

1. 개요

본 설계 과제는 xv6 메모리 및 파일시스템 구조 이해 및 응용을 목표로 하고 있다.

구체적으로 4-1)에서는 xv6 운영체제에 새로운 메모리 할당 기능인 'ssualloc()', 할당된 가상 및 물리 페이지의 수를 확인 할 수 있는 'getvp()', 'getpp()' 함수를 구현하여 테스트를 진행한다.

4-2)에서는 xv6 파일 시스템의 구조를 확장하여 multi-level 파일 시스템을 구현하는 것을 목표로 한다. 기존의 xv6 파일 시스템은 12개의 direct 블록과 1개의 indirect 블록을 사용하지만, 본 과제에서는 대용량 파일 생성이 가능하게 파일 시스템의 구조를 확장한다.

설계과제 4-1) SSU Memory Allocation:

가상 메모리 할당

ssualloc() 시스템 콜

1. 하나의 매개변수를 입력받는다.

-매개변수는 할당할 가상 메모리의 크기이다.

-매개변수는 양수이며, 페이지의 크기의 배수이다.

-유효하지 않은 값을 입력받았을 시, -1을 반환한다.

-유효한 값을 입력받고, 가상 메모리를 성공적으로 할당했을 때 할당한 메모리의 주소를 반환한다.

2. 가상 메모리는 할당하지만, 물리 메모리는 할당하지 않는다.

-할당된 가상 메모리에 사용자가 접근할 때 해당 가상 메모리에 상응하는 물리 메모리가 존재하지 않을 경우 특정 trap이 발생하며, 이때 물리 메모리를 페이지 단위로 할당한다.

- ssualloc()을 통해 하나 이상의 가상 메모리 페이지가 할당된 경우 접근된 페이지에 대해서만 물리 메모리 페이지를 할당해야 한다.

getvp(), getpp()

1. 매개변수를 입력 받지 않는다.

getvp() : 호출한 프로세스의 가상 메모리 페이지 개수를 반환한다.

getpp() : 호출한 프로세스의 물리 메모리 페이지 개수를 반환한다.

설계과제 4-2) Multi-level File System:

본 과제에서는 기존의 xv6 파일 시스템의 구조를 수정하여 6개의 direct 블록, 4개의 indirect 블록, 2개의 2-level indirect 블록, 그리고 1개의 3-level indirect 블록을 사용할 수 있도록 구현을 한다.

따라서, $2,130,438(6+128*4+128*128*2+128*128*128)$ 개의 데이터 블록을 저장 공간으로 사용할 수 있으며 약 1GB의 파일 크기를 지원한다.

대용량 파일 생성을 위해 param.h의 FSSIZE를 기존 1000에서 2500000으로 수정한다.

ssufs_test를 실행하여 테스트를 진행하여 구현이 올바르게 되었는지 검증해본다.

2. 상세설계

-구체적 수정 코드는 4. 소스코드에서 빨간색으로 표시하여 첨부했습니다.

4-1)

ssualloc() 함수: 지정된 크기의 메모리를 할당하고, 할당된 메모리의 시작 주소를 반환한다. 할당 크기는 페이지 크기(4096 바이트)의 배수인 양수여야 한다. 할당 성공 시 시작 주소를, 실패하면 -1을 반환하도록 설계하였다.

1) ssualloc() (proc.c)

구현 내용

입력된 size가 0 이하이거나 페이지 크기(PGSIZE)의 배수가 아니면, 잘못된 요청으로 간주하고 -1을 반환한다.

새로운 가상 메모리 크기(new_sz)를 현재 프로세스 크기(curproc->sz)에 더해 계산한다.

새 크기가 현재 크기보다 크면, 가상 메모리 크기(curproc->sz)를 갱신한다.

할당된 메모리의 시작 주소를 반환한다. 이 주소는 기존 가상 메모리 크기와 새로 할당된 크기의 차이를 이용하여 계산하도록 설계하였다.

2) getvp() (proc.c)

구현 내용

현재 프로세스의 총 가상 메모리 크기(curproc->sz)를 페이지 크기(PGSIZE)로 나누어, 총 가상 페이지 수를 계산한다.

