등의 필터링 조건을 간단하게 표현하기 위해 사용한다.

2-2-2-4. mkdir_locked(struct inode *dp, char *name)

```
static struct inode*
mkdir_locked(struct inode *dp, char *name)
{
  struct inode *ip;

  // allocate new directory inode
  ip = ialloc(dp->dev, T_DIR);
  if(ip == 0) panic("mkdir_locked: ialloc");
  ilock(ip);
  ip->nlink = 1;        // for "."
  iupdate(ip);

  // "." and ".."
  if(dirlink(ip, ".",  ip->inum) < 0)  panic("mkdir_locked: '.'");
  if(dirlink(ip, "..", dp->inum) < 0)  panic("mkdir_locked: '..'");

  // link into parent
  if(dirlink(dp, name, ip->inum) < 0)  panic("mkdir_locked: link parent");

  // parent has one more subdir
  dp->nlink++;
  iupdate(dp);

  iunlock(ip);
  return ip; // caller still holds a ref via icache
}
```

부모 디렉터리 dp가 이미 ilock()된 상태에서 하위 디렉터리 inode를 새로 할당하고, ./.. 링크 및 부모 링크를 설정한 뒤 부모의 링크 카운트 증가까지 처리한다. 스냅샷 트리 복제 시 /snapshot/<id> 및 하위 디렉터리를 생성하는 데 사용된다.

2-2-2-5. create_file_locked(struct inode *dp, char *name)

```
static struct inode*
create_file_locked(struct inode *dp, char *name)
{
  // allocate new file inode
  struct inode *ip = ialloc(dp->dev, T_FILE);
  if(ip == 0) panic("create_file_locked: ialloc");

  ilock(ip); // lock new inode
  ip->nlink = 1;
  iupdate(ip);
  // link into parent
  if(dirlink(dp, name, ip->inum) < 0) panic("create_file_locked: dirlink");
  iunlock(ip); // unlock new inode
  return ip;
}
```

부모 디렉터리 dp가 ilock()된 상태에서 일반 파일 inode를 새로 할당하고, 디렉터리 엔트리를 연결한다. 스냅샷 트리 복제 시 대상 트리에 파일 껍데기를 만들 때 사용된다.

2-2-2-6. rc_inc_file_blocks(struct inode *f)

```
static void rc_inc_file_blocks(struct inode *f)
{
  // direct
  for(int i=0;i<NDIRECT;i++){
    uint b = f->addrs[i];
    if(b) rc_inc(b);
  }
  // indirect entries (if present)
  uint ib = f->addrs[NDIRECT];
  if(ib){
    struct buf *bp = bread(f->dev, ib);
    uint *addr = (uint*)bp->data;
    for(int k=0;k<NINDIRECT;k++){
      uint b = addr[k];
      if(b) rc_inc(b);
    }
    brelse(bp);
  }
}
```

파일 f가 보유하는 데이터 블록의 참조수를 증가시키는 헬퍼이다. NDIRECT 구간의 직접 포인터를 순회하여 0이 아닌 블록에 rc_inc(b)를 호출하고, 간접 포인터 블록이 있으면 이를 읽어 NINDIRECT 엔트리를 훑어 또 한 번 rc_inc(b)를 호출한다. 이 증가 작업으로 스냅샷과 현재 트리가 같은 데이터 블록을 공유(ref>1) 하게 되어, 이후 현재 트리에서 동일 블록에 쓰기가 들어오면 COW가 자동으로 발동한다. 디렉터리 데이터 블록은 명세상 COW 대상이 아니므로 여기서는 파일(T_FILE)만 다룬다.

2-2-2-7. dir_is_root(struct inode *dp)

```
static int dir_is_root(struct inode *dp){
  return dp->inum == ROOTINO;
}
```

해당 디렉터리가 루트인지 판정한다. 루트에서만 /snapshot 엔트리를 복제 대상에서 제외하기 위한 조건으로 사용한다.

2-2-2-8. clone_tree(struct inode *dst_dir, struct inode *src_dir)

```c
static int
clone_tree(struct inode *dst_dir, struct inode *src_dir)
{
  struct dirent de;
  uint off = 0;
  int n;

  while((n = readi(src_dir, (char*)&de, off, sizeof(de))) == sizeof(de)){
    off += sizeof(de);
    // skip invalid entries
    if(de.inum == 0) continue;
    if(streq(de.name, ".") || streq(de.name, "..")) continue;
    if(dir_is_root(src_dir) && streq(de.name, "snapshot")) continue;

    // lookup child in src_dir
    uint child_off = 0;
    struct inode *ch = dirlookup(src_dir, de.name, &child_off);
    if(ch == 0) continue;
    ilock(ch);

    if(ch->type == T_DIR){
      // create subdir in dst_dir
      struct inode *sub = dirlookup(dst_dir, de.name, 0);
      if(!sub) sub = mkdir_locked(dst_dir, de.name);

      ilock(sub);
      int ok = clone_tree(sub, ch); // recursive clone
      iunlock(sub);
      iput(sub);

      iunlock(ch);
      iput(ch);
      // check recursive result
      if(!ok) return 0;
    } else if(ch->type == T_FILE){
      struct inode *nf = dirlookup(dst_dir, de.name, 0);
      if(!nf) nf = create_file_locked(dst_dir, de.name);

      ilock(nf);

      // copy metadata + block pointers
      nf->size = ch->size;
      for(int i=0;i<NDIRECT;i++) nf->addrs[i] = ch->addrs[i];
      nf->addrs[NDIRECT] = ch->addrs[NDIRECT];
      iupdate(nf);

      // increment refcounts for data blocks
      rc_inc_file_blocks(ch);

      iunlock(nf);
      iput(nf);
      iunlock(ch);
      iput(ch);

    } else if(ch->type == T_DEV){
      // skip device files
      iunlock(ch);
      iput(ch);
      continue;
    } else {
      // unknown type
      iunlock(ch);
      iput(ch);
    }
  }
  if(n < 0) return 0;
  return 1;
}
```

src_dir를 읽어 디렉터리 엔트리 단위로 순회하면서 dst_dir로 재귀 복제하는 핵심 루틴이다.

- 공통 필터: ./..는 건너뛴다. 루트에서 이름이 "snapshot"인 엔트리는 제외한다.
- T_DIR: 대상에 동일 이름의 하위 디렉터리가 없으면 mkdir_locked()로 생성하고, 재귀 호출로 하위 트리를 복제한다.
- T_FILE: 대상에 파일 껍데기가 없으면 create_file_locked()로 생성한다. 그 후 파일 크기와 블록 포인터(직접+간접)를 메타데이터로 복제하고 iupdate()로 반영한다. 이어서 원본 파일 ch의 데이터 블록들에 대해 rc_inc_file_blocks(ch)를 호출하여 포인터 공유(ref++) 를 형성한다.
- T_DEV: 캡처 대상에서 제외한다.

루틴 전반에 걸쳐 자식 inode는 ilock()으로 보호되고, 작업이 끝나면 iunlock/iput으로 정리한다. 반환값은 성공/실패를 나타내며, 상위 호출자에서 스냅샷 생성의 성공 여부를 판정하는 데 사용된다.

2-2-2-9. clear_tree(struct inode *dir, int skip_snapshot)

```c
static int
clear_tree(struct inode *dir, int skip_snapshot)
{
  struct dirent de;
  uint off = 0;
  int n;

  while((n = readi(dir, (char*)&de, off, sizeof(de))) == sizeof(de)){
    off += sizeof(de);
    if(de.inum == 0) continue;
    if(streq(de.name, ".") || streq(de.name, "..")) continue;
    if(skip_snapshot && dir_is_root(dir) && streq(de.name, "snapshot"))
      continue;

    uint off_child = 0;
    struct inode *ip = dirlookup(dir, de.name, &off_child);
    if(ip == 0) continue;
    ilock(ip);

    if(ip->type == T_DIR){
      int ok = clear_tree(ip, 0);
      if(!ok){ iunlockput(ip); return 0; }

      // remove subdir entry from parent
      struct dirent z; memset(&z, 0, sizeof(z));
      if(writei(dir, (char*)&z, off_child, sizeof(z)) != sizeof(z)){
        iunlockput(ip); return 0;
      }
      // fix link counts
      dir->nlink--; iupdate(dir);
      ip->nlink--;  iupdate(ip);
      iunlockput(ip);
    } else if(ip->type == T_FILE){
      // unlink file entry
      struct dirent z; memset(&z, 0, sizeof(z));
      if(writei(dir, (char*)&z, off_child, sizeof(z)) != sizeof(z)){
        iunlockput(ip); return 0;
      }
      ip->nlink--; iupdate(ip);
      iunlockput(ip);

    } else if(ip->type == T_DEV){
      // unlink device entry
      struct dirent z; memset(&z, 0, sizeof(z));
      if(writei(dir, (char*)&z, off_child, sizeof(z)) != sizeof(z)){
        iunlockput(ip); return 0;
      }
      ip->nlink--; iupdate(ip);
      iunlockput(ip);

    } else {
      iunlockput(ip);
    }
  }
  if(n < 0) return 0;
  return 1;
}
```

디렉터리 dir 하위의 모든 엔트리를 깨끗이 제거하는 정리 루틴이다. 보고서 B-1(생성)에서는 직접 사용하지 않지만, 이후 B-2(rollback)/B-3(delete)에서 현재 트리 비우기나 스냅샷 트리 제거에 사용된다. 루트에서 skip_snapshot이 참이면 "snapshot"

엔트리는 보존한다. 각 엔트리 제거는 디렉터리 엔트리 영역을 0으로 덮고 링크 카운트를 조정하는 방식으로 수행하며, 실제 데이터 블록 해제는 파일 시스템 공통 경로(itrunc/bfree)에서 참조수 게이트에 의해 안전하게 처리된다.

2-2-2-10. sys_snapshot_create(void)

```
int
sys_snapshot_create(void)
{
  begin_op();

  rc_ensure_mounted();   // lazy-mount /snapshot and load .refmap

  struct inode *root = namei("/");
  struct inode *snap = namei("/snapshot");
  if(!root || !snap){ end_op(); return -1; }

  // find next available numeric ID under /snapshot
  char name[16];
  int id = 0;
  for(;; id++){
    itoa10(id, name);
    ilock(snap);
    struct inode *exist = dirlookup(snap, name, 0);
    iunlock(snap);
    if(!exist) break;
    iput(exist);
  }

  // create /snapshot/[id]
  ilock(snap);
  struct inode *dst = dirlookup(snap, name, 0);
  if(!dst) dst = mkdir_locked(snap, name);
  iunlock(snap);

  // clone root -> /snapshot/[id]
  ilock(root);
  ilock(dst);
  int ok = clone_tree(dst, root);
  iunlock(dst);
  iunlock(root);

  end_op();

  rc_flush();  // persist refmap once

  if(!ok) return -1;
  return id;
}
```

시스템콜 본체이다.

(i) begin_op()로 트랜잭션을 시작한다.

(ii) rc_ensure_mounted()로 /snapshot 레이아웃 및 .refmap의 존재를 지연 보장한다.

(iii) namei("/"), namei("/snapshot")로 루트와 스냅샷 루트를 얻는다.

(iv) /snapshot 하위에서 존재하지 않는 최소 정수 ID를 선형 탐색으로 찾는다(itoa10(id,name) 사용).

(v) 해당 이름으로 /snapshot/<id>를 만들기 위해 mkdir_locked()를 호출한다(이미 있으면 재사용 보호).

(vi) 루트와 대상 디렉터리를 ilock()한 뒤 clone_tree(dst, root)를 호출해 트리를 복제하고, 파일마다 rc_inc_file_blocks()를 통해 ref++ 한다.

(vii) end_op()로 파일 시스템 트랜잭션을 마친 뒤, rc_flush()로 메모리의 refmap을 한 번 디스크에 영속화한다.

(viii) 실패 시 음수, 성공 시 할당한 스냅샷 ID를 반환한다.

2-2-3. sysfile.c

2-2-3-1. 개요

이 모듈에서는 /snapshot 하위를 항상 읽기 전용 영역으로 고정하기 위해, 시스템콜 진입부에서 경로 기반 차단을 수행하도록 수정하였다. 구체적으로 /snapshot 또는 그 하위 경로에 대해 쓰기/생성/삭제/링크 생성 등 모든 변경 연산을 거부하고, 읽기(O_RDONLY)만 허용한다. 이렇게 커널의 파일 연산 관문에서 봉인해 두면 스냅샷 트리는 불변으로 유지되고, 데이터 수정은 오직 현재 트리에서만 발생하여 A단계의 COW가 정확히 작동한다.

2-2-3-2. path_is_under_snapshot(const char *path)

```
static int path_is_under_snapshot(const char *path){
  if(!path || path[0] != '/') return 0;
  if(strncmp(path, "/snapshot", 9) == 0){
    if(path[9] == 0 || path[9] == '/') return 1;
  }
  return 0;
}
```

절대경로만 대상으로 "/snapshot" 그 자체 혹은 "/snapshot/" 접두인지 확인하여 스냅샷 영역 여부를 판정한다. xv6는 심볼릭 링크가 없어 문자열 접두 판정만으로 충분하다.

