컴퓨터학부 20221494 박찬솔

## 1. 개요

xv6의 프로세스 메모리 동적 할당 함수 malloc은 가상 주소 공간을 생성하면서 물리 주소 공간도 함께 할당받는다. 한편, ssualloc 시스템 콜은 가상 주소 공간을 생성하지만, 실제로 해당 주소 공간에 접근할 때 물리 주소 공간을 할당받는다. 또한, 프로세스마다 할당받은 가상 주소와 물리 주소의 페이지 개수를 각각 세고, 이를 시스템 콜로서 개수를 가져올 수 있도록 한다. xv6의 파일 시스템은 12개의 직접(direct) 블록과 1개의 이중 간접(indirect) 블록이 있다. 이를 최대 삼중 간접으로 수정하여 최대 1GB 크기의 파일 저장을 지원하도록 수정한다. 처리한 트랩에 관한 사항은 보고서의 가장 아래에 있다.

## 2. 설계

시스템 콜 추가는 user.h, usys.S, syscall.h, syscall.c, sysproc.c의 파일을 수정한다. 시스템 콜 추가에 관한 내용은 사소하므로 생략한다.

ssualloc은 물리 주소 공간을 할당받지 않으며, 당연히 페이지 테이블(디렉토리)에도 어떠한 엔트리를 추가하지 않는다. 그러나, 프로세스가 해당 가상 주소 공간에 접근하려고 시도하면 당연히 물리 주소 공간이 없으므로 illegal한 접근으로써 메모리 폴트 트랩(FLT)이 발생한다. 이때, 트랩 핸들러에서 폴트가 발생한 가상 주소 공간과 물리 주소 공간을 매핑시켜준다. 매핑은 mappages 함수를 사용하며, 물리 주소 할당에는 kalloc 함수를 사용한다.

삼중 간접 파일 시스템으로 수정하기 위해서는 inode와 dinode 구조체 및 기존에 단일 간접(single indirect) 블록을 나타내던 매크로 상수 등을 모두 수정한다. 또한, dinode의 구조체가 512의 약수가 되도록 만들기 위해서 메모리 크기만 차지하는 패딩을 붙여준다. fs.c와 mkfs.c에서 addr을 참조하는 함수들을 적당히 수정한다.

### 2-1. 소스코드별 함수 개요

#### 2-1-1. sysproc.c

ssualloc, getvp, getpp 시스템 콜을 구현한 파일

##### 2-1-1-1. void* ssualloc(int);

매개변수로 주어진 값만큼의 가상 주소 공간을 할당한다.

##### 2-1-1-2. int getvp(); int getpp()

할당된 가상, 물리 주소 공간의 페이지 수를 반환한다.

#### 2-1-2. trap.c

페이지 폴트를 처리하는 트랩 핸들러 파일

##### 2-1-2-1. void trap(struct trapframe *tf)

PGFLT가 발생하면 해당 공간에 대한 물리 메모리를 할당한다.