3) 물리 페이지 수 반환 함수 getpp

구현 내용

curproc->sz 크기만큼 반복하며, 각 가상 주소에 대응하는 페이지 테이블 항목(pte)을 walkpgdir_fp를 통해 가져온다.

해당 pte가 존재하며, 'Present' 비트(PTE_P)가 활성화되어 있다면, 이는 물리 메모리에 할당된 페이지를 의미한다. 즉, 페이지 테이블을 순회하며 PTE_P 플래그가 설정된 페이지 엔트리 수를 계산한다.

4) trap 처리에서 페이지 폴트 처리(trap.c-trap())

목적 : 메모리 페이지 폴트가 발생 시 적절하게 처리하기 위함.

페이지 폴트 처리 구현 핵심 로직

페이지 폴트(T_PGFLT)를 처리하는 switch문을 추가한다.

현재 프로세스(curproc)와 폴트 주소(fault_addr)를 얻는다.

fault_addr이 프로세스의 크기(sz) 이내일 경우, 새로운 물리 페이지를 할당(kalloc())한다.

새로 할당된 페이지가 비어 있음을 보장하기 위해 memset으로 초기화한다.

mappages_wrapper를 사용하여 가상 주소(fault_addr)를 물리 주소(mem)에 매핑한다.

성공적으로 매핑된 경우, 할당된 페이지 크기(sz_allocated)를 페이지 크기(PGSIZE)만큼 증가시키는 방식으로 갱신한다.

fault_addr이 프로세스 크기를 초과하면, 이는 불법 접근으로 간주하고 프로세스를 종료한다.(curproc->killed = 1).

trap 처리: 페이지 폴트는 프로세스가 아직 물리적으로 매핑되지 않은 가상 주소에 접근할 때 발생한다. trap 함수 내에서, 페이지 폴트 발생하면 kalloc()을 호출하여 물리 페이지를 할당하고, mappages()를 사용하여 가상 주소와 물리 주소를 매핑한다. 이 과정에서 PGROUNDOWN과 PGROUNDUP을 사용하여 주소를 페이지 경계로 정렬한다. ssualloc()에 의해 할당된 페이지는 초기에 물리적으로 매핑되지 않으므로, 첫 접근 시에 페이지 폴트 처리를 통해 실제 메모리가 할당되게 된다.

5) 가상 메모리 관리 변경 (vm.c)

목적: mappages 함수를 사용하여 가상 주소와 물리 주소를 매핑하고 관리하기 위함.

수정 사항:

mappages_wrapper() 래퍼 함수 추가 : mappages 함수를 간단하게 호출하기 위함.

해당 래퍼 함수는 mappages와 동일한 인자를 받으며, 내부적으로 mappages를 호출한다. 이를 통해 trap 함수에서 페이지 매핑을 더 간결하게 처리할 수 있다.

6) 기타

- 시스템 콜 추가(ssualloc, getvp, getpp)

각 시스템 콜의 프로토 타입 추가(defs.h)

각 시스템 콜의 매크로를 통해 정의(usys.S)

각 시스템 콜 고유 번호 할당(syscall.h)

사용자 인터페이스 프로토타입 추가(user.h)

시스템 콜 테이블 수정, 시스템 콜 함수 매핑(syscall.c)

- 함수 포인터 정의 (walkpgdir_fp_t)

walkpgdir_fp_t는 페이지 디렉토리를 walk()에 대한 함수 포인터 타입을 정의한다. 이는 페이지 테이블 항목을 가져오는 데 사용된다.

extern walkpgdir_fp_t walkpgdir_fp;로 외부에서 정의된 walkpgdir_fp 함수 포인터를 선언한다. 이를 통해 코드의 다른 코드에서도 해당 포인터를 참조하고 할당할 수 있게 한다.

4-2)

struct inode의 addrs 배열을 수정(file.h)

원래는 NDIRECT+1 크기였지만, 이를 NDIRECT+7로 변경하여 추가적인 indirect 블록들을 관리할 수 있게 한다.(direct 블록의 수를 줄이고, 추가적인 indirect 블록들을 관리하기 위함)

NDIRECT를 12에서 6으로 변경 (fs.h)

NINODEBLOCKS, NNINODEBLOCKS를 추가하여 각각 1-level과 2-level indirect 블록의 수를 정의한다.