2-2-3-3. sys_open()

```
int
sys_open(void)
{
  char *path;
  int fd, omode;
  struct file *f;
  struct inode *ip;

  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  if(path_is_under_snapshot(path)){
    if(omode & O_WRONLY || omode & O_RDWR || omode & O_CREATE){
      cprintf("SNAP R/O DENY op=open path=%s\n", path);
      return -1; // Deny open if path is under /snapshot
    }
  }
```

인자 파싱 후, 대상 경로가 스냅샷 하위이고 열기 모드에 쓰기 계열 플래그(O_WRONLY/O_RDWR)가 하나라도 포함되면 즉시 실패(-1)를 반환한다. xv6에는 O_TRUNC가 없으므로 사용하지 않으며, 읽기 전용(O_RDONLY) 열기는 허용한다. 이로써 스냅샷 파일에 대한 내용 변경/생성을 원천 차단한다.

2-2-3-4. sys_unlink()

```
int
sys_unlink(void)
{
  struct inode *ip, *dp;
  struct dirent de;
  char name[DIRSIZ], *path;
  uint off;

  if(argstr(0, &path) < 0)
    return -1;

  // Deny unlink if path is under /snapshot
  if(path_is_under_snapshot(path)){
    cprintf("SNAP R/O DENY op=mkdir path=%s\n", path);
    return -1;
  }
```

삭제 대상 경로가 스냅샷 하위이면 즉시 실패 처리한다. 디렉터리 엔트리 삭제로 스냅샷 트리를 변형하는 행위를 금지한다.

2-2-3-5. sys_mkdir()

```
int
sys_mkdir(void)
{
  char *path;
  struct inode *ip;

  // invalid path check
  if(argstr(0, &path) < 0) return -1;

  // Deny mkdir if path is under /snapshot
  if(path_is_under_snapshot(path)){
    cprintf("SNAP R/O DENY op=mkdir path=%s\n", path);
    return -1;
  }
```

생성 대상 경로가 스냅샷 하위이면 즉시 실패 처리한다. 스냅샷 트리에 신규 디렉터리 추가를 금지한다.

2-2-3-6. sys_mknod()

```
int
sys_mknod(void)
{
  struct inode *ip;
  char *path;
  int major, minor;

  if(argstr(0, &path) < 0 || argint(1, &major) < 0 || argint(2, &minor) < 0) return -1;

  // Deny mknod if path is under /snapshot
  if(path_is_under_snapshot(path)){
   cprintf("SNAP R/O DENY op=mkdir path=%s\n", path);
    return -1;
  }
```

생성 대상 경로가 스냅샷 하위이면 즉시 실패 처리한다. 스냅샷 영역에 장치 파일을 만들 수 없도록 한다.

2-2-3-7. sys_link()

```
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;

  // Deny link if new path is under /snapshot
  if(path_is_under_snapshot(new)){
    cprintf("SNAP R/O DENY op=mkdir path=%s\n", new);
    return -1;
  }
```

새 경로(new) 가 스냅샷 하위인 경우 하드링크 생성을 거부한다. 스냅샷 디렉터리 구조에 새 엔트리 추가가 발생하지 않도록 막는다.

2-2-4. syscall.h

```
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_snapshot_create 22     // Create a snapshot
```

시스템콜 번호를 새로 정의하여 테이블 인덱스와 일치하도록 한다.

2-2-5. syscall.c

```
extern int sys_uptime(void);
extern int sys_snapshot_create(void);   // snapshot create syscall
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_snapshot_create]   sys_snapshot_create,   // snapshot create syscall
```

프로토타입 선언 추가 및 syscalls[] 매핑 테이블에 항목 등록한다.

2-2-6. usys.S

```
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(snapshot_create)
```

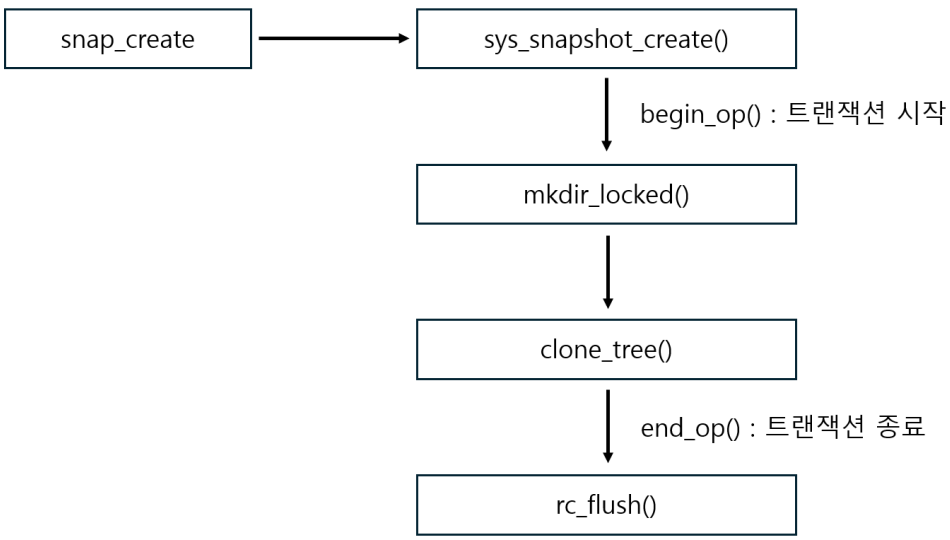
사용자 공간 트랩 스텁 생성 위해 매크로 한 줄 추가한다.

2-2-7. user.h

```
// new syscalls for snapshots
int snapshot_create(void);      // creates a snapshot, returns snapshot id
```

유저 레벨 호출을 위해 함수 원형 선언 추가.

2-2-8. 실행흐름



snapshot_create()는 "/snapshot/<id> 트리 생성 → 루트(/) 트리 재귀 복제(모양만) → 파일 데이터 블록 ref++ → refmap 동기화" 순서로 진행된다. 트랜잭션 경계(begin_op/end_op) 안에서 디렉터리/파일 잠금 규칙을 지키며 수행되고, 스냅샷 하위 R/O 보장은 sysfile.c에서 별도로 강제된다. 세부 단계는 다음과 같다.

(i) 사용자 호출

유저 프로그램이 snapshot_create()를 호출하면 시스템콜 진입 후 커널 경로로 넘어간다.

(ii) 준비(지연 마운트 & 메타 준비)

rc_ensure_mounted()가 /snapshot 디렉터리와 .refmap 파일 존재를 보장한다.

- /snapshot 없으면 생성, .refmap 없으면 생성.
- 필요 시 refcount 테이블을 디스크와 동기화(load/save).

(iii) 트랜잭션 시작 및 루트 획득

begin_op()로 로그 트랜잭션 시작.

namei("/"), namei("/snapshot")로 원본 루트와 스냅샷 루트 inode 획득.

(iv) 스냅샷 ID 결정 및 목적지 디렉터리 생성

/snapshot 하위를 정수 문자열(0,1,2,...)로 선형 검색하여 비어 있는 최소 ID를 찾는다.

mkdir_locked()로 /snapshot/<id> 디렉터리 생성(이미 있지 않도록 보장).

(v) 트리 복제(모양만)

clone_tree(dst=/snapshot/<id>, src=/)를 호출한다. 이때 규칙은 다음과 같다.

- ./..는 스킵, 루트에서 "snapshot" 엔트리는 스킵.
- T_DIR은 재귀적으로 하위 디렉터리를 만들어 진행.
- T_FILE은 파일 껍데기를 만들고 크기/데이터 포인터(직접/간접) 메타를 복제한 뒤, rc_inc_file_blocks(ch)로 데이터 블록 참조수(ref)를 +1 한다.
- T_DEV는 복제 제외.

이로써 스냅샷 트리와 현재 트리가 같은 데이터 블록을 공유(ref>1) 하게 된다.

(vi) 트랜잭션 종료 & refmap 영속화

end_op()로 파일시스템 트랜잭션 종료.

rc_flush()로 메모리 상 refcount 테이블을 .refmap에 1회 기록한다.

(vii) 반환

성공 시 할당된 스냅샷 ID를 반환, 중간 실패 시 음수 반환.

- 불변성 보장(별도 경로) : 스냅샷 하위의 모든 변경 연산(open with write/create, unlink, mkdir, mknod, link 등)은 sysfile.c의 path_is_under_snapshot() 검사로 즉시 거부되어, 생성된 스냅샷 트리는 항상 읽기 전용으로 유지된다.
- A단계 COW와의 접점(참고) : 이후 현재 트리에서 공유 블록에 쓰기가 들어오면 writei()가 rc_get(b)>1을 감지해 블록 단위 COW(신규 할당→내용 복사→매핑 교체→원본 ref--)를 수행하므로, 스냅샷의 과거 내용은 변하지 않는다.

2-3. B : snapshot_rollback

2-3-1. 개요

snapshot_rollback(snap_id)는 현재 파일시스템을 지정 스냅샷 시점으로 되돌리는 복구 연산으로, /snapshot/<id> 트리를 기준으로 루트(/)를 재구성하는 시스템콜이다. 구현의 핵심은 첫째, 현재 루트에서 /snapshot 엔트리만 남기고 모든 엔트리를 정리하여 기존 데이터 블록의 참조수를 감소시킨다. 둘째, 대상 스냅샷 트리를 형상만 복제하되 각 일반 파일의 데이터 블록 포인터를 그대로 공유하도록 만들어 실제 데이터 복사 없이 시점을 복구하는 데 있다. 이때 파일 inode는 새로 할당하여 구성하며, 디렉터리 데이터 블록은 명세에 따라 COW 대상에서 제외한다. 복구 과정에서 스냅샷 파일이 가리키는 직접/간접 데이터 블록의 refcount를 증가시켜 현재 트리와 스냅샷이 포인터 공유(ref>1) 상태가 되도록 하고, 이후 현재 트리에 쓰기가 들어오면 A 단계의 writei() 경로가 rc_get(b)>1을 감지하여 블록 단위 COW(새 블록 할당→원본 보존)를 자동 수행한다. 경로 보호는 기존과 동일하게 /snapshot 하위에서의 모든 변경 연산을 차단하여 스냅샷의 불변성을 유지한다. 전체 연산은 begin_op/end_op 트랜잭션 경계 안에서 진행되며, 정리 단계의 bfree()는 refcount 0에서만 실제 해제를 수행하므로 스냅샷이 참조하는 블록은 안전하게 유지된다. 결과적으로 snapshot_rollback은 메타데이터 중심의 재배열과 참조수 조정만으로 시점을 복구하고, 데이터 블록 복제 비용은 필요 시점의 쓰기 시로 지연된다.

2-3-2. sysproc.c

```c
int
sys_snapshot_rollback(void)
{
  int id;
  if(argint(0, &id) < 0) return -1;

  begin_op();

  rc_ensure_mounted();

  struct inode *snap = namei("/snapshot");
  struct inode *root = namei("/");
  if(!snap || !root){ end_op(); return -1; }

  char name[16]; itoa10(id, name);

  // find snapshot source dir
  ilock(snap);
  struct inode *src = dirlookup(snap, name, 0);
  iunlock(snap);
  if(!src){ end_op(); return -1; }

  // 1) clear current root (keep /snapshot if present)
  ilock(root);
  int ok = clear_tree(root, /*skip_snapshot=*/1);
  iunlock(root);
  if(!ok){ iput(src); end_op(); return -1; }

  // 2) clone snapshot tree into root
  ilock(root);
  ilock(src);
  ok = clone_tree(root, src);
  iunlock(src);
  iunlock(root);

  iput(src);
  end_op();

  rc_flush();
  return ok? 0 : -1;
}
```

rc_ensure_mounted()로 /snapshot/.refmap을 지연 보장한 뒤, namei("/snapshot/<id>")로 대상 스냅샷 존재를 확인하고, begin_op() 경계 안에서 다음을 수행한다

(i) clear_tree(root, /*skip_snapshot=*/1)로 현재 루트에서 "snapshot" 항목만 남기고 모든 엔트리 제거

(ii) clone_tree(dst=root, src=/snapshot/<id>)로 디렉터리/파일 모양을 재귀 복제하고, 파일마다 rc_inc_file_blocks()를 호출해 데이터 블록 참조수(ref)를 증가시켜 포인터 공유(ref>1) 상태를 형성한다.

디렉터리 데이터 블록은 명세대로 COW 대상에서 제외하고, T_DEV는 복제 대상에서 제외한다.

(iii) end_op() 후 rc_flush()를 1회 호출해 refmap을 디스크에 동기화하며, 성공 시 0, 오류 시 음수를 반환한다. 이렇게 함으로써 롤백은 메타데이터 재배열과 참조수 조정만으로 완료되고, 이후 현재 트리에서의 쓰기는 writei() 경로에서 자동 COW로 분기되어 스냅샷 불변성이 유지된다.

2-3-3. syscall.h

```c
#define SYS_close  21
#define SYS_snapshot_create 22     // Create a snapshot
#define SYS_snapshot_rollback 23   // Rollback to a snapshot
```

시스템콜 번호를 새로 정의하여 테이블 인덱스와 일치하도록 한다.