### 2-2. 함수 호출 그래프

#### 2-2-1. ssualloc_test

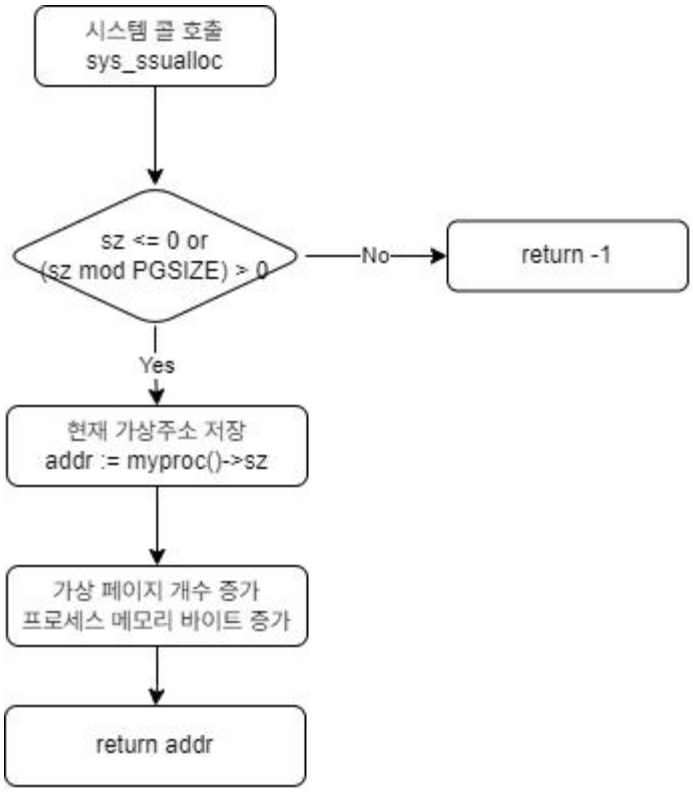
main (ssualloc_test.c) -> ssualloc (system call) -> access memory (illegal access) -> <memory fault> -> trap -> create physical address space & mapping (kernel) -> access memory (legal access / user) -> exit

#### 2-2-2. ssufs_test

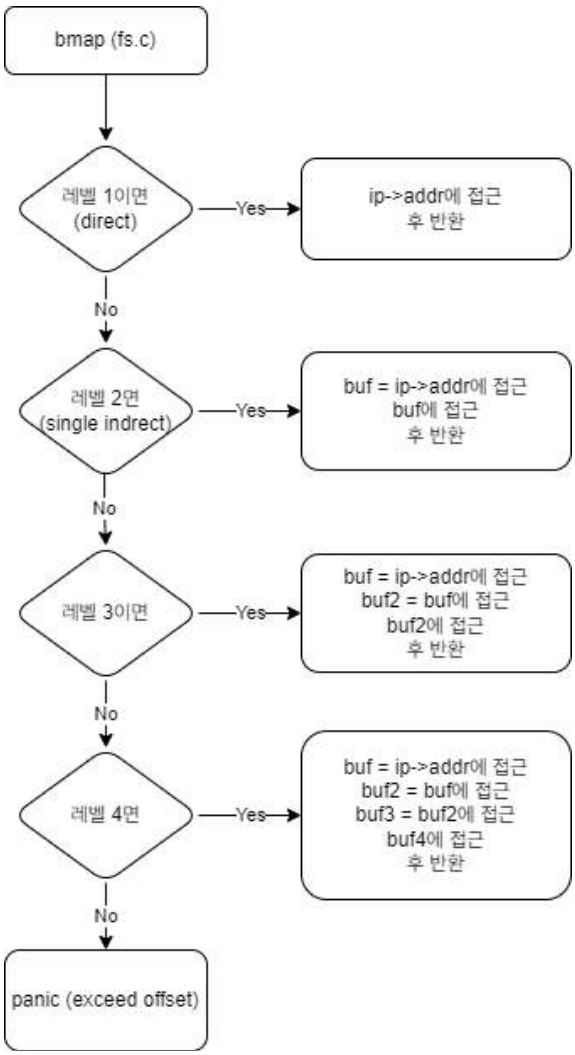
main (ssufs_test.c) -> write (system call) -> bmap (fs.c) -> ... -> read (system call) -> bmap (fs.c) -> ... -> exit

2-3. 순서도

2-3-1. ssualloc 가상 주소 공간 할당




2-3-3. fs.c 순서도



3. 실행 결과

3-1. ssualloc_test

```
$ ssualloc_test
Start: memory usages: virtual pages: 3, physical pages: 3
ssualloc() usage: argument wrong...
ssualloc() usage: argument wrong...
After allocate one virtual page: virtual pages: 4, physical pages: 3
After access one virtual page: virtual pages: 4, physical pages: 4
After allocate three virtual pages: virtual pages: 7, physical pages: 4
After access of first virtual page: virtual pages: 7, physical pages: 5
After access of third virtual page: virtual pages: 7, physical pages: 6
After access of second virtual page: virtual pages: 7, physical pages: 7
$
```