MAXFILE 계산을 수정하여 새로운 블록 구조를 반영한다.

(fs.c)

Direct 블록, 1-level, 2-level, 그리고 3-level indirect 블록에 대한 처리를 추가한다.

itrunc 함수를 수정하여 multi-level indirect 블록 구조에 따라 inode를 올바르게 자를(truncate) 수 있도록 한다.

NDIRECT 변경: Direct 블록의 수를 줄임으로써, 더 많은 indirect 블록들을 inode 구조 내에 포함시킬 수 있다.

bmap 함수의 수정: 파일의 논리적 블록 번호를 파일 시스템의 실제 블록 번호로 매핑을 해주기 위해 새로운 파일 시스템 구조에 맞게 수정한다.

i) 파일 시스템 구조 변경

Direct Blocks: 기존의 12개에서 6개로 줄임.

1-Level Indirect Blocks: 새롭게 4개 추가.

2-Level Indirect Blocks: 새롭게 2개 추가.

3-Level Indirect Blocks: 새롭게 1개 추가.

ii) bmap 함수 수정

Direct Blocks 처리: bn이 NDIRECT 미만일 때 직접 참조.

1-Level Indirect Blocks 처리: bn에서 NDIRECT를 뺀 값이 NINODEBLOCKS * NINDIRECT 미만일 때 처리.

2-Level과 3-Level Indirect Blocks 처리: 상응하는 계산을 통해 각 레벨의 indirect block을 처리.