2-3-4. syscall.c

```c
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_snapshot_create(void);   // snapshot create syscall
extern int sys_snapshot_rollback(void); // snapshot rollback syscall
```

```c
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_snapshot_create]   sys_snapshot_create,   // snapshot create syscall
[SYS_snapshot_rollback] sys_snapshot_rollback, // snapshot rollback syscall
```

프로토타입 선언 추가 및 syscalls[] 매핑 테이블에 항목 등록한다.

2-3-5. usys.S

```c
SYSCALL(uptime)
SYSCALL(snapshot_create)
SYSCALL(snapshot_rollback)
```

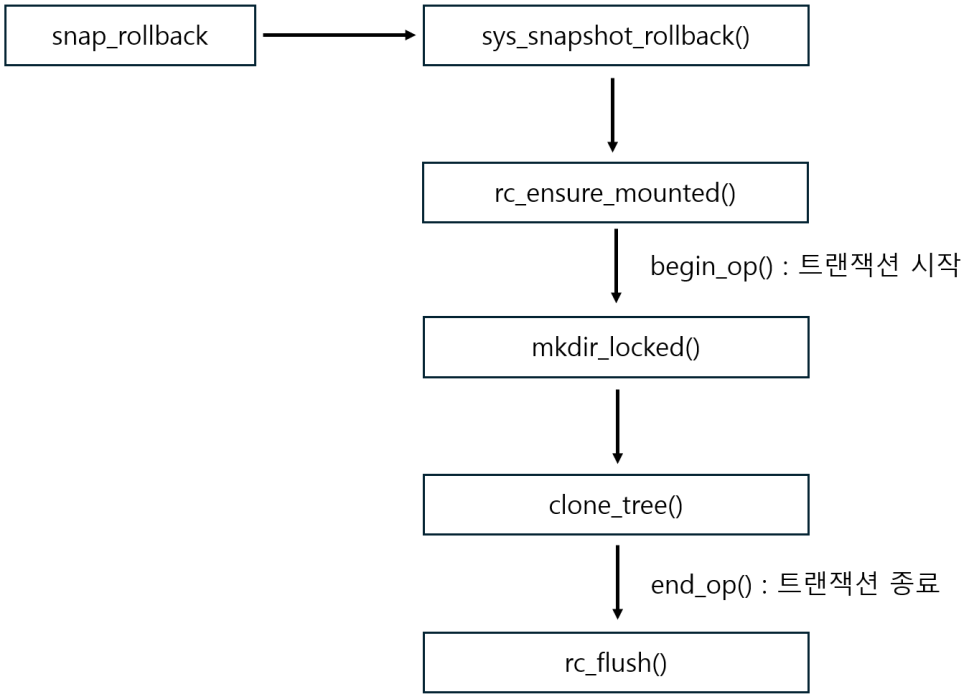
사용자 공간 트랩 스텁 생성 위해 매크로 한 줄 추가한다.

2-3-6. user.h

```c
// new syscalls for snapshots
int snapshot_create(void);      // creates a snapshot, returns snapshot id
int snapshot_rollback(int id);  // rollbacks to the snapshot with given id
```

유저 레벨 호출을 위해 함수 원형 선언 추가.

2-3-7. 실행흐름



(i) 시작 및 지연 준비

사용자 프로그램이 snapshot_rollback(snap_id)를 호출하면 커널은 sys_snapshot_rollback()으로 진입한다. 먼저 rc_ensure_mounted()를 호출하여 /snapshot 디렉터리와 .refmap의 존재를 보장하고, 필요 시 메모리 참조수 테이블과 온디스크 .refmap 사이의 동기화를 수행한다. 이는 이후 refcount 조정과 최종 flush의 전제가 된다.

(ii) 타깃 스냅샷 확인

namei("/snapshot")로 스냅샷 루트를, namei("/snapshot/<id>")로 대상 스냅샷 디렉터리를 얻는다. <id>가 존재하지 않으면 즉시 음수(-1)를 반환한다.

(iii) 트랜잭션 시작

begin_op()로 파일시스템 로그 트랜잭션을 시작한다. 롤백 과정의 디렉터리/파일 메타데이터 변경이 원자적으로 커밋되도록 하기 위함이다.

(iv) 현재 루트 정리 (스냅샷 항목만 보존)

root = namei("/")를 ilock(root) 후 clear_tree(root, /*skip_snapshot=*/1)을 호출한다.

- 이 루틴은 루트 하위의 모든 엔트리를 순회하면서 "snapshot" 항목을 제외하고 모두 제거한다.
- 파일은 디렉터리 엔트리를 0으로 덮고 링크 카운트를 감소시킨 뒤 iunlockput()로 정리한다. 실제 데이터 블록 해제는 공통 경로(itrunc()→bfree())를 통해 수행되며, bfree()는 A단계에서 도입한 참조수 게이트 덕분에 refcount==0인 경우에만 비트맵을 클리어한다. 스냅샷이 여전히 참조하는 블록은 해제되지 않는다.
- 하위 디렉터리는 재귀적으로 비운 뒤 부모 엔트리 제거와 링크 카운트 조정까지 일관되게 수행한다.

(v) 스냅샷 트리 복제 (모양만, 데이터는 공유)

루트와 타깃 스냅샷 디렉터리를 잠근 뒤 clone_tree(dst=root, src=/snapshot/<id>)를 호출한다. 복제 규칙은 다음과 같다.

- ./..는 스킵하며, 루트에서 "snapshot" 엔트리는 스킵하여 스냅샷 디렉터리 자체를 현재 트리로 끌어오지 않는다.
- T_DIR은 대상에 동일 이름이 없으면 생성하고 재귀적으로 내려간다.
- T_FILE은 파일 껍데기를 만들고 크기/블록 포인터(직접/간접) 메타데이터만 복제한다. 이어서 rc_inc_file_blocks(ch)를 호출하여 원본 파일이 가리키는 모든 데이터 블록의 refcount를 +1 한다. 이로써 현재 트리와 스냅샷 트리는 동일 블록을 공유(ref>1) 하게 된다.
- T_DEV(장치 파일)은 명세에 따라 복제 대상에서 제외한다.

(vi) 트랜잭션 종료 및 refmap 영속화

end_op()로 로그 트랜잭션을 종료/커밋하고, rc_flush()를 호출해 메모리 참조수 테이블을 .refmap으로 한 번 내려쓴다. 이 시점의 flush는 롤백 후 참조수 상태를 디스크에 안정적으로 반영하기 위함이다.

(vii) 반환

모든 단계가 성공하면 0을, 중간 단계에서 오류가 발생하면 음수를 반환한다.

2-4. B : snapshot_delete

2-4-1. 개요

snapshot_delete(snap_id)는 /snapshot/<id> 트리를 안전하게 제거하여 해당 스냅샷이 보유하던 참조를 해소하는 시스템콜이다. 동작 원리는 다음과 같다.

(i) 타깃 디렉터리(/snapshot/<id>) 존재를 확인한 뒤 트랜잭션 경계(begin_op/end_op) 안에서

(ii) 스냅샷 트리의 파일/디렉터리를 재귀적으로 언링크/정리하고(간단한 엔트리 제거와 링크 카운트 조정)

(iii) 파일에 대해선 커널의 일반 해제 경로(itrunc() → bfree())를 통해 데이터 블록을 해제 시도하게 하는 것이다.

이때 블록 해제는 A 단계에서 도입한 refcount 게이트에 의해 참조수가 0일 때에만 실제 비트맵 해제가 일어나며, 현재 파일시스템이 같은 블록을 공유하고 있으면 bfree() 초입의 rc_dec()만 수행되고 물리 해제는 건너뛰어 중복 해제나 데이터 손실을 방지한다.

디렉터리 데이터 블록은 COW 대상이 아니므로 스냅샷 삭제 시에도 별도 refcount 조정이 필요 없고, T_DEV(장치 파일)은 애초에 스냅샷에 포함되지 않으므로 고려 대상이 아니다. 삭제가 완료되면 /snapshot 부모에서 <id> 엔트리를 제거하고, 마지막에 rc_flush()로 메모리 참조수 테이블을 .refmap에 동기화하여 온디스크 일관성을 유지한다. 잘못된 ID(대상 디렉터리 없음)인 경우에는 음수를 반환하도록 하여 오류를 명확히 표지한다. 전체 과정은 /snapshot 하위의 일반 변경 연산을 사용자 시스템콜에서 차단하는 정책과 충돌하지 않도록 커널 내부 전용 경로로 수행되며, 락 순서(부모→자식)와 참조 정리를 준수해 데드락/누수를 방지하도록 설계했다.

2-4-2. sysproc.c

```
int
sys_snapshot_delete(void)
{
  int id;
  if(argint(0, &id) < 0) return -1;

  begin_op();

  rc_ensure_mounted();

  struct inode *snap = namei("/snapshot");
  if(!snap){ end_op(); return -1; }

  char name[16]; itoa10(id, name);

  ilock(snap);
  uint off = 0;
  struct inode *idroot = dirlookup(snap, name, &off);
  iunlock(snap);

  if(!idroot){ end_op(); return -1; }
  ilock(idroot);

  // clear children of /snapshot/[id]
  int ok = clear_tree(idroot, 0);
  iunlock(idroot);

  // unlink [id] entry from /snapshot
  ilock(snap);
  struct dirent z; memset(&z, 0, sizeof(z));
  if(writei(snap, (char*)&z, off, sizeof(z)) != sizeof(z)){
    iunlock(snap); iput(idroot); end_op(); return -1;
  }
  snap->nlink--; iupdate(snap);
  iunlock(snap);

  // decrease link count of idroot
  ilock(idroot);
  idroot->nlink--;
  iupdate(idroot);
  iunlock(idroot);

  iput(idroot);
  end_op();

  rc_flush();
  return ok? 0 : -1;
}
```

이 시스템콜은 인자로 받은 id를 문자열로 변환해 /snapshot/<id> 디렉터리를 찾아 트랜잭션 경계 안에서 안전하게 제거하도록 구성했다. 진입 후 begin_op()로 로그 트랜잭션을 시작하고, rc_ensure_mounted()로 /snapshot과 .refmap의 준비 상태를 보장한다. 이어 namei("/snapshot")로 스냅샷 루트를 얻고, itoa10(id, name)과 dirlookup()을 통해 대상 디렉터리(idroot)와 그 부모 엔트리 오프셋(off)을 확보한다. 타깃이 없으면 음수로 즉시 종료하며, 있으면 ilock(idroot) 후 clear_tree(idroot, 0)를 호출해 하위 파일/디렉터리를 재귀적으로 정리한다. 그다음 부모(/snapshot)를 잠그고, writei()로 off 위치의 디렉터리 엔트리를 제로로 덮어 언링크하며 부모의 nlink를 감소시켜 부모-자식 연결 해제를 완료한다. 타깃 디렉터리 자체의 nlink도 1 감소시켜 참조를 마무리하고 iput(idroot)로 해제한다. 마지막으로 end_op()로 커밋한 뒤 rc_flush()를 1회 호출해 메모리 refcount 테이블을 .refmap에 영속화한다. 중간 단계가 모두 성공하면 0, 그렇지 않으면 -1을 반환한다. 삭제 과정에서 데이터 블록 해제는 공통 경로(itrunc()→bfree())로 흘러가며, A단계에서 도입한 refcount 게이트에 의해 참조수가 0일 때에만 실제 비트맵 해제가 일어나 공유 블록의 오해제를 방지한다.

2-4-3. syscall.h

```
#define SYS_close  21
#define SYS_snapshot_create 22     // Create a snapshot
#define SYS_snapshot_rollback 23   // Rollback to a snapshot
#define SYS_snapshot_delete 24     // Delete a snapshot
```

시스템콜 번호를 새로 정의하여 테이블 인덱스와 일치하도록 한다.

2-4-4. syscall.c

```
extern int sys_uptime(void);
extern int sys_snapshot_create(void);   // snapshot create syscall
extern int sys_snapshot_rollback(void); // snapshot rollback syscall
extern int sys_snapshot_delete(void);   // snapshot delete syscall
```

프로토타입 선언 추가 및 syscalls[] 매핑 테이블에 항목 등록한다.