3-2. ssufs_test

+) 추가로 다음의 코드를 추가하여 데이터 저장도 정상적으로 동작함을 확인했다.

```
/* write */              for (int j =0; j < BSIZE; j++) buf[j] = (i + j) % p;
/* read & verify */      if (buf[j] != (i + j) % p) _error(...);
```

50000 블럭의 파일(약 24MB)을 작성하고 읽는데 대략 2~3분 정도의 시간이 걸린다.

```
$ ssufs_test
### test1 start
create and write 5 blocks...    ok
close file descriptor...        ok
open and read file...           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...           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...           ok
unlink file3...                 ok
open file3 again...             failed
### test3 passed...

### test4 start
create and write 50000 blocks...        ok
close file descriptor...        ok
open and read file...           ok
unlink file4...                 ok
open file4 again...             failed
### test4 passed...

$
```

4. 소스코드

4-1. defs.h (174번째 줄에 추가)

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

4-2. exec.c (exec 함수, 40번째 줄에 추가)

```
...
  if((pgdir = setupkvm()) == 0)
    goto bad;

  myproc()->virtual_cnt = 0;
  myproc()->physical_cnt = 0;

  // Load program into memory.
  sz = 0;
...
```

4-3. file.h (25번째 줄, 변경)

```
...
uint addrs[LVL_SUM+1];
...
```

4-4. fs.c

```
...

//PAGEBREAK!
// Inode content
//
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;
  uint relative = 0;

  if(bn < LVL_1){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
```

```
    return addr;
  }
  bn -= LVL_1;
  relative += LVL_1;

  if(bn < LVL_2 * NINDIRECT){
    // Load indirect block, allocating if necessary.
    uint w = bn / NINDIRECT + relative;
    uint x = bn % NINDIRECT;
    if((addr = ip->addrs[w]) == 0)
      ip->addrs[w] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[x]) == 0){
      a[x] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
  bn -= LVL_2 * NINDIRECT;
  relative += LVL_2;

  if(bn < LVL_3 * NINDIRECT * NINDIRECT){
    uint w = bn / (NINDIRECT * NINDIRECT) + relative;
    uint x = bn / NINDIRECT % NINDIRECT;
    uint y = bn % NINDIRECT;
    if((addr = ip->addrs[w]) == 0)
      ip->addrs[w] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[x]) == 0) {
      a[x] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[y]) == 0){
      a[y] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
  bn -= LVL_3 * NINDIRECT * NINDIRECT;
  relative += LVL_3;
```

```
  if(bn < LVL_4 * NINDIRECT * NINDIRECT * NINDIRECT){
    uint w = bn / (NINDIRECT * NINDIRECT * NINDIRECT) + relative;
    uint x = bn / (NINDIRECT * NINDIRECT) % NINDIRECT;
    uint y = bn / NINDIRECT % NINDIRECT;
    uint z = bn % NINDIRECT;
    if((addr = ip->addrs[w]) == 0)
      ip->addrs[w] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[x]) == 0) {
      a[x] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[y]) == 0) {
      a[y] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[z]) == 0){
      a[z] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}

static void itrunc_recursive(struct inode *ip, int recursive_step, uint *addr, int root) {
  if (recursive_step == 2) begin_op();
  {
    if (!*addr) goto bye;
    if (recursive_step == 0) {
      bfree(ip->dev, *addr);
      if (root) *addr = 0;
      goto bye;
    }
//     if (recursive_step == 3) cprintf("%d %d %d\n", recursive_step, *addr, root);
    if (recursive_step > 2) begin_op();
    struct buf *bp = bread(ip->dev, *addr);
```

```
      uint *a = (uint *) bp->data;
      uint arr[BSIZE];
      for (int i = 0; i < BSIZE; i++) arr[i] = a[i];
      brelse(bp);
      if (recursive_step > 2) end_op();
      for (int i = 0; i < NINDIRECT; i++) {
        itrunc_recursive(ip, recursive_step - 1, &arr[i], 0);
      }
//      if (recursive_step == 3) cprintf("%d %d %d\n", recursive_step, *addr, root);
      if (recursive_step > 2) begin_op();
//      if (recursive_step == 3) cprintf("bef free %d %d %d\n", recursive_step, *addr, root);
      bfree(ip->dev, *addr);
//      if (recursive_step == 3) cprintf("aftr free %d %d %d\n", recursive_step, *addr, root);
      if (recursive_step > 2) end_op();
//      if (recursive_step == 3) cprintf("%d %d %d\n", recursive_step, *addr, root);
      if (root) *addr = 0;
    }
    bye:;
    if (recursive_step == 2) end_op();
}

// Truncate inode (discard contents).
// Only called when the inode has no links
// to it (no directory entries referring to it)
// and has no in-memory reference to it (is
// not an open file or current directory).
void
itrunc(struct inode *ip)
{
    end_op();
//    cprintf("enter itrunc\n");
    for (int i = 0; i < LVL_SUM; i++) {
      int depth;
      if (i < LVL_1) depth = 0;
      else if (i < LVL_1 + LVL_2) depth = 1;
      else if (i < LVL_1 + LVL_2 + LVL_3) depth = 2;
      else depth = 3;
      if (depth < 2) begin_op();
      itrunc_recursive(ip, depth, &ip->addrs[i], 1);
      if (depth < 2) end_op();
//      cprintf("run itrunc : %d\n", i);
    }

    begin_op();
//    cprintf("exit1 itrunc\n");
    ip->size = 0;
    iupdate(ip);
```

```
//   cprintf("exit2 itrunc\n");
}

// Copy stat information from inode.
// Caller must hold ip->lock.
void
stati(struct inode *ip, struct stat *st)
...
```