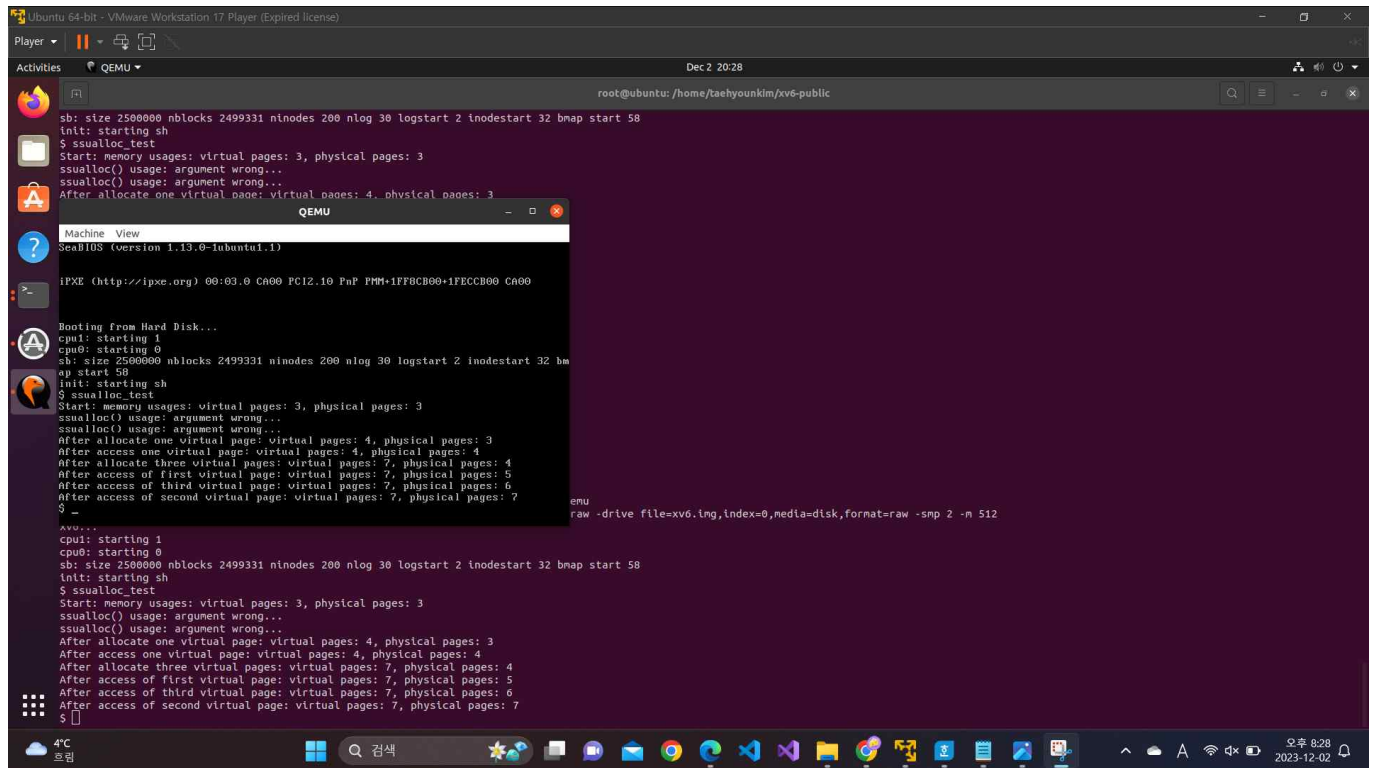
iii) Makefile 수정

파일 시스템의 변경 사항을 반영하기 위해 Makefile을 수정함.

FSSIZE는 xv6 파일 시스템에서 사용 가능한 전체 블록의 수를 나타낸다. 기존에 설정된 FSSIZE 값인 1000은 상대적으로 작은 용량의 파일 시스템을 의미하기 때문에 대용량 파일이나 데이터를 저장하고 관리하는 상황에 제한적이다. 본 과제에서 구현한 multi-level 파일 시스템은 기존에 비해 훨씬 더 많은 데이터 블록을 관리할 수 있도록 설계되었으므로 더 많은 데이터 블록을 수용할 수 있게 FSSIZE를 2500000으로 변경해준다.

3. 결과

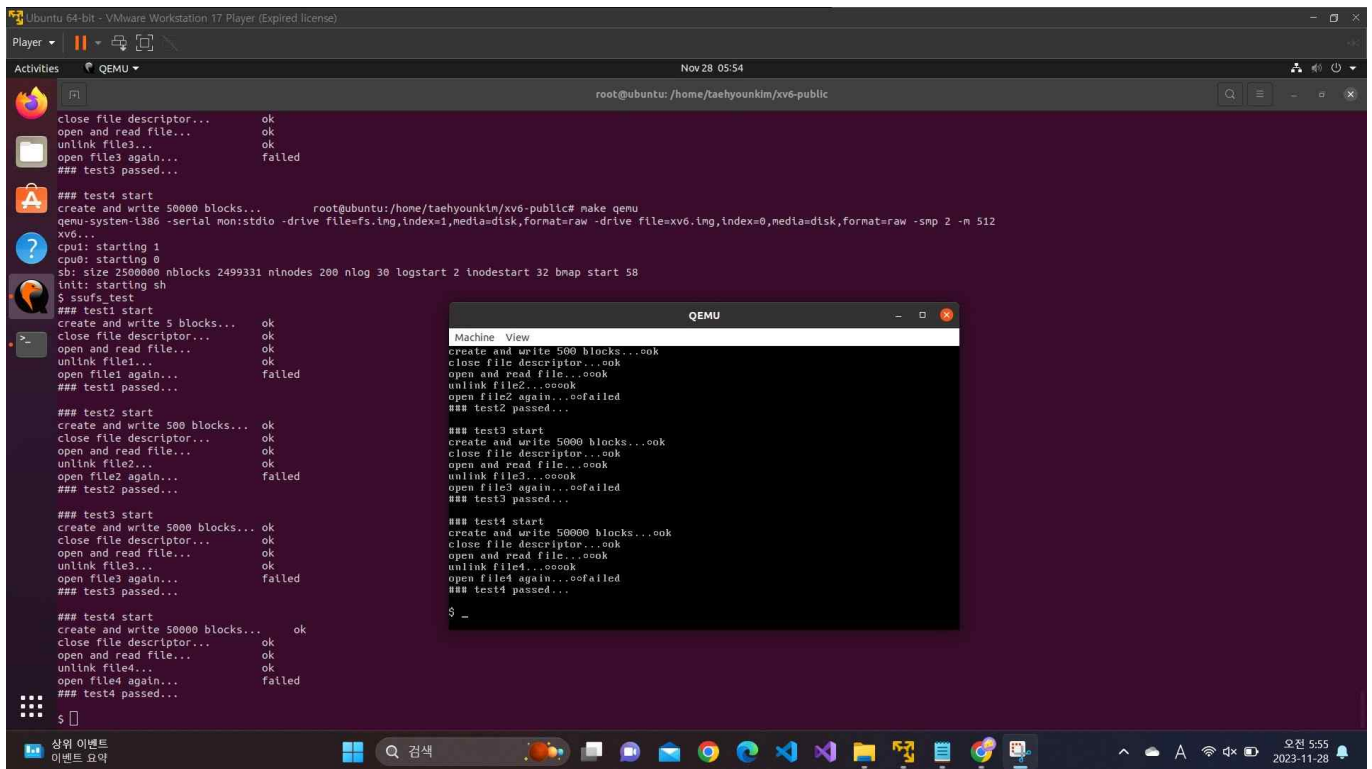
1) ssualloc_test 실행 결과



```
sb: size 2500000 nblocks 2499331 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ssualloc_test
Start: memory usages: virtual pages: 3, physical pages: 3
ssuallloc() usage: argument wrong...
ssuallloc() usage: argument wrong...
After allocate one virtual page: virtual pages: 4, physical pages: 3
Machine View
SeaBIOS (version 1.13.0-ubuntu1.1)
IPXE (http://ipxe.org) 00:03:0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 2500000 nblocks 2499331 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ssualloc_test
Start: memory usages: virtual pages: 3, physical pages: 3
ssuallloc() usage: argument wrong...
ssuallloc() 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
$ -
emu
raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -n 512
$ -
cpu1: starting 1
cpu0: starting 0
sb: size 2500000 nblocks 2499331 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ssualloc_test
Start: memory usages: virtual pages: 3, physical pages: 3
ssuallloc() usage: argument wrong...
ssuallloc() 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
$ -
```