2-4-5. usys.S

```
SYSCALL(uptime)
SYSCALL(snapshot_create)
SYSCALL(snapshot_rollback)
SYSCALL(snapshot_delete)
```

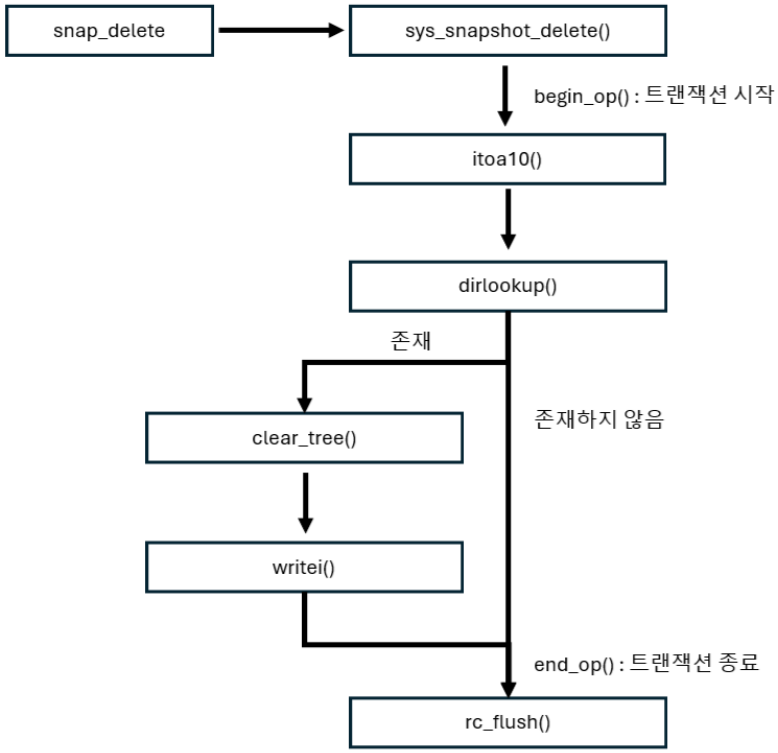
사용자 공간 트랩 스텁 생성 위해 매크로 한 줄 추가한다.

2-4-6. user.h

```
// new syscalls for snapshots
int snapshot_create(void);      // creates a snapshot, returns snapshot
int snapshot_rollback(int id);  // rollbacks to the snapshot with given
int snapshot_delete(int id);    // deletes the snapshot with given id
```

유저 레벨 호출을 위해 함수 원형 선언 추가.

2-4-7. 실행흐름



사용자 프로그램이 snapshot_delete(id)를 호출하면 커널은 sys_snapshot_delete()로 진입하여 먼저 begin_op()로 로그 트랜잭션을 시작한 뒤 rc_ensure_mounted()로 /snapshot 디렉터리와 .refmap 준비 상태를 보장한다. 이어 namei("/snapshot")로 스냅샷 루트를 얻고, itoa10(id, name)과 dirlookup(snap, name, &off)로 /snapshot/<id> 디렉터리(idroot)와 부모 엔트리 오프셋 off를 확보한다. 대상이 없으면 즉시 -1을 반환한다. 존재할 경우 ilock(idroot) 후 clear_tree(idroot, 0)를 호출하여 하위 항목을 모두 재귀적으로 정리한다. 이 과정에서 파일은 디렉터리 엔트리를 0으로 덮고 링크 카운트를 감소시키며, 디렉터리도 하위를 비운 뒤 부모에서 엔트리를 제거한다. 실제 데이터 블록 해제는 공통 경로(itrunc()→bfree())로 수행되며, bfree() 초입의 참조수 게이트(rc_dec 결과가 0일 때만 물리 해제)가 적용되어 공유 중인 블록은 해제되지 않는다. 하위 정리 후 부모(/snapshot)를 잠그고 off 위치의 디렉터리 엔트리를 제로로 덮어 <id> 항목을 언링크하고, 부모 nlink를 감소/갱신한다. 이어 idroot 자체의 nlink도 1 감소시켜 참조를 마무리한 뒤 iput(idroot)로 해제하고 end_op()로 트랜잭션을 커밋한다. 마지막으로 rc_flush()를 1회 호출해 메모리 참조수 테이블을 .refmap에 영속화하며, 전체가 정상 진행되었으면 0, 중간 실패가 있으면 -1을 반환한다.

2-5. C : mk_test_file

2-5-1. 개요

mk_test_file은 스냅샷/블록-COW 검증의 전처리용 데이터 세터이며, 직접 블록 12개가 가득 찬 파일을 만든 뒤 추가 6바이트로 간접 영역을 처음 열게 설계된 테스트 프로그램이다. 이 프로그램으로 생성한 파일을 대상으로 print_addr를 실행하면, addr[0]..addr[11]까지는 직접 블록 주소, addr[12]는 간접 포인터 블록(메타 블록) 주소, 그리고 addr[12] -> [0] (bn: 12)는 간접 테이블의 0번 엔트리(즉, 논리 블록 12번)의 실제 데이터 블록 주소가 출력된다. 이렇게 구성하면 스냅샷 생성 전후, 쓰기 전후의 블록 공유/분기(COW) 여부를 주소 변화로 직관적으로 확인할 수 있다. 주소 자체는 전역 할당 상태에 따라 달라질 수 있으나, 형태(12개 직접 + 1개 간접 포인터 + 간접 첫 데이터)라는 패턴은 동일하게 재현된다. 결과적으로 mk_test_file은 C 단계 전체의 주소 관찰 실험에서 일관된 레이아웃을 가진 베이스라인 파일을 마련해 주는 역할을 한다.

2-5-2. main(int argc, char *argv[])

```c
int
main(int argc, char *argv[])
{
  int fd;

  // create a test file with some data
  if(argc < 2){
    printf(1, "need argv[1]\n");
    exit();
  }

  // open file for write only, create if not exist
  if((fd = open(argv[1], O_CREATE | O_WRONLY)) < 0) {
    printf(1, "open error for %s\n", argv[0]);
    exit();
  }

  // write some data to the file
  char buf[513];
  for(int i=1; i<511; i++) buf[i] = 0;
  buf[511] = '\n';
  for(int i=0; i<12; i++){
    buf[0] = i % 10 + '0';
    write(fd, buf, 512);
  }

  // write string "hello\n" to the file
  char *str = "hello\n";
  write(fd, str, 6);

  // close the file
  close(fd);
  exit();
}
```

프로그램은 첫 번째 인자를 파일명으로 받아 쓰기 전용+없으면 생성 모드로 연다. 그다음 길이 512바이트의 버퍼를 준비하는데, buf[0]에는 반복 인덱스의 1자리 숫자 문자('0'~'9')를, buf[1..510]에는 0을 채우고, buf[511]에는 개행 문자를 둔다. 이 512바이트 레코드를 12회 연속 write하여 파일 크기를 정확히 12×512=6144바이트로 맞춘다. 이어서 "hello₩n" 6바이트를 추가로 write하여 다음 논리 블록(bn=12)을 최초로 접근하게 만든다. xv6의 inode 구조상 NDIRECT=12이므로, 이 시점에 커널의 bmap()은 간접 포인터 블록(메타 블록) 1개와 해당 데이터 블록 1개를 새로 할당한다. 따라서 전체적으로는 데이터 블록 12개(직접) + 간접 포인터 블록 1개 + 간접 데이터 블록 1개가 연쇄적으로 확보되며, print_addr의 기대 포맷(직접 12행 + 간접 포인터 1행 + 간접 첫 엔트리 1행)이 그대로 관찰된다. 이 동작 해석은 xv6의 블록 매핑 규칙에 대한 연역적 추론으로, NDIRECT 경계에서의 bmap() 분기(간접 블록 생성)와 그에 따른 추가 write의 기록 경로를 근거로 한다.

2-6. C : append

2-6-1. 개요

append는 지정한 파일의 끝(EOF)에 문자열을 덧붙이는 간단한 유틸리티이다. 스냅샷/블록 COW 검증 관점에서는, 스냅샷 생성 후 공유(ref>1) 상태의 데이터 블록에 쓰기를 유도하여 커널의 writei() 경로가 COW 분기(새 블록 할당→내용 복사→매핑 교체) 를 수행하는지 관찰하는 데 쓰인다. 즉, 스냅샷 직후 append로 원본 파일을 수정하면, 스냅샷 측 블록은 그대로 보존되고 현재 파일만 다른 블록 주소로 분기되는 현상을 print_addr로 확인할 수 있다. 이러한 용도 때문에 append는 파일의 맨 끝으로 오프셋을 정확히 이동한 뒤 쓰기를 수행하도록 구성되어 있다

2-6-2. main(int argc, char *argv[])

```c
int
main(int argc, char *argv[])
{
  // Check for correct number of arguments
  if(argc != 3){
    printf(2, "Usage: append filename string\n");
    exit();
  }

  // open the file for read+write, create if needed
  int fd = open(argv[1], O_RDWR | O_CREATE);
  if(fd < 0){
    printf(2, "append: cannot open %s\n", argv[1]);
    exit();
  }

  // move offset to the end by reading until EOF
  char buf[1];
  while(read(fd, buf, 1) == 1); // Drain to EOF
  // now at end, write the string
  if(write(fd, argv[2], strlen(argv[2])) < 0){
    printf(2, "append: write failed\n");
    close(fd);
    exit();
  }

  close(fd);
  exit();
}
```

프로그램은 인자 2개(파일명, 덧붙일 문자열)를 요구한다. 부족하면 사용법을 출력하고 종료한다. 읽기/쓰기 가능 + 없으면 생성 모드로 파일을 연다. 이어서 while() 루프를 통해 EOF까지 1바이트씩 읽어내며 파일 오프셋을 끝으로 이동한다. EOF에 도달한 뒤 write(fd, argv[2], strlen(argv[2]))로 문자열을 이어서 기록한다. 이 쓰기가 기존 마지막 데이터 블록 내부라면 내용만 갱신되고, 파일

크기 확장이 필요하면 bmap() 경유로 새 데이터 블록(또는 간접 포인터/데이터 블록)이 할당된다. 특히 스냅샷 직후 공유(ref>1) 블록에 덧붙이면, 커널의 COW 로직이 발동하여 새 블록으로 분기하며, 이후 print_addr에서 해당 논리 블록의 물리 주소가 변경된 것을 확인할 수 있다. 마지막으로 실패 시 메시지를 출력하고, 정상인 경우 파일을 닫고 종료한다.

2-7. C : print_addr

2-7-1. 개요

print_addr는 지정한 파일이 어떤 디스크 블록을 참조하는지를 고정 포맷으로 출력하는 진단/검증용 사용자 프로그램이다. 이 도구는 A단계(COW)와 B단계(스냅샷 생성/복구)에서 블록 공유→쓰기 시 분기(COW)라는 핵심 동작을 주소 수준에서 가시화하기 위해 설계되었으며, NDIRECT=12 직접 블록과 간접 포인터 블록(및 그 첫 엔트리)을 명세 예시와 동일한 형식으로 보여준다. 실행 경로는 사용자 공간의 print_addr → 커널 시스템콜 sys_print_addr로 이어지며, 커널은 대상 파일 inode를 잠그고 직접/간접 포인터를 수집한 뒤 사용자 공간에 전달한다. 사용자는 스냅샷 직후 원본과 /snapshot/<id>/...의 주소 동일성(공유), 원본에 쓰기 후 주소 변경(COW 분기), 롤백 수행 후 주소 원복을 단계별로 확인할 수 있다. 또한 /snapshot 하위는 커널에서 쓰기가 차단되어 있어(읽기만 허용) 스냅샷 측 주소는 항상 불변 기준점으로 활용 가능하다.

2-7-2. print_addr.c

2-7-2-1. 개요

이 사용자 프로그램은 인자 검증만 수행하고 나머지는 커널 시스템콜에 위임하도록 설계되었으며, 출력 포맷은 커널 측 sys_print_addr가 cprintf로 명세 예시와 완전히 동일한 형식으로 직접 출력하도록 구성하였다. 사용자 공간에서 별도의 포맷팅을 하지 않기 때문에, 포맷 불일치나 버퍼 복사/경계 처리로 인한 오차 없이 주소 정보가 일관되게 표시된다는 장점이 있다. 실행 흐름은 "인자 점검 → print_addr(argv[1]) 시스템콜 진입 → 커널에서 inode 잠금 및 직접/간접 블록 주소 수집/출력 → 종료" 순서로 단순화된다.

2-7-2-2. main(int argc, char *argv[])

```
int
main(int argc, char *argv[])
{
  if(argc != 2){
    printf(1, "Usage: print_addr <path>\n");
    exit();
  }

  print_addr(argv[1]);
  exit();
}
```

프로그램은 인자가 정확히 1개(대상 파일 경로)인지 검사한 뒤, 부족하면 사용법 문자열을 출력하고 종료한다. 인자가 올바르면 print_addr(argv[1])를 호출하여 해당 경로를 커널에 그대로 전달한다. 커널은 대상이 파일인지 확인하고, NDIRECT에 해당하는 직접 블록 12개 주소, 간접 포인터 블록 주소, 그리고 간접 블록의 0번 엔트리 주소를 수집하여 고정 포맷으로 출력한다. 유저 프로그램은 반환값에 의존하지 않고 즉시 exit()로 종료하므로, 출력 전체는 커널에서 책임지고 생성된다는 점이 핵심이다. 결과적으로 사용자 단의 로직은 인자 검증 + 시스템콜 위임만 남겨, 테스트 시나리오에서 포맷 일치성 및 신뢰성을 확보한다.