4-5. fs.h

```
...
struct superblock {
  uint size;          // Size of file system image (blocks)
  uint nblocks;       // Number of data blocks
  uint ninodes;       // Number of inodes.
  uint nlog;          // Number of log blocks
  uint logstart;      // Block number of first log block
  uint inodestart;    // Block number of first inode block
  uint bmapstart;     // Block number of first free map block
};

#define LVL_1 6
#define LVL_2 4
#define LVL_3 2
#define LVL_4 1
#define LVL_SUM (LVL_1 + LVL_2 + LVL_3 + LVL_4)
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (LVL_1 + LVL_2 * NINDIRECT + LVL_3 * NINDIRECT * NINDIRECT + LVL_4 *
NINDIRECT * NINDIRECT * NINDIRECT)

// On-disk inode structure
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[LVL_SUM];   // Data block addresses
};

// Inodes per block.
#define IPB           (BSIZE / sizeof(struct dinode))
...
```

4-6. Makefile

```
...
UPROGS=\
```

```
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _ssualloc_test\
        _ssufs_test\
...
```

4-7. mkfs.c

```
...
  winode(inum, &din);
  return inum;
}

void
balloc(int used)
{
  uchar buf[BSIZE];
  int i;

  printf("balloc: first %d blocks have been allocated\n", used);
  for (int a = 0; a < used / (BSIZE * 8) + (used % (BSIZE * 8) > 0); a++) {
    bzero(buf, BSIZE);
    for(i = 0; i < BSIZE * 8; i++){
      if (i + a * BSIZE * 8 >= used) break;
      buf[i/8] = buf[i/8] | (0x1 << (i%8));
    }
    printf("balloc: write bitmap block at sector %d\n", sb.bmapstart + a);
    wsect(sb.bmapstart + a, buf);
  }
}

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

void
iappend(uint inum, void *xp, int n)
```

```
{
  char *p = (char*)xp;
  uint fbn, off, n1;
  struct dinode din;
  char buf[BSIZE];
  uint indirect[NINDIRECT];
  uint x;

  rinode(inum, &din);
  off = xint(din.size);
  // printf("append inum %d at off %d sz %d\n", inum, off, n);
  while(n > 0){
    fbn = off / BSIZE;
    uint idx = fbn;
    uint relative = 0;
//     printf("%d\n", fbn);

    assert(idx < MAXFILE);
    if(idx < LVL_1){
      if(xint(din.addrs[idx]) == 0){
        din.addrs[idx] = xint(freeblock++);
      }
      x = xint(din.addrs[idx]);
    }
    idx -= LVL_1;
    relative += LVL_1;

    if (0 <= idx && idx < LVL_2 * NINDIRECT) {