예시와 동일하게 정상적으로 작동하였음.

2) ssufs_test 실행결과



```
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...

$ _
```

```
Machine View
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...

$ _
```

xv6 파일 시스템에서 다양한 크기의 파일을 생성, 읽기, 쓰기, 삭제하는 각각의 테스트는 성공적으로 수행되었다.

특이 사항(-제출한 파일의 Makefile의 코드는 windows에서 작업한 코드입니다.)
windows에서 가상머신을 통해 작업을 하고 실행을 하였을 때,
test 1,2의 진행이 1분, test 3 10분, test 4 20분 대략 30분의 시간이 소요되었다.

윈도우가 아닌 맥북에서 환경에 맞게 makefile을 수정해 준 뒤 ssufs_test를 실행해본 결과 컴파일부터 프로그램 종료까지 3분이내의 시간이 소요되는 것을 확인하였습니다.

4. 수정한 소스코드(Makefile,defs.h,file.h,fs.c, fs.h, mmu.h,param.h, proc.c, proc.h, syscall.c,syscall.h, sysproc.c, trap.c, types.h, user.h, usys.S, vm.c)

Makefile

```
...
UPROGS=\
    _usertests\
    _wc\
    _zombie\
    _ssualloc_test\
    _ssufs_test\

188 fs.img: mkfs README $(UPROGS)
189     ./mkfs fs.img README $(UPROGS)
...
EXTRA=\
255     printf.c umalloc.c\
256     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
257     .gdbinit.tmpl gdbutil\
258     ssualloc_test.c ssufs_test.c\

dist:
    rm -rf dist
...
```

defs.h

```
1 #include "mmu.h"
2
3 struct buf;
4 struct context;
5 struct file;
...
158 int      fetchstr(uint, char**);
159 void      syscall(void);

161 //sysproc.c
162 int ssualloc(int);
163 int getvp(void);
164 int getpp(void);
165
166 // timer.c
167 void      timerinit(void);
...
192 void      switchkvm(void);
```

```

193 int          copyout(pde_t*, uint, void*, uint);
194 void          clearpteu(pde_t *pgdir, char *uva);
195 typedef pte_t *(*walkpgdir_fp_t)(pde_t *pgdir, const void *va, int alloc);
196 extern walkpgdir_fp_t walkpgdir_fp;
197 int mappages_wrapper(pde_t *pgdir, void *va, uint size, uint pa, int perm);
198
199 // number of elements in fixed-size array
200 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))