2-7-3. sysproc.c

```
int
sys_print_addr(void)
{
  char *path;
  // get path argument
  if(argstr(0, &path) < 0)
    return -1;

  // get inode
  struct inode *ip = namei(path);
  if(ip == 0)
    return -1;

  // lock inode
  ilock(ip);

  // Only print for regular files/dirs. For others, print nothing.
  if(ip->type != T_FILE && ip->type != T_DIR){
    iunlockput(ip);
    return 0;
  }

  // number of logical blocks in file
  uint nblk = (ip->size + BSIZE - 1) / BSIZE;

  // direct blocks
  uint limit = nblk < NDIRECT ? nblk : NDIRECT;
  for(uint i = 0; i < limit; i++){
    uint a = ip->addrs[i];
    if(a != 0)
      cprintf("addr[%d] : %x\n", i, a);
  }
  // indirect pointer + entries
  if(nblk > NDIRECT){
    uint ib = ip->addrs[NDIRECT];
    if(ib != 0){
      cprintf("addr[%d] : %x (INDIRECT POINTER)\n", NDIRECT, ib);
      struct buf *bp = bread(ip->dev, ib);
      uint *addr = (uint*)bp->data;
      uint cnt = nblk - NDIRECT;
      if(cnt > NINDIRECT) cnt = NINDIRECT;
      for(uint k = 0; k < cnt; k++){
        if(addr[k] != 0)
          cprintf("addr[%d] -> [%d] (bn : %d) : %x\n",
                  NDIRECT, k, NDIRECT + k, addr[k]);
      }
      brelse(bp);
    }
  }

  // finish with newline
  cprintf("\n");

  // release inode
  iunlockput(ip);
  return 0;
}
```

sys_print_addr()는 사용자로부터 전달받은 경로를 argstr(0,&path)로 해석하고 namei(path)로 대상 inode를 찾은 뒤 ilock(ip)로 잠가, 일반 파일 또는 디렉터리(T_FILE/T_DIR)에 한하여 블록 주소를 명세와 동일한 고정 포맷으로 cprintf로 직접 출력하는 커널 측 루틴이다. 우선 파일 크기 기반으로 사용 중인 논리 블록 수 nblk=(ip->size+BSIZE-1)/BSIZE를 계산하고, i=0..min(nblk,NDIRECT)-1 범위의 직접 블록에 대해 addr[%d] : %x 줄을 출력하며, nblk>NDIRECT이면 간접 포인터 블록 ip->addrs[NDIRECT]를 addr[12] : %x (INDIRECT POINTER)로 표시한 다음 bread()로 해당 포인터 블록을 읽어 간접 테이블 엔트리 0..cnt-1(cnt=min(nblk-NDIRECT,NINDIRECT)) 중 0이 아닌 항목을 addr[12] -> [%d] (bn : %d) : %x 형식으로 출력한다. 처리 후에는 brelse(bp)로 버퍼를 해제하고, 마지막에 개행을 한 번 더 출력하여 포맷을 마무리한 다음 iunlockput(ip)로 락과 참조를 정리하고 0을 반환하며, 대상이 파일/디렉터리가 아니면 아무것도 출력하지 않고 조용히 종료하도록 방어 로직을 포함한다.

2-7-4. syscall.h

```
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_snapshot_create 22     // Create a snapshot
#define SYS_snapshot_rollback 23   // Rollback to a snapshot
#define SYS_snapshot_delete 24     // Delete a snapshot
#define SYS_print_addr 25          // Print address mapping
```

시스템콜 번호를 새로 정의하여 테이블 인덱스와 일치하도록 한다.

2-7-5. syscall.c

```
extern int sys_uptime(void);
extern int sys_snapshot_create(void);   // snapshot create syscall
extern int sys_snapshot_rollback(void); // snapshot rollback syscall
extern int sys_snapshot_delete(void);   // snapshot delete syscall
extern int sys_print_addr(void);        // print address syscall
```

```
[SYS_close]   sys_close,
[SYS_snapshot_create]   sys_snapshot_create,   // snapshot create syscall
[SYS_snapshot_rollback] sys_snapshot_rollback, // snapshot rollback sysca
[SYS_snapshot_delete]   sys_snapshot_delete,   // snapshot delete syscall
[SYS_print_addr]        sys_print_addr,        // print address syscall
```

프로토타입 선언 추가 및 syscalls[] 매핑 테이블에 항목 등록한다.

2-7-6. usys.S

```
SYSCALL(uptime)
SYSCALL(snapshot_create)
SYSCALL(snapshot_rollback)
SYSCALL(snapshot_delete)
SYSCALL(print_addr)
```

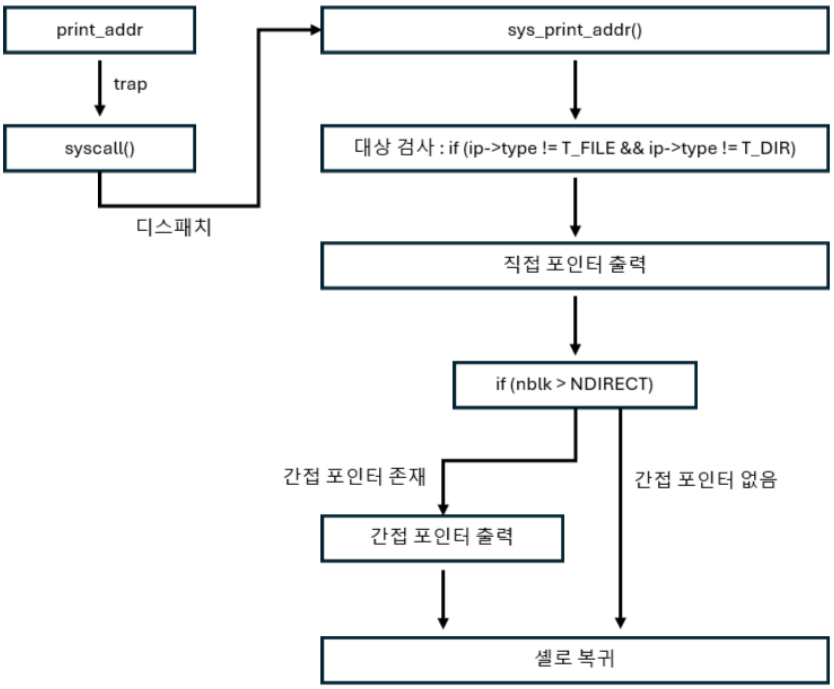
사용자 공간 트랩 스텁 생성 위해 매크로 한 줄 추가한다.

2-7-7. user.h

```
// new syscalls for print_addr
int print_adddr(const char* path); // prints the physical address mapping
```

유저 레벨 호출을 위해 함수 원형 선언 추가.

2-7-8. 호출흐름



사용자 프로그램 print_addr <path>가 실행되면 유저 스텁을 거쳐 시스템콜 트랩으로 커널의 syscall()에 진입하고, 시스템콜 번호(SYS_print_addr)로 디스패치되어 sys_print_addr()가 호출된다. 커널은 argstr(0,&path)로 인자를 해석하고 namei(path)로 대상 inode를 찾은 뒤 ilock(ip)로 잠근다. 대상이 일반 파일/디렉터리가 아니면 조용히 종료하고, 맞다면 파일 크기로부터 논리 블록 수 nblk를 계산한다. 이후 직접 블록 영역(최대 12개)을 addr[i] : <hex> 형식으로 순차 출력하고, 필요한 경우 간접 포인터 블록 주소를 addr[12] : <hex> (INDIRECT POINTER)로 출력한 뒤 그 블록을 bread()하여 테이블 엔트리의 실제 데이터 블록 주소를 addr[12] -> [k] (bn : 12+k) : <hex> 형식으로 나열한다. 마지막으로 개행을 한 번 더 출력하고 버퍼락을 해제한 뒤 0을 반환한다. 경로가 없거나 inode 해석 실패 시에는 음수로 빠르게 반환한다. 출력은 전부 커널에서 cprintf로 생성되므로, 사용자 공간은 포맷팅을 수행하지 않아도 명세와 완전히 동일한 결과를 얻을 수 있다.

3. 실행결과

3-1. print_addr

```
$ print_addr README
addr[0] : 3c
addr[1] : 3d
addr[2] : 3e
addr[3] : 3f
addr[4] : 40
```

```
$ print_addr hi
addr[0] : 359
addr[1] : 35a
addr[2] : 35b
addr[3] : 35c
addr[4] : 35d
addr[5] : 35e
addr[6] : 35f
addr[7] : 360
addr[8] : 361
addr[9] : 362
addr[10] : 363
addr[11] : 364
addr[12] : 365 (INDIRECT POINTER)
addr[12] -> [0] (bn : 12) : 366
```

사용자 프로그램 print_addr 가 파일이 참조하는 디스크 블록 주소를 논리 블록 번호(bn) 순서대로 출력함을 보이고, 이를 통해 xv6 의 직접(0~11)/간접(12~) 블록 배치와 우리 구현의 주소 포맷/해석 규칙이 정확히 작동함을 확인하는 항목이다. 출력은 반드시 addr[%d] : %x 형식을 따르며, 간접 포인터 블록은 addr[12] : %x (INDIRECT POINTER)로 표기하고 이어서 간접 엔트리들을 addr[12] -> [k] (bn : 12+k) : %x 로 나열한다. 이 프로그램은 커널의 inode 를 잠그고 ip->addrs[]의 12 개 직접 포인터와, 필요 시 간접 포인터 블록(한 블록짜리 테이블)을 읽어 실제 데이터 블록 번호를 16 진수로 출력한다. 따라서 스냅샷 전후 혹은 쓰기(COW) 전후의 주소 동일/분기 여부를 눈으로 비교하는 기준선 도구로 쓰인다.

그림 왼쪽의 print_addr README 는 작은 파일이라 직접 블록 5 개만 사용하여 addr[0] : 3c … addr[4] : 40 까지 연속 주소가 출력되었다. 이는 README 가 12 블록 임계치(NDIRECT=12)를 넘지 않아 간접 영역이 필요 없음을 보여준다. 그림 오른쪽의 print_addr hi 는 mk_test_file 로 만든 큰 파일에 대한 결과이다. 먼저 addr[0] : 359 … addr[11] : 364 까지 12 개의 직접 블록이 채워져 있고, 이어 addr[12] : 365 (INDIRECT POINTER)가 등장한다. 여기서 addr[12]는 데이터가 아니라 "간접 포인터 테이블" 블록의 주소이며, 그 테이블의 첫 엔트리([0])가 실제 데이터 블록을 가리킨다. 따라서 addr[12] -> [0] (bn : 12) : 366 은 "논리 블록 12(bn=12)의 데이터는 간접 테이블의 0 번 칸이 가리키는 실제 블록 0x366 에 있다"는 뜻이다.

3-2. snapshot 테스트

3-2-1. snap_create 테스트

```
$ echo AAAA > a
$ mkdir d
$ echo BBBB > d/b
$ snap_create
snapshot_create -> 0
$ ls /snapshot
.              1 28 64
..             1 1 512
.refmap        2 29 1882
0              1 30 416
$ cat a
AAAA
$ cat /snapshot/0/a
AAAA
$ cat d/b
BBBB
$ cat /snapshot/0/d/b
BBBB
$ ls /snapshot/0/dev
ls: cannot open /snapshot/0/dev
```

snap_create 호출 직후 스냅샷 트리가 정확히 생성 및 캡처되었음을 단계적으로 입증하는 로그이다.

먼저 현재 파일시스템에 a 와 d/b 를 만든 뒤 snap_create 를 실행하자 식별자 0 이 반환되며(/snapshot/0), 동시에 /snapshot 루트에 메타데이터 파일(.refmap)이 함께 존재함이 ls /snapshot 로 확인된다. 이어서 cat a 와 cat /snapshot/0/a, cat d/b 와 cat /snapshot/0/d/b 의 결과가 각각 동일하게 출력되어, 스냅샷이 생성 시점의 파일/디렉터리 내용을 루트부터 재귀적으로 그대로

캡처했음을 보여준다. 또한 ls /snapshot/0/dev 가 "열 수 없다"로 실패하는데, 이것으로 장치 파일(T_DEV) 및 /snapshot 자체가 캡처 제외 대상임을 확인할 수 있다. 본 항목에서는 내용 동일성만 보였지만, 스냅샷 생성이 즉시 완료되고 대용량 복제가 보이지 않는 점과 .refmap 이 생성되어 있는 점은 데이터 블록을 복사하지 않고 포인터를 공유한 뒤, 이후 쓰기 시에만 COW 로 분기하는 설계와 일치한다.

3-2-2. snapshot R/O 테스트

```
$ echo x > /snapshot/0/a
SNAP R/O DENY op=open path=/snapshot/0/a
open /snapshot/0/a failed
$ echo x > /snapshot/newfile
SNAP R/O DENY op=open path=/snapshot/newfile
open /snapshot/newfile failed
```

스냅샷 경로에 대한 쓰기 시도가 커널에서 즉시 거부됨을 보여주는 증거이다. 두 경우 모두 커널이 SNAP R/O DENY op=open path=...를 출력하고 사용자 공간에서는 open ... failed 가 반환되었다. 이는 sysfile.c 의 path_is_under_snapshot() 경로에서 /snapshot 하위에 대해 O_WRONLY/O_RDWR 플래그를 가진 open 을 -1 로 거절하도록 한 정책이 정확히 작동한 결과이며, 스냅샷 트리를 읽기 전용으로 유지한다. 이 거절은 트랜잭션(begin_op) 이전의 진입점에서 수행되므로 로그/디스크에 불필요한 사이드이펙트를 남기지 않으며, 스냅샷이 참조하는 블록에 대한 불변성 가정과 이후 COW 분기의 전제가 보장된다.