      uint a = idx / NINDIRECT + relative;
      uint b = idx % NINDIRECT;
      if(xint(din.addrs[a]) == 0){
        din.addrs[a] = xint(freeblock++);
      }
      rsect(xint(din.addrs[a]), (char*)indirect);
      if(indirect[b] == 0){
        indirect[b] = xint(freeblock++);
        wsect(xint(din.addrs[a]), (char*)indirect);
      }
      x = xint(indirect[b]);
    }
    idx -= LVL_2 * NINDIRECT;
    relative += LVL_2;

    if (0 <= idx && idx < LVL_3 * NINDIRECT * NINDIRECT) {
      uint a = idx / (NINDIRECT * NINDIRECT) + relative;
      uint b = idx / NINDIRECT % NINDIRECT;
      uint c = idx % NINDIRECT;
```

```
    if(xint(din.addrs[a]) == 0){
      din.addrs[a] = xint(freeblock++);
    }
    rsect(xint(din.addrs[a]), (char*)indirect);
    if(indirect[b] == 0){
      indirect[b] = xint(freeblock++);
      wsect(xint(din.addrs[a]), (char*)indirect);
    }
    uint tmp = indirect[b];
    rsect(xint(tmp), (char*)indirect);
    if(indirect[c] == 0){
      indirect[c] = xint(freeblock++);
      wsect(xint(tmp), (char*)indirect);
    }
    x = xint(indirect[c]);
  }
  idx -= LVL_3 * NINDIRECT * NINDIRECT;
  relative += LVL_3;

  if (0 <= idx && idx < LVL_4 * NINDIRECT * NINDIRECT * NINDIRECT) {
    uint a = idx / (NINDIRECT * NINDIRECT * NINDIRECT) + relative;
    uint b = idx / (NINDIRECT * NINDIRECT) % NINDIRECT;
    uint c = idx / NINDIRECT % NINDIRECT;
    uint d = idx % NINDIRECT;
    if(xint(din.addrs[a]) == 0){
      din.addrs[a] = xint(freeblock++);
    }
    rsect(xint(din.addrs[a]), (char*)indirect);
    if(indirect[b] == 0){
      indirect[b] = xint(freeblock++);
      wsect(xint(din.addrs[a]), (char*)indirect);
    }
    uint tmp = indirect[b];
    rsect(xint(tmp), (char*)indirect);
    if(indirect[c] == 0){
      indirect[c] = xint(freeblock++);
      wsect(xint(tmp), (char*)indirect);
    }
    tmp = indirect[c];
    rsect(xint(tmp), (char*)indirect);
    if(indirect[d] == 0){
      indirect[d] = xint(freeblock++);
      wsect(xint(tmp), (char*)indirect);
    }
    x = xint(indirect[d]);
  }
```

```
    n1 = min(n, (fbn + 1) * BSIZE - off);
    rsect(x, buf);
    bcopy(p, buf + off - (fbn * BSIZE), n1);
    wsect(x, buf);
    n -= n1;
    off += n1;
    p += n1;
  }
  din.size = xint(off);
  winode(inum, &din);
}
```