```

file.h

```
...struct inode {
```

```

22  short minor;
23  short nlink;
24  uint size;
25  uint addrs[NDIRECT+7];
};

```

fs.c

...

```

373 static uint
374 bmap(struct inode *ip, uint bn)
375 {
376     uint addr, *a, *b, *c;
377     struct buf *bp;

```

```

    // Direct blocks
    if (bn < NDIRECT) {
        if ((addr = ip->addrs[bn]) == 0)
            ip->addrs[bn] = addr = balloc(ip->dev);
        return addr;
    }
    bn -= NDIRECT;

```

```

    // 1-level indirect blocks
    if (bn < NINODEBLOCKS * NINDIRECT) {
        int indirIndex = NDIRECT + bn / NINDIRECT;
        if ((addr = ip->addrs[indirIndex]) == 0)
            ip->addrs[indirIndex] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint *)bp->data;
        if ((addr = a[bn % NINDIRECT]) == 0) {
            a[bn % NINDIRECT] = addr = balloc(ip->dev);
            log_write(bp);

```



```

    }
    brelse(bp);
    return addr;
}

bn -= NINODEBLOCKS * NINDIRECT;

// 2-level indirect blocks
if (bn < NNINODEBLOCKS * NNINDIRECT) {
    uint blockIndex = NDIRECT + NINODEBLOCKS + bn / NNINDIRECT;
    if ((addr = ip->addrs[blockIndex]) == 0)
        ip->addrs[blockIndex] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    bn %= NNINDIRECT;

    if ((addr = a[bn / NINDIRECT]) == 0) {
        a[bn / NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    b = (uint *)bp->data;
    if ((addr = b[bn % NINDIRECT]) == 0) {
        b[bn % NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    return addr;
}
bn -= NNINODEBLOCKS * NNINDIRECT;

// 3-level indirect blocks
if (bn < NNNINDIRECT) {
    if ((addr = ip->addrs[NDIRECT + NINODEBLOCKS + NNINODEBLOCKS]) == 0)
        ip->addrs[NDIRECT + NINODEBLOCKS + NNINODEBLOCKS] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    bn /= NINDIRECT * NINDIRECT;

```

```

    if ((addr = a[bn]) == 0) {
        a[bn] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    struct buf *bp2 = bread(ip->dev, addr);
    b = (uint *)bp2->data;
    bn %= NINDIRECT * NINDIRECT;

    if ((addr = b[bn / NINDIRECT]) == 0) {
        b[bn / NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp2);
    }
    brelse(bp2);

    struct buf *bp3 = bread(ip->dev, addr);
    c = (uint *)bp3->data;
    if ((addr = c[bn % NINDIRECT]) == 0) {
        c[bn % NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp3);
    }
    brelse(bp3);
465     return addr;
466 }
467 panic("bmap: out of range");
468 }
static void
itrunc(struct inode *ip)
{
    int i, j, k;
    struct buf *bp, *bp2, *bp3;
    uint *a, *a2, *a3;

    // Free direct blocks
    for(i = 0; i < NDIRECT; i++){
        if(ip->addrs[i]){
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    // Free indirect blocks (four of them)
    for(i = 0; i < 4; i++){

```

```

if(ip->addrs[NDIRECT + i]){
    bp = bread(ip->dev, ip->addrs[NDIRECT + i]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
        if(a[j])
            bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + i]);
    ip->addrs[NDIRECT + i] = 0;
}
}

```

```

// Free doubly indirect blocks (two of them)
for(i = 0; i < 2; i++){
    if(ip->addrs[NDIRECT + 4 + i]){
        bp = bread(ip->dev, ip->addrs[NDIRECT + 4 + i]);
        a = (uint*)bp->data;
        for(j = 0; j < NINDIRECT; j++){
            if(a[j]){
                bp2 = bread(ip->dev, a[j]);
                a2 = (uint*)bp2->data;
                for(k = 0; k < NINDIRECT; k++){
                    if(a2[k])
                        bfree(ip->dev, a2[k]);
                }
                brelse(bp2);
                bfree(ip->dev, a[j]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 4 + i]);
        ip->addrs[NDIRECT + 4 + i] = 0;
    }
}