3-2-3. COW 동작 테스트

```
$ print_addr a
addr[0] : 359

$ print_addr /snapshot/0/a
addr[0] : 359

$ echo ZZZZ > a
$ print_addr a
addr[0] : 363

$ print_addr /snapshot/0/a
addr[0] : 359

$ cat a
ZZZZ
$ cat /snapshot/0/a
AAAA
```

스냅샷 생성 이후 print_addr a 와 print_addr /snapshot/0/a 가 모두 addr[0] : 359 로 동일하게 시작하여, 스냅샷 시점에 데이터 블록 복사 없이 포인터를 공유했음을 보여준다. 이후 원본 파일에 echo ZZZZ > a 로 쓰기를 수행하자, 원본의 addr[0]만 363 으로 변경되고 스냅샷의 addr[0]은 여전히 359 로 유지되어 쓰기 시점에만 새 블록을 할당하는 COW 가 동작했음을 확인하였다. 내용 역시 cat a 는 ZZZZ, cat /snapshot/0/a 는 AAAA 로 분리되어, 스냅샷 트리가 읽기 전용으로 고정되고 원본 변경으로부터 완전히 격리됨을 입증한다.

```
$ mk_test_file hi
$ print_addr hi
addr[0] : 364
addr[1] : 365
addr[2] : 366
addr[3] : 367
addr[4] : 368
addr[5] : 369
addr[6] : 36a
addr[7] : 36b
addr[8] : 36c
addr[9] : 36d
addr[10] : 36e
addr[11] : 36f
addr[12] : 370 (INDIRECT POINTER)
addr[12] -> [0] (bn : 12) : 371
```

```
$ snap_create
snapshot_create -> 1
$ print_addr hi
addr[0] : 364
addr[1] : 365
addr[2] : 366
addr[3] : 367
addr[4] : 368
addr[5] : 369
addr[6] : 36a
addr[7] : 36b
addr[8] : 36c
addr[9] : 36d
addr[10] : 36e
addr[11] : 36f
addr[12] : 370 (INDIRECT POINTER)
addr[12] -> [0] (bn : 12) : 371
```

```
$ print_addr /snapshot/1/hi
addr[0] : 364
addr[1] : 365
addr[2] : 366
addr[3] : 367
addr[4] : 368
addr[5] : 369
addr[6] : 36a
addr[7] : 36b
addr[8] : 36c
addr[9] : 36d
addr[10] : 36e
addr[11] : 36f
addr[12] : 370 (INDIRECT POINTER)
addr[12] -> [0] (bn : 12) : 371
```

```
$ echo HELLO > hi
$ print_addr hi
addr[0] : 374
addr[1] : 365
addr[2] : 366
addr[3] : 367
addr[4] : 368
addr[5] : 369
addr[6] : 36a
addr[7] : 36b
addr[8] : 36c
addr[9] : 36d
addr[10] : 36e
addr[11] : 36f
addr[12] : 370 (INDIRECT POINTER)
addr[12] -> [0] (bn : 12) : 371
```

```
$ print_addr /snapshot/1/hi
addr[0] : 364
addr[1] : 365
addr[2] : 366
addr[3] : 367
addr[4] : 368
addr[5] : 369
addr[6] : 36a
addr[7] : 36b
addr[8] : 36c
addr[9] : 36d
addr[10] : 36e
addr[11] : 36f
addr[12] : 370 (INDIRECT POINTER)
addr[12] -> [0] (bn : 12) : 371
```

(캡처본은 좌->우, 상->하 순으로 순서대로 실행한 결과이다. 1-2-3 / 4-5)

mk_test_file hi 실행 직후 print_addr hi 에 addr[12] : 370 (INDIRECT POINTER)와 addr[12] -> [0] : 371 이 나타나 간접 블록(테이블)과 그 첫 엔트리(bn=12)가 사용 중임을 확인하였다. snap_create 직후 print_addr hi 와 print_addr /snapshot/1/hi 의 모든 주소, 특히 addr[12]와 간접 엔트리 값이 완전히 동일하여 스냅샷 시점에 포인터 공유가 성립함을 다시 한 번 보여준다. 그 다음 echo HELLO > hi 는 파일의 맨 앞(직접 블록 bn=0)만 덮어쓰므로, 원본 hi 에서 addr[0]만 374 로 변경되고 스냅샷의 addr[0]은 그대로 유지되었다. 반면 addr[12]와 그 간접 엔트리는 양쪽 모두 변화가 없는데, 이는 이번 쓰기가 간접 영역을 건드리지 않았기 때문에 간접 포인터 테이블 자체의 COW 경로가 트리거되지 않은 정상 결과이다.

요약하면, 대용량 파일에서도 변경된 블록만 주소가 분기하고 스냅샷 쪽 주소와 내용은 불변으로 유지되어, 명세의 COW/스냅샷 불변 조건이 충족되었음을 보여준다.

3-2-4. snap_rollback 테스트

```
$ snap_create
snapshot_create -> 2
$ echo WORLD > a
$ snap_rollback 2
snapshot_rollback(2) -> 0
$ cat a
ZZZZ
$ print_addr a
addr[0] : 363

$ print_addr /snapshot/2/a
addr[0] : 363

$ snap_rollback 9999
snapshot_rollback(9999) -> -1
$ snap_rollback -1
Usage: snap_rollback <id>
$ snap_rollback foo
Usage: snap_rollback <id>
```

먼저 snap_create 로 ID=2 스냅샷을 만든 뒤 a 를 WORLD 로 덮어쓰고 snap_rollback 2 를 수행하자 반환값 0 이 출력되어 복구 성공을 알린다. 이어서 cat a 가 ZZZZ 를 출력하는데, 이는 ID=2 생성 시점의 내용(이전에 a 가 ZZZZ 였던 상태)으로 정확히 되돌아갔음을 의미한다. 주소 관점에서 print_addr a 와 print_addr /snapshot/2/a 가 모두 addr[0] : 363 으로 일치하여, 데이터 블록 포인터 수준까지 스냅샷 시점과 동일하게 복원되었음을 확인할 수 있다. 이는 복구가 대량 복사 없이 포인터 기반으로 수행되었음을 시사한다. 한편, snap_rollback 9999 가 음수 반환으로 실패하고 snap_rollback -1, snap_rollback foo 가 Usage 안내를 내는 것은 잘못된 ID 입력 시 오류 처리/인자 검증이 구현돼 있음을 보여준다.

3-2-5. snap_delete 테스트

```
$ snap_create
snapshot_create -> 3
$ rm a
$ snap_delete 3
snapshot_delete(3) -> 0
$ ls /snapshot/3
ls: cannot open /snapshot/3
$ snap_delete 9999
snapshot_delete(9999) -> -1
$ snap_delete -1
Usage: snap_delete <id>
$ snap_delete foo
Usage: snap_delete <id>
```

snapshot_delete 의 정상/에러 경로가 명세대로 동작함을 보여주는 로그이다. 먼저 snap_create 로 ID=3 스냅샷을 만든 뒤 현재 파일 a 를 삭제(rm a)하여, 해당 블록들이 스냅샷 쪽에서만 참조될 수 있는 상황을 조성하였다. 이어서 snap_delete 3 이 0 을 반환하고, 직후 ls /snapshot/3 이 "열 수 없다"로 실패하는 것은 /snapshot/3 트리 전체가 해제되어 디렉터리 엔트리 자체가 사라졌음을 의미한다. 내부적으로는 /snapshot/3 하위의 파일,디렉터리를 재귀적으로 제거하면서 각 파일이 가리키는 데이터 블록의 참조수를 감소시키고, 참조수가 0 이 된 블록만 실제로 해제(bfree)되도록 게이트를 두었기 때문에, "블록에 대한 참조가 모두 없어질 때 해당 블록을 할당 해제라는 COW 요구도 함께 만족한다. 이후 snap_delete 9999 가 -1 반환으로 실패하고, snap_delete -1, snap_delete foo 가 Usage 안내를 출력하는 것은 잘못된 ID, 인자에 대한 거부가 구현되어 있음을 보여준다.

4. 소스코드

4-1. append.c

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

int
main(int argc, char *argv[])
{
  // Check for correct number of arguments
  if(argc != 3){
    printf(2, "Usage: append filename string₩n");
    exit();
  }

  // open the file for read+write, create if needed
  int fd = open(argv[1], O_RDWR | O_CREATE);
  if(fd < 0){
    printf(2, "append: cannot open %s₩n", argv[1]);
    exit();
  }

  // move offset to the end by reading until EOF
  char buf[1];
  while(read(fd, buf, 1) == 1); // Drain to EOF
  // now at end, write the string
  if(write(fd, argv[2], strlen(argv[2])) < 0){
    printf(2, "append: write failed₩n");
    close(fd);
    exit();
  }

  close(fd);
  exit();
}
```

4-2. defs.h

```
// ... (생략) ...
// snapshot.c
void   rc_init(void);                  // Initialize the read cache
void   rc_ensure_mounted(void);        // Ensure the snapshot filesystem is mounted
int    rc_get(uint blkno);             // Get a block from the read cache
int    rc_inc(uint blkno);             // Increment the reference count for a block in the read cache
int    rc_dec(uint blkno);             // Decrement the reference count for a block in the read cache
void   rc_mark_alloc(uint blkno);      // Mark a block as allocated in the read cache
void   rc_flush(void);                 // Flush the read cache to disk
```

4-3. fs.c

```
// ... (생략) ...
// Allocate a zeroed disk block.
```

```
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);

        rc_mark_alloc(b + bi); // read cache

        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}

// Copy-on-Write: set block number 'newb' to inode 'ip' at block number 'bn'
static void
cow_set_block(struct inode *ip, uint bn, uint newb)
{
  if(bn < NDIRECT){
    ip->addrs[bn] = newb;
    iupdate(ip);
    return;
  }
  bn -= NDIRECT;

  // indirect block
  if(bn >= NINDIRECT)
    panic("cow_set_block: bn out");

  // read indirect block
  uint a = ip->addrs[NDIRECT];
  if(a == 0)
    panic("cow_set_block: no indirect");

  // COW for indirect block itself
  if(rc_get(a) > 1){
```

```
    uint na = balloc(ip->dev);   // allocate new indirect block
    struct buf *obp = bread(ip->dev, a);
    struct buf *nbp = bread(ip->dev, na);
    memmove(nbp->data, obp->data, BSIZE);
    log_write(nbp);
    brelse(nbp);
    brelse(obp);
    ip->addrs[NDIRECT] = na;   // update to new indirect block
    iupdate(ip);
    rc_dec(a);   // drop our ref to old
    a = na;
    cprintf("[COW-INDIR] inum=%d old=%u new=%u₩n", ip->inum, a, na);
  }

  // update indirect block
  struct buf *bp = bread(ip->dev, a);
  uint *addr = (uint*)bp->data;
  if(addr[bn] == 0)
    panic("cow_set_block: indir slot empty");

  // set new block
  addr[bn] = newb;
  log_write(bp);
  brelse(bp);

  iupdate(ip);
}

// Free a disk block.
static void
bfree(int dev, uint b)
{
  if(b >= sb.size) panic("bfree: b out of range"); // sanity check
  int after = rc_dec(b); // read cache
  if(after > 0) return;   // still referenced
// ... (중략) ...
void
iinit(int dev)
{
  int i = 0;

  initlock(&icache.lock, "icache");
  for(i = 0; i < NINODE; i++) {
    initsleeplock(&icache.inode[i].lock, "inode");
  }

  readsb(dev, &sb);
  rc_init(); // read cache
// ... (중략) ...
```

```
int
writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip, src, n);
  }

  int cow_enabled = (ip->type == T_FILE);

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > MAXFILE*BSIZE)
    return -1;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    // COW: if block is shared, do block-level COW
    uint bn = off/BSIZE;
    uint b = bmap(ip, bn);                  // may allocate if needed

    // If existing block is shared, do block-level COW.
    if(cow_enabled && b != 0){
      if(rc_get(b) > 1){
        uint nb = balloc(ip->dev);                  // allocate new block
        struct buf *ob = bread(ip->dev, b);     // old block
        struct buf *nbp = bread(ip->dev, nb); // new block
        memmove(nbp->data, ob->data, BSIZE);   // copy data
        log_write(nbp);                             // write new block
        brelse(nbp);                                // release new block
        brelse(ob);                                 // release old block
        cow_set_block(ip, bn, nb);                 // switch mapping
        rc_dec(b);                                  // drop our ref to old
        b = nb;
      }
    }
    bp = bread(ip->dev, b); // read the block, no leaked buffer

    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }
  // ... (생략) ...
```

4-4. Makefile