4-8. param.h

```
...
#define NBUF          (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE        2500000  // size of file system in blocks
...
```

4-9. proc.c (exec 함수, 192번째 줄 추가)

```
...
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  np->virtual_cnt = 0;
  np->physical_cnt = 0;

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
...
```

4-10. proc.h (구조체 proc에 필드 추가)

```
...
  char name[16];               // Process name (debugging)

  int virtual_cnt;
  int physical_cnt;
};
...
```

4-11. ssualloc_test.c

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

```
int main(void)
{
  int ret;
  printf(1, "Start: memory usages: virtual pages: %d, physical pages: %d\n", getvp(), getpp());
  ret = ssualloc(-1234);

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

  ret = ssualloc(1234);

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

  ret = ssualloc(4096);

  if(ret < 0 )
    printf(1, "ssualloc(): failed...\n");
  else {
    printf(1, "After allocate one virtual page: virtual pages: %d, physical pages: %d\n", getvp(),
getpp());
    char *addr = (char *) ret;

    addr[0] = 'I';
    printf(1, "After access one virtual page: virtual pages: %d, physical pages: %d\n", getvp(),
getpp());
  }

  ret = ssualloc(12288);

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

    addr[0] = 'a';
    printf(1, "After access of first virtual page: virtual pages: %d, physical pages: %d\n", getvp(),
getpp());
    addr[10000] = 'b';
    printf(1, "After access of third virtual page: virtual pages: %d, physical pages: %d\n", getvp(),
```

```
getpp());
      addr[8000] = 'c';
      printf(1, "After access of second virtual page: virtual pages: %d, physical pages: %d\n",
getvp(), getpp());
   }

   exit();
}
```

4-12. ssufs_test.c

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

#define BSIZE 512

char buf[BSIZE];

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

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

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

   filename[4] = (ntest % 10) + '0';

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

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

   for (i = 0; i < blocks; i++) {
      ret = write(fd, buf, BSIZE);
      if (ret < 0) break;
   }
   if (ret < 0)
```

```
    _error("File write error\n");
  else
    _success();

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

  if (close(fd) < 0)
    _error("File close error\n");
  else
    _success();

  printf(1, "open and read file...\t\t");
  fd = open(filename, O_RDONLY);

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

  for (i = 0; i < blocks; i++) {
    ret = read(fd, buf, BSIZE);
    if (ret < 0) break;
  }
  if (ret < 0)
    _error("File read error\n");

  if (close(fd) < 0)
    _error("File close error\n");
  else
    _success();

  printf(1, "unlink %s...\t\t\t", filename);

  if (unlink(filename) < 0)
    _error("File unlink error\n");
  else
    _success();

  printf(1, "open %s again...\t\t", filename);
  fd = open(filename, O_RDONLY);

  if (fd < 0)
    printf(1, "failed\n");
  else
...     printf(1, "this statement cannot be runned\n");

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

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

  test(1, 5);
  test(2, 500);
  test(3, 5000);
  test(4, 50000);
  exit();
}
```

4-13. syscall.c

```
...

extern int sys_uptime(void);
extern int sys_ssualloc(void);
extern int sys_getvp(void);
extern int sys_getpp(void);

static int (*syscalls[])(void) = {

...