```

```

// Free the triply indirect block
if(ip->addrs[NDIRECT + 6]){
    bp = bread(ip->dev, ip->addrs[NDIRECT + 6]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
        if(a[i]){
            bp2 = bread(ip->dev, a[i]);
            a2 = (uint*)bp2->data;

```

```

        for(j = 0; j < NINDIRECT; j++){
            if(a2[j]){
                bp3 = bread(ip->dev, a2[j]);
                a3 = (uint*)bp3->data;
                for(k = 0; k < NINDIRECT; k++){
                    if(a3[k])
                        bfree(ip->dev, a3[k]);
                }
                brelse(bp3);
                bfree(ip->dev, a2[j]);
            }
        }
        brelse(bp2);
        bfree(ip->dev, a[i]);
    }
}

brelse(bp);
bfree(ip->dev, ip->addrs[NDIRECT + 6]);
ip->addrs[NDIRECT + 6] = 0;
}

ip->size = 0;
iupdate(ip);
}

```

fs.h

```

21  uint bmapstart;    // Block number of first free map block
22 };
23
24 #define NDIRECT 6
25 #define NINDIRECT (BSIZE / sizeof(uint))
26 #define NNINDIRECT (NINDIRECT * NINDIRECT)
27 #define NNNINDIRECT (NINDIRECT * NINDIRECT * NINDIRECT)
28 #define NINODEBLOCKS 4 // Number of 1-level indirect blocks
29 #define NNINODEBLOCKS 2 // Number of 2-level indirect blocks
30
31 #define MAXFILE (NDIRECT + NINODEBLOCKS * NINDIRECT + NNINODEBLOCKS * NNINDIRECT +
NNINDIRECT)
32
33 // On-disk inode structure
34 struct dinode {
...

```

```

37 short minor;           // Minor device number (T_DEV only)
38 short nlink;           // Number of links to inode in file system
39 uint size;             // Size of file (bytes)
40 uint addrs[NDIRECT+ NINODEBLOCKS + NNINODEBLOCKS + 1]; // Data block addresses (+1 =>
+7, since ndriect 11 => 6)
41 };
42
43 // Inodes per block.
...

```

mmu.h

```

1 #ifndef _MMU_H_
2 #define _MMU_H_
3// This file contains definitions for the
4 // x86 memory management unit (MMU).
5 #include "types.h"
6 // Eflags register
7 #define FL_IF           0x00000200      // Interrupt Enable
8
...
183 #endif
184 #endif // _MMU_H_

```

param.h

```

...
10 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
11 #define LOGSIZE      (MAXOPBLOCKS*3) // max data blocks in on-disk log
12 #define NBUF         (MAXOPBLOCKS*3) // size of disk block cache
13 #define FSSIZE       2500000 // size of file system in blocks

```

proc.c

```

1 #include "mmu.h"
2 #include "types.h"
3 #include "defs.h"
4 #include "param.h"
5 #include "memlayout.h"
6 #include "x86.h"
7 #include "proc.h"
8 #include "spinlock.h"
9
10
11 struct {
12  struct spinlock lock;
13  struct proc proc[NPROC];