```
OBJS = ₩
```

```
        bio.o₩
  // ... (중략) ...
        vm.o₩
        snapshot.o₩
  // ... (중략) ...
  UPROGS=₩
  // ... (중략) ...
        _zombie₩
        _mk_test_file₩
        _append₩
        _print_addr₩
        _snap_create₩
        _snap_delete₩
        _snap_rollback₩
  // ... (생략) ...
```

4-5. mk_test_file.c

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

  int
  main(int argc, char *argv[])
  {
    int fd;

    // create a test file with some data
    if(argc < 2){
      printf(1, "need argv[1]₩n");
      exit();
    }

    // open file for write only, create if not exist
    if((fd = open(argv[1], O_CREATE | O_WRONLY)) < 0) {
      printf(1, "open error for %s₩n", argv[0]);
      exit();
    }

    // write some data to the file
    char buf[513];
    for(int i=1; i<511; i++) buf[i] = 0;
    buf[511] = '₩n';
    for(int i=0; i<12; i++){
      buf[0] = i % 10 + '0';
      write(fd, buf, 512);
    }

    // write string "hello₩n" to the file
    char *str = "hello₩n";
    write(fd, str, 6);
```

```
    // close the file
    close(fd);
    exit();
  }
```

4-6. print_addr.c

```
  // print_addr.c
  #include "types.h"
  #include "stat.h"
  #include "user.h"

  int
  main(int argc, char *argv[])
  {
    if(argc != 2){
      printf(1, "Usage: print_addr <path>₩n");
      exit();
    }

    print_addr(argv[1]);
    exit();
  }
```

4-7. snap_creaete.c

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

  int
  main(int argc, char *argv[])
  {
    if(argc != 1) {
      printf(1, "Usage: snap_create₩n");
      exit();
    }

    int id = snapshot_create();
    // error handling if snapshot creation failed
    if(id < 0){
      printf(1, "snapshot_create failed₩n");
      exit();
    }
    printf(1, "snapshot_create -> %d₩n", id); // print the snapshot id

    exit();
  }
```

4-8. snap_delete.c

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

```
static int parse_uint(const char *s, int *ok){
  int n=0; *ok=0;
  if(!s || !*s) return 0;
  for(int i=0; s[i]; i++){
    if(s[i]<'0' || s[i]>'9') return 0;
    n = n*10 + (s[i]-'0');
  }
  *ok=1; return n;
}

int main(int argc, char **argv){
  if(argc != 2){
    printf(1, "Usage: snap_delete <id>₩n");
    exit();
  }
  int ok=0; int id = parse_uint(argv[1], &ok);
  if(!ok){
    printf(1, "Usage: snap_delete <id>₩n");
    exit();
  }
  int r = snapshot_delete(id);
  printf(1, "snapshot_delete(%d) -> %d₩n", id, r);
  exit();
}
```

4-9. snap_rollback.c

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

static int parse_uint(const char *s, int *ok){
  int n=0; *ok=0;
  if(!s || !*s) return 0;
  for(int i=0; s[i]; i++){
    if(s[i]<'0' || s[i]>'9') return 0;
    n = n*10 + (s[i]-'0');
  }
  *ok=1; return n;
}

int main(int argc, char **argv){
  if(argc != 2){
    printf(1, "Usage: snap_rollback <id>₩n");
    exit();
  }
  int ok=0; int id = parse_uint(argv[1], &ok);
  if(!ok){
    printf(1, "Usage: snap_rollback <id>₩n");
    exit();
```

```
    }
    int r = snapshot_rollback(id);
    printf(1, "snapshot_rollback(%d) -> %d₩n", id, r);
    exit();
  }
```

4-10. snapshot.c

```
  // snapshot.c — A-1: refcount core (lazy-init, no disk I/O at boot)

  #include "types.h"
  #include "defs.h"
  #include "param.h"
  #include "stat.h"
  #include "mmu.h"
  #include "spinlock.h"
  #include "sleeplock.h"
  #include "fs.h"
  #include "buf.h"
  #include "file.h"

  extern struct superblock sb;

  #ifndef BPB
  #define BPB (BSIZE*8)
  #endif

  #define SNAPDIR   "snapshot"
  #define RC_FNAME ".refmap"

  static struct spinlock rc_lock;

  static ushort *rc_tab;        // in-memory refcount table (data blocks only)
  static uint       rc_nblks;     // == sb.nblocks
  static uint       rc_data_start; // first absolute data block

  static struct inode *rc_ip;       // inode for /snapshot/.refmap (lazy)
  static int rc_saving;               // reentrancy guard for save_refmap()
  static int rc_ready;                // .refmap is present & table synced?

  // -------- internal helpers --------

  // get index into rc_tab for block b
  static inline int rc_idx(uint b, uint *idx_out){
    if(b < rc_data_start) return 0; // not a data block
    uint idx = b - rc_data_start;     // data block index
    if(idx >= rc_nblks) return 0;     // out of range
    *idx_out = idx;
    return 1;
  }
```

```
// ensure /snapshot and /snapshot/.refmap exist
static void ensure_snapshot_layout(void){
  struct inode *root = namei("/");
  if(root == 0) panic("ensure_snapshot_layout: no root");

  // /snapshot
  ilock(root);
  struct inode *snap = dirlookup(root, SNAPDIR, 0); // look for /snapshot
  if(snap == 0){
    snap = ialloc(root->dev, T_DIR);
    if(!snap) panic("mk SNAPDIR: ialloc");
    ilock(snap);
    snap->nlink = 1; iupdate(snap); // link count for "."
    // create . and .. entries
    if(dirlink(snap, ".",   snap->inum) < 0) panic("mk SNAPDIR: .");
    if(dirlink(snap, "..", root->inum) < 0) panic("mk SNAPDIR: ..");
    if(dirlink(root, SNAPDIR, snap->inum) < 0) panic("mk SNAPDIR: link");
    root->nlink++; iupdate(root);
    iunlock(snap);
  }
  iunlock(root);

  // /snapshot/.refmap
  ilock(snap);
  struct inode *ip = dirlookup(snap, RC_FNAME, 0);
  if(ip == 0){
    ip = ialloc(snap->dev, T_FILE);
    if(!ip) panic("mk .refmap: ialloc");
    ilock(ip);
    ip->nlink = 1; iupdate(ip); // link count
    // link into /snapshot
    if(dirlink(snap, RC_FNAME, ip->inum) < 0) panic("mk .refmap: link");
    iunlock(ip);
  }
  iunlock(snap);

  rc_ip = ip; // hold rc_ip reference
  iput(snap);
  iput(root);
}

// save rc_tab into rc_ip on disk
static void save_refmap(void){
  if(!rc_ip || !rc_tab) panic("save_refmap: uninit");
  if(rc_saving) return;
  rc_saving = 1;

  begin_op();
  ilock(rc_ip);
```

```
  uint need = rc_nblks * sizeof(ushort);
  if(rc_ip->size != need){
    uint off = 0;
    char z[BSIZE];
    memset(z, 0, BSIZE);
    while(off < need){
      uint m = (need - off > BSIZE) ? BSIZE : (need - off);
      if(writei(rc_ip, z, off, m) != m) panic("save_refmap: grow");
      off += m;
    }
  }
  if(writei(rc_ip, (char*)rc_tab, 0, need) != need)
    panic("save_refmap: write");

  iunlock(rc_ip);
  end_op();
  rc_saving = 0;
}

// load .refmap from disk into rc_tab
static void load_refmap(void){
  uint need = rc_nblks * sizeof(ushort);
  if(need > PGSIZE) panic("load_refmap: table too big");

  rc_tab = (ushort*)kalloc();
  if(!rc_tab) panic("load_refmap: kalloc");
  memset(rc_tab, 0, PGSIZE);

  ilock(rc_ip);
  uint have = rc_ip->size;
  if(have >= need){
    if(readi(rc_ip, (char*)rc_tab, 0, need) != need)
      panic("load_refmap: read");
    iunlock(rc_ip);
  }else{
    iunlock(rc_ip);
    save_refmap();
  }
}

// -------- public API --------

// Boot-time init (NO disk I/O). Just set up memory table.
void rc_init(void){
  initlock(&rc_lock, "rc");
  // set parameters from superblock
  rc_nblks       = sb.nblocks;
  rc_data_start = sb.bmapstart + ((sb.size + BPB - 1)/BPB);
```

```
  // allocate in-memory table
  uint need = rc_nblks * sizeof(ushort);
  if(need > PGSIZE) panic("rc_init: table too big");

  // allocate zeroed table
  rc_tab = (ushort*)kalloc();
  if(!rc_tab) panic("rc_init: kalloc");
  memset(rc_tab, 0, PGSIZE);

  // mark uninitialized
  rc_ip = 0;
  rc_ready = 0;
  rc_saving = 0;

  cprintf("rc_init(minimal): data_start=%u data_blocks=%u bytes=%u₩n",
          rc_data_start, rc_nblks, rc_nblks * (int)sizeof(ushort));
}

// Call once later (e.g., at start of snapshot syscalls).
void rc_ensure_mounted(void){
  if(rc_ready) return;

  ensure_snapshot_layout();   // /snapshot, .refmap

  uint need = rc_nblks * sizeof(ushort);

  if(rc_tab == 0){
    // load from disk
    load_refmap();
  } else {
    ilock(rc_ip);
    uint have = rc_ip->size; // existing size on disk
    iunlock(rc_ip);
    if(have < need) save_refmap(); // grow on disk
  }
  rc_ready = 1;
}

// get current refcount for block b
int rc_get(uint b){
  uint idx; if(!rc_idx(b, &idx)) return 0;
  int v; acquire(&rc_lock); v = rc_tab[idx]; release(&rc_lock); return v;
}

// return new refcount after increment; never above 0xFFFF
int rc_inc(uint b){
  // get index
  uint idx; if(!rc_idx(b, &idx)) return 0;
```

```
  int v; acquire(&rc_lock);
  // prevent overflow
  if(rc_tab[idx] < 0xFFFF) rc_tab[idx]++;
  // return new value
  v = rc_tab[idx]; release(&rc_lock); return v;
}

// return new refcount after decrement; never below 0
int rc_dec(uint b){
  // get index
  uint idx; if(!rc_idx(b, &idx)) return 0;
  int v; acquire(&rc_lock);
  if(rc_tab[idx] > 0) rc_tab[idx]--;   // prevent underflow
  // return new value
  v = rc_tab[idx]; release(&rc_lock); return v;
}

// Mark block as allocated (refcount >= 1). No-op if already allocated.
void rc_mark_alloc(uint b){
  // get index
  uint idx; if(!rc_idx(b, &idx)) return;
  acquire(&rc_lock);
  // mark as allocated
  if(rc_tab[idx] == 0) rc_tab[idx] = 1;
  release(&rc_lock);
}

// For now, keep it safe: do nothing if called during save.
void rc_flush(void){
  if(!rc_ready) return;     // not mounted yet → nothing to do
  if(rc_saving) return;     // reentrancy guard
  save_refmap();
}
```

4-11. syscall.c

```
// ... (생략) ...
extern int sys_snapshot_create(void);     // snapshot create syscall
extern int sys_snapshot_rollback(void); // snapshot rollback syscall
extern int sys_snapshot_delete(void);     // snapshot delete syscall
extern int sys_print_addr(void);              // print address syscall

static int (*syscalls[])(void) = {
[SYS_fork]       sys_fork,
// ... (중략) ...
[SYS_close]     sys_close,
[SYS_snapshot_create]     sys_snapshot_create,     // snapshot create syscall
[SYS_snapshot_rollback] sys_snapshot_rollback, // snapshot rollback syscall
[SYS_snapshot_delete]     sys_snapshot_delete,     // snapshot delete syscall
[SYS_print_addr]              sys_print_addr,              // print address syscall
};
```

```
  // ... (생략) ...
```
4-12. syscall.h
```
  // ... (생략) ...
  #define SYS_close   21
  #define SYS_snapshot_create 22        // Create a snapshot
  #define SYS_snapshot_rollback 23     // Rollback to a snapshot
  #define SYS_snapshot_delete 24        // Delete a snapshot
  #define SYS_print_addr 25               // Print address mapping
```
4-13. sysfile.c
```
  // ... (생략) ...
  int
  sys_link(void)
  {
    char name[DIRSIZ], *new, *old;
    struct inode *dp, *ip;

    if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
      return -1;

    // Deny link if new path is under /snapshot
    if(path_is_under_snapshot(new)){
      cprintf("SNAP R/O DENY op=mkdir path=%s₩n", new);
      return -1;
    }
  // ... (중략) ...
  int
  sys_unlink(void)
  {
    struct inode *ip, *dp;
    struct dirent de;
    char name[DIRSIZ], *path;
    uint off;

    if(argstr(0, &path) < 0)
      return -1;

    // Deny unlink if path is under /snapshot
    if(path_is_under_snapshot(path)){
      cprintf("SNAP R/O DENY op=mkdir path=%s₩n", path);
      return -1;
    }
  // ... (중략) ...
int
sys_open(void)
{
  char *path;
  int fd, omode;
  struct file *f;
  struct inode *ip;
```

```
  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  if(path_is_under_snapshot(path)){
    if(omode & O_WRONLY || omode & O_RDWR || omode & O_CREATE){
      cprintf("SNAP R/O DENY op=open path=%s₩n", path);
      return -1; // Deny open if path is under /snapshot
    }
  }
 // ... (중략) ...
 int
 sys_mkdir(void)
 {
   char *path;
   struct inode *ip;

   // invalid path check
   if(argstr(0, &path) < 0) return -1;

   // Deny mkdir if path is under /snapshot
   if(path_is_under_snapshot(path)){
     cprintf("SNAP R/O DENY op=mkdir path=%s₩n", path);
     return -1;
   }
 // ... (중략) ...
 int
 sys_mknod(void)
 {
   struct inode *ip;
   char *path;
   int major, minor;

   if(argstr(0, &path) < 0 || argint(1, &major) < 0 || argint(2, &minor) < 0) return -1;

   // Deny mknod if path is under /snapshot
   if(path_is_under_snapshot(path)){
    cprintf("SNAP R/O DENY op=mkdir path=%s₩n", path);
     return -1;
   }
 // ... (생략) ...
```