[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_ssualloc]   sys_ssualloc,
[SYS_getvp]   sys_getvp,
[SYS_getpp]   sys_getpp,
};
...
```

4-14. syscall.h

```
...
#define SYS_close  21
#define SYS_ssualloc  22
#define SYS_getvp  23
#define SYS_getpp  24
```

4-15. sysproc.c

```
...
  return xticks;
}

int sys_ssualloc(void) {
```

```
  int sz;
  if (argint(0, &sz) < 0) return -1;
  if (sz <= 0) return -1;
  if (sz % PGSIZE) return -1;
  int addr = myproc()->sz;
  myproc()->virtual_cnt += sz / PGSIZE;
  myproc()->sz += sz;
  return addr;
}

int sys_getvp(void) {
  return myproc()->virtual_cnt;
}

int sys_getpp(void) {
  return myproc()->physical_cnt;
}
```

4-16. trap.c (83번째 줄 default 내부에)

```
...
  //PAGEBREAK: 13
  default:
    if(tf->trapno == T_PGFLT){
      uint va = rcr2();
      if (myproc()->sz > va) {
        char *pa = kalloc();
//        memset(pa, 0, PGSIZE);
        mappages(myproc()->pgdir, (char*)PGROUNDDOWN(va), PGSIZE, V2P(pa), PTE_W|PTE_U);
        myproc()->physical_cnt++;
        break;
      }
    }
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
...
```

4-17. user.h

```
...
int uptime(void);
int ssualloc(int);
int getvp();
int getpp();

// ulib.c
...
```

4-18. usys.S

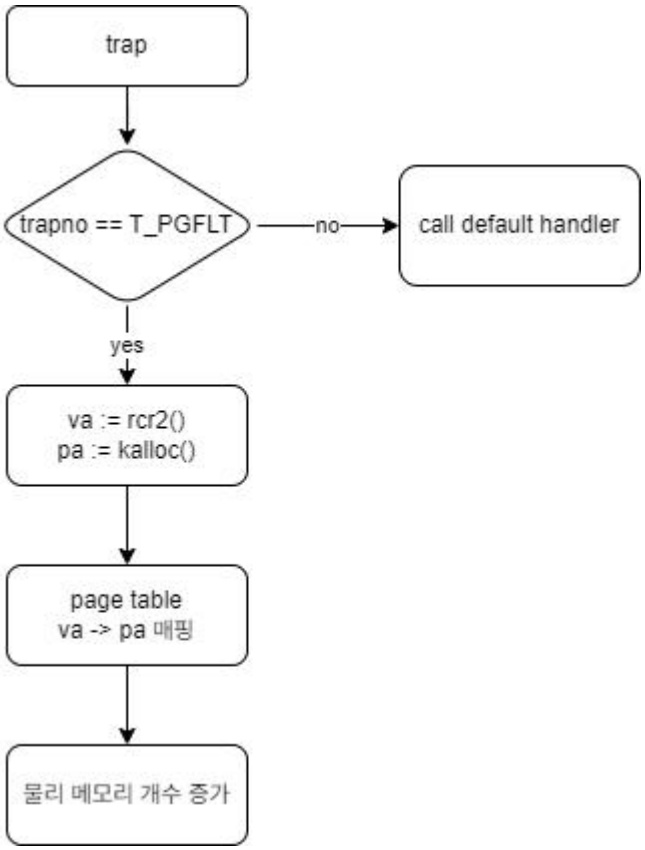
```
SYSCALL(uptime)
SYSCALL(ssualloc)
SYSCALL(getvp)
SYSCALL(getpp)
...
```

4-18. vm.c (60번째 줄, 함수 헤더에서 static 제거)

```
...
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
...
```

5. ssualloc 관련 trap

* trap 함수 순서도




처리 과정 (자연어)
- trap 번호로 T_PGFLT가 오면 메모리 폴트가 발생했다는 것이고, 이는 프로세스가 접근한 가상 메모리에 대응하는 물리 메모리 정보가 페이지 테이블에 없어 발생하는 트랩이다.
- 해당 trap에 대해서 rcr2()를 호출하면 접근하려는 가상 주소를 알 수 있다.
- kalloc()은 물리 공간에 메모리를 할당받는 함수이다.
- mappages 함수를 사용해서 가상 주소와 방금 할당받은 물리 주소를 매핑한다.
- 물리 페이지 개수가 하나 증가시킨다.