```

...

```
22 static void wakeup1(void *chan);
```

```
25 void
```

```
26 pinit(void)
```

```
27 {
```

```
28     initlock(&ptable.lock, "ptable");
```

```
29 }
```

```
31 int ssualloc(int size) {
```

```
32     struct proc *curproc = myproc();
```

```
33
```

```
34     if (size <= 0 || size % PGSIZE != 0) {
```

```
35         return -1;
```

```
36     }
```

```
37
```

```
38     // Allocate virtual memory
```

```
39     uint new_sz = curproc->sz + size;
```

```
40     if (new_sz > curproc->sz) {
```

```
41         // Allocate virtual memory but don't map it to physical pages
```

```
42         curproc->sz = new_sz;
```

```
43
```

```
44         return new_sz - size; // Return the start address of allocated memory
```

```
45     }
```

```
46
```

```
47     return -1;
```

```
48 }
```

```
49
```

```
50 int getvp(void) {
```

```
51     struct proc *curproc = myproc();
```

```
52     return curproc->sz / PGSIZE; //Number of virtual pages
```

```
53 }
```

```
54
```

```
55 int getpp(void) {
```

```
56     struct proc *curproc = myproc();
```

```
57     int count = 0;
```

```
58     for (uint i = 0; i < curproc->sz; i += PGSIZE) {
```

```
59         pte_t *pte = walkpgdir_fp(curproc->pgdir, (void *) i, 0);
```

```
60         if (pte && (*pte & PTE_P)) {
```

```
61             count++;
```

```
62         }
```

```
63     }
```

```
64     return count; // Number of physical pages
```

```
65 }
```

```

66
67 // Must be called with interrupts disabled
68 int
69 cpuid() {
...

proc.h
...
struct proc {
...
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    char name[16];              // Process name (debugging)
    uint sz_allocated;          // Size of virtual memory allocated (bytes)
};
...

syscall.c
...
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_sualloc(void);
extern int sys_getvp(void);
extern int sys_getpp(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
...
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_sualloc] sys_sualloc,
[SYS_getvp]   sys_getvp,
[SYS_getpp]   sys_getpp,
};
...

syscall.h
...
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_sualloc 22

```

```
#define SYS_getvp 23
#define SYS_getpp 24
```

sysproc.c

```
...
int
sys_ssualloc(void) {
    int size;
    if(argint(0, &size) < 0)
        return -1;
    return ssualloc(size);
}
```

```
int
sys_getvp(void) {
    return getvp();
}
```

```
int
sys_getpp(void) {
    return getpp();
}
...
```

trap.c

```
...
trap(struct trapframe *tf){
...
    return;
}
...
switch(tf->trapno){
    case T_PGFLT:
    {
        struct proc *curproc = myproc();
        uint fault_addr = rcr2(); // Get the faulting address from CR2 register
        if(fault_addr < curproc->sz){
            char *mem = kalloc(); // Allocate a physical page
            if (mem == 0) {
                curproc->killed = 1; // Kill the process if out of memory
            } else {
                memset(mem, 0, PGSIZE);
                if (mappages_wrapper(curproc->pgdir, (char*)PGROUNDDOWN(fault_addr), PGSIZE,
V2P(mem), PTE_W|PTE_U) == 0) {
```



```

        // Only update sz_allocated when a new page is successfully mapped
        curproc->sz_allocated += PGSIZE;
    } else {
        kfree(mem);
        curproc->killed = 1;
    }
}
} else {
    // Handle as a regular page fault (illegal access)
    curproc->killed = 1;
}
}
break;
case T_IRQ0 + IRQ_TIMER:
...

```

types.h

```

#ifndef _TYPES_H_
#define _TYPES_H_

#ifndef __ASSEMBLER__
typedef unsigned int    uint;
typedef unsigned short ushort;
typedef unsigned char   uchar;
typedef uint pde_t;

#endif // __ASSEMBLER__

#endif // _TYPES_H_

```

user.h

```

...
char* sbrk(int);
int sleep(int);
int uptime(void);
int ssualloc(int);
int getvp(void);
int getpp(void);

```

// ulib.c

```

int stat(const char*, struct stat*);
...

```

usys.S

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

vm.c

```
...
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va. If alloc!=0,
// create any required page table pages.
/*static*/ pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    walkpgdir(pde_t *pgdir, const void *va, int alloc)
        return &pgtab[PTX(va)];
}

walkpgdir_fp_t walkpgdir_fp = walkpgdir;

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
    return 0;
}

// Wrapper function
int mappages_wrapper(pde_t *pgdir, void *va, uint size, uint pa, int perm) {
    return mappages(pgdir, va, size, pa, perm);
}

// There is one page table per process, plus one that's used when
// a CPU is not running any process (kpgdir). The kernel uses the
// current process's page table during system calls and interrupts;
kvmalloc(void)
{
    kpgdir = setupkvm();
    switchkvm();

    walkpgdir_fp = walkpgdir;
```

```
}  
// Switch h/w page table register to the kernel-only page table,  
...
```