4-14. sysproc.c

```
 // ... (생략) ...

 // ===== Snapshots: helpers + syscalls (fixed includes/types) =====

 static int itoa10(int x, char *buf){
   int i=0, j=0; char tmp[16];
   if(x==0){ buf[0]='0'; buf[1]=0; return 1; }
```

```
  while(x>0){ tmp[i++] = '0' + (x%10); x/=10; }
  while(i>0) buf[j++] = tmp[--i];
  buf[j]=0; return j;
}

static int streq(const char *a, const char *b){
  while(*a && *b){ if(*a!=*b) return 0; a++; b++; }
  return *a==0 && *b==0;
}

// ---------- directory/file creators (parent must be ilock()'d) ----------

static struct inode*
mkdir_locked(struct inode *dp, char *name)
{
  struct inode *ip;

  // allocate new directory inode
  ip = ialloc(dp->dev, T_DIR);
  if(ip == 0) panic("mkdir_locked: ialloc");
  ilock(ip);
  ip->nlink = 1;            // for "."
  iupdate(ip);

  // "." and ".."
  if(dirlink(ip, ".",   ip->inum) < 0)   panic("mkdir_locked: '.'");
  if(dirlink(ip, "..", dp->inum) < 0)   panic("mkdir_locked: '..'");

  // link into parent
  if(dirlink(dp, name, ip->inum) < 0)   panic("mkdir_locked: link parent");

  // parent has one more subdir
  dp->nlink++;
  iupdate(dp);

  iunlock(ip);
  return ip; // caller still holds a ref via icache
}

static struct inode*
create_file_locked(struct inode *dp, char *name)
{
  // allocate new file inode
  struct inode *ip = ialloc(dp->dev, T_FILE);
  if(ip == 0) panic("create_file_locked: ialloc");

  ilock(ip); // lock new inode
  ip->nlink = 1;
  iupdate(ip);
```

```
  // link into parent
  if(dirlink(dp, name, ip->inum) < 0) panic("create_file_locked: dirlink");
  iunlock(ip); // unlock new inode
  return ip;
}

// ---------- refcount helpers for file data blocks ----------

static void rc_inc_file_blocks(struct inode *f)
{
  // direct
  for(int i=0;i<NDIRECT;i++){
    uint b = f->addrs[i];
    if(b) rc_inc(b);
  }
  // indirect entries (if present)
  uint ib = f->addrs[NDIRECT];
  if(ib){
    rc_inc(ib); // indirect block itself
    struct buf *bp = bread(f->dev, ib);
    uint *addr = (uint*)bp->data;
    for(int k=0;k<NINDIRECT;k++){
      uint b = addr[k];
      if(b) rc_inc(b);
    }
    brelse(bp);
  }
}

static int dir_is_root(struct inode *dp){
  return dp->inum == ROOTINO;
}

// clone contents of src_dir into dst_dir
static int
clone_tree(struct inode *dst_dir, struct inode *src_dir)
{
  struct dirent de;
  uint off = 0;
  int n;

  while((n = readi(src_dir, (char*)&de, off, sizeof(de))) == sizeof(de)){
    off += sizeof(de);
    // skip invalid entries
    if(de.inum == 0) continue;
    if(streq(de.name, ".") || streq(de.name, "..")) continue;
    if(dir_is_root(src_dir) && streq(de.name, "snapshot")) continue;

    // lookup child in src_dir
```

```
uint child_off = 0;
struct inode *ch = dirlookup(src_dir, de.name, &child_off);
if(ch == 0) continue;
ilock(ch);

if(ch->type == T_DIR){
   // create subdir in dst_dir
   struct inode *sub = dirlookup(dst_dir, de.name, 0);
   if(!sub) sub = mkdir_locked(dst_dir, de.name);

   ilock(sub);
   int ok = clone_tree(sub, ch); // recursive clone
   iunlock(sub);
   iput(sub);

   iunlock(ch);
   iput(ch);
   // check recursive result
   if(!ok) return 0;

} else if(ch->type == T_FILE){
   struct inode *nf = dirlookup(dst_dir, de.name, 0);
   if(!nf) nf = create_file_locked(dst_dir, de.name);

   ilock(nf);

   // copy metadata + block pointers
   nf->size = ch->size;
   for(int i=0;i<NDIRECT;i++) nf->addrs[i] = ch->addrs[i];
   nf->addrs[NDIRECT] = ch->addrs[NDIRECT];
   iupdate(nf);

   // increment refcounts for data blocks
   rc_inc_file_blocks(ch);

   iunlock(nf);
   iput(nf);
   iunlock(ch);
   iput(ch);

} else if(ch->type == T_DEV){
   // skip device files
   iunlock(ch);
   iput(ch);
   continue;
} else {
   // unknown type
   iunlock(ch);
   iput(ch);
```

```
    }
  }
  if(n < 0) return 0;
  return 1;
}

// clear all children under 'dir'
static int
clear_tree(struct inode *dir, int skip_snapshot)
{
  struct dirent de;
  uint off = 0;
  int n;

  while((n = readi(dir, (char*)&de, off, sizeof(de))) == sizeof(de)){
    off += sizeof(de);
    if(de.inum == 0) continue;
    if(streq(de.name, ".") || streq(de.name, "..")) continue;
    if(skip_snapshot && dir_is_root(dir) && streq(de.name, "snapshot"))
      continue;

    uint off_child = 0;
    struct inode *ip = dirlookup(dir, de.name, &off_child);
    if(ip == 0) continue;
    ilock(ip);

    if(ip->type == T_DIR){
      int ok = clear_tree(ip, 0);
      if(!ok){ iunlockput(ip); return 0; }

      // remove subdir entry from parent
      struct dirent z; memset(&z, 0, sizeof(z));
      if(writei(dir, (char*)&z, off_child, sizeof(z)) != sizeof(z)){
        iunlockput(ip); return 0;
      }
      // fix link counts
      dir->nlink--; iupdate(dir);
      ip->nlink--;   iupdate(ip);
      iunlockput(ip);

    } else if(ip->type == T_FILE){
      // unlink file entry
      struct dirent z; memset(&z, 0, sizeof(z));
      if(writei(dir, (char*)&z, off_child, sizeof(z)) != sizeof(z)){
        iunlockput(ip); return 0;
      }
      ip->nlink--; iupdate(ip);
      iunlockput(ip);
```

```
    } else if(ip->type == T_DEV){
      // unlink device entry
      struct dirent z; memset(&z, 0, sizeof(z));
      if(writei(dir, (char*)&z, off_child, sizeof(z)) != sizeof(z)){
        iunlockput(ip); return 0;
      }
      ip->nlink--; iupdate(ip);
      iunlockput(ip);

    } else {
      iunlockput(ip);
    }
  }
  if(n < 0) return 0;
  return 1;
}

// ---------- syscalls ----------

int
sys_snapshot_create(void)
{
  begin_op();

  rc_ensure_mounted();    // lazy-mount /snapshot and load .refmap

  struct inode *root = namei("/");
  struct inode *snap = namei("/snapshot");
  if(!root || !snap){ end_op(); return -1; }

  // find next available numeric ID under /snapshot
  char name[16];
  int id = 0;
  for(;; id++){
    itoa10(id, name);
    ilock(snap);
    struct inode *exist = dirlookup(snap, name, 0);
    iunlock(snap);
    if(!exist) break;
    iput(exist);
  }

  // create /snapshot/[id]
  ilock(snap);
  struct inode *dst = dirlookup(snap, name, 0);
  if(!dst) dst = mkdir_locked(snap, name);
  iunlock(snap);

  // clone root -> /snapshot/[id]
```

```
  ilock(root);
  ilock(dst);
  int ok = clone_tree(dst, root);
  iunlock(dst);
  iunlock(root);

  end_op();

  rc_flush();   // persist refmap once

  if(!ok) return -1;
  return id;
}

int
sys_snapshot_delete(void)
{
  int id;
  if(argint(0, &id) < 0) return -1;
  if(id < 0) return -1;

  begin_op();

  rc_ensure_mounted();

  struct inode *snap = namei("/snapshot");
  if(!snap){ end_op(); return -1; }

  char name[16]; itoa10(id, name);

  ilock(snap);
  uint off = 0;
  struct inode *idroot = dirlookup(snap, name, &off);
  iunlock(snap);

  if(!idroot){ end_op(); return -1; }
  ilock(idroot);

  // clear children of /snapshot/[id]
  int ok = clear_tree(idroot, 0);
  iunlock(idroot);

  // unlink [id] entry from /snapshot
  ilock(snap);
  struct dirent z; memset(&z, 0, sizeof(z));
  if(writei(snap, (char*)&z, off, sizeof(z)) != sizeof(z)){
    iunlock(snap); iput(idroot); end_op(); return -1;
  }
  snap->nlink--; iupdate(snap);
```

```
  iunlock(snap);

  // decrease link count of idroot
  ilock(idroot);
  idroot->nlink--;
  iupdate(idroot);
  iunlock(idroot);

  iput(idroot);
  end_op();

  rc_flush();
  return ok? 0 : -1;
}

int
sys_snapshot_rollback(void)
{
  int id;
  if(argint(0, &id) < 0) return -1;
  if(id < 0) return -1;

  begin_op();

  rc_ensure_mounted();

  struct inode *snap = namei("/snapshot");
  struct inode *root = namei("/");
  if(!snap || !root){ end_op(); return -1; }

  char name[16]; itoa10(id, name);

  // find snapshot source dir
  ilock(snap);
  struct inode *src = dirlookup(snap, name, 0);
  iunlock(snap);
  if(!src){ end_op(); return -1; }

  // 1) clear current root (keep /snapshot if present)
  ilock(root);
  int ok = clear_tree(root, /*skip_snapshot=*/1);
  iunlock(root);
  if(!ok){ iput(src); end_op(); return -1; }

  // 2) clone snapshot tree into root
  ilock(root);
  ilock(src);
  ok = clone_tree(root, src);
  iunlock(src);
```

```
  iunlock(root);

  iput(src);
  end_op();

  rc_flush();
  return ok? 0 : -1;
}

// ========== print_addr ==========
int
sys_print_addr(void)
{
  char *path;
  // get path argument
  if(argstr(0, &path) < 0)
    return -1;

  // get inode
  struct inode *ip = namei(path);
  if(ip == 0)
    return -1;

  // lock inode
  ilock(ip);

  // Only print for regular files/dirs. For others, print nothing.
  if(ip->type != T_FILE && ip->type != T_DIR){
    iunlockput(ip);
    return 0;
  }

  // number of logical blocks in file
  uint nblk = (ip->size + BSIZE - 1) / BSIZE;

  // direct blocks
  uint limit = nblk < NDIRECT ? nblk : NDIRECT;
  for(uint i = 0; i < limit; i++){
    uint a = ip->addrs[i];
    if(a != 0)
      cprintf("addr[%d] : %x₩n", i, a);
  }

  // indirect pointer + entries
  if(nblk > NDIRECT){
    uint ib = ip->addrs[NDIRECT];
    if(ib != 0){
      cprintf("addr[%d] : %x (INDIRECT POINTER)₩n", NDIRECT, ib);
      struct buf *bp = bread(ip->dev, ib);
```

```
        uint *addr = (uint*)bp->data;
        uint cnt = nblk - NDIRECT;
        if(cnt > NINDIRECT) cnt = NINDIRECT;
        for(uint k = 0; k < cnt; k++){
          if(addr[k] != 0)
            cprintf("addr[%d] -> [%d] (bn : %d) : %x₩n",
                    NDIRECT, k, NDIRECT + k, addr[k]);
        }
        brelse(bp);
      }
    }

    // finish with newline
    cprintf("₩n");

    // release inode
    iunlockput(ip);
    return 0;
  }
```

4-15. user.h

```
  // ... (생략) ...
  // new syscalls for snapshots
  int snapshot_create(void);          // creates a snapshot, returns snapshot id
  int snapshot_rollback(int id);   // rollbacks to the snapshot with given id
  int snapshot_delete(int id);       // deletes the snapshot with given id

  // new syscalls for print_addr
  int print_addr(const char* path); // prints the physical address mapping
```

4-16. usys.S

```
  // ... (생략) ...
  SYSCALL(uptime)
  SYSCALL(snapshot_create)
  SYSCALL(snapshot_rollback)
  SYSCALL(snapshot_delete)
  SYSCALL(print_addr)
```