20192392 컴퓨터학부 김한승

1. 개요

본 과제를 통해 XV6 메모리 및 파일 시스템 구조에 대한 이해를 길렀습니다.

4-1 과제 내용으로는 가상 메모리 할당 (가상 주소 공간 확장) 을 통해 ssualloc\_test 유저 프로그램을 실행할 시,

현재 프로세스가 몇 개의 가상 페이지와 물리 페이지를 할당 받고, 접근 하였는지 확인합니다.

이를 위해 시스템 콜 getvp() 와 getpp()를 추가로 구현하였습니다.

매개 변수는 할당할 가상 주소 공간이며, 매개 변수는 양수입니다. 또한, 페이지 크기 (4K)의 배수를 입력하여야 정상적으로 가상 주소 공간을 할당합니다.

ssualloc() 시스템콜은 가상 메모리 공간을 할당하며, 물리 메모리를 할당하지 않습니다.

이를 통해, 추가로 구현해야 할 사항으로 trap.c내에 trap() 함수에 페이지 폴트 인터럽트에 관한 내용을 추가 구현하여야 했습니다.

4-2 과제 내용으로는 기존 XV6 파일 시스템에서 inode 구조체가 가리키는 블록 구조를 변경하는 것이었습니다.

기존 XV6 파일 시스템에서는 inode 구조체에서 addr 0 ~ 11번 인덱스까지 직접 블록 해당 인덱스가 직접 디스크 블록을 가리키는 것 이며, addr 12 번 인덱스를 1레벨 간접 블록을 사용하였습니다.

이를, 새롭게 addr 0 ~ 5번 까지 인덱스를 직접 블록 , addr 6 ~ 9 번까지 인덱스를 1레벨 간접 블록 addr 10 ~ 11 번 까지 인덱스를 2레벨 간접 블록 addr 12번 인덱스를 3레벨 간접 블록으로 구현하여

총 2,130,438(6 + 128 \* 4 + 128 \* 128 \* 2 + 128 \* 128 \* 128) 개의 데이터 블록을 저장공간으로 사용할 수 있도록 하는 것이 핵심이었습니다.

4-1번 가상 주소 공간 할당 프로세스가 할당받은 물리 , 가상 페이지 공간 추적과 4-2 번 과제 multi-level 파일 시스템 구현을 위한 상세 설계을 2번 상세설계에서 콜 그래프와 함께 설명하겠습니다.

2. 상세 설계

2-1. 가상 메모리 공간 할당 및 메모리 정보 확인

user.h내에서 유저 프로그램에서 시스템 콜 사용을 위한 함수

int ssualloc(int); int getvp(void); int getpp(void); 해당 세개의 시스템 콜 함수가 4-1 과제 수행을 위한 핵심이었습니다.

sysproc.c내 시스템 콜 핸들러 함수

int sys\_ssualloc(void); int sys\_getvp(void); int sys\_getpp(void); 함수를 설명하겠습니다.

또한, 가상 메모리 할당 후, 가상 메모리에 매핑이 되는 물리 메모리를 할당 받지 못한 경우 pagefault()를 통해

가상 메모리에 상응하는 물리 메모리를 페이지 단위로 할당하도록 xv6 수정을 하였습니다.

이에 대한 구현은 trap.c내 trap() 함수 및 page fault 발생시 이에 따른 pagefault() 함수 호출을 통해 해결하였습니다.

두 개의 함수 프로토타입을 쌍으로 설명하는 이유는 한 개의 함수는 sysproc.c내에 구현부가 선언된 시스템 콜 핸들러 함수 이며, 한 개의 함수는 유저 프로그램에서 시스템 콜 호출을 위한 함수 프로토타입 user.h내에 선언된 함수 프로토타입입니다. OS는 유저 프로그램에서 OS가 제공하는 API(시스템 콜) 들을 호출하고, 이에 따라 명시적으로 시스템 함수를 호출하여 일어나는 인터럽트를 통해 OS가 유저가 사용한 시스템 함수와 매핑이 되는 시스템 콜 핸들러 함수를 커널 모드에서 수행 합니다. 이에 따라 시스템 콜 함수가 호출되는 2-1-1 ~ 2-1-3는 같은 함수 콜 그래프를 갖습니다.

2-1-1. int sys\_getvp(void); , int getvp(void);

텍스트, 도표, 스크린샷, 스케치이(가) 표시된 사진

자동 생성된 설명

getvp() 시스템 콜 함수를 호출하는 콜 그래프 입니다. 해당 콜 그래프의 결과를 유저 프로그램인

ssualloc\_test에서 리턴받은 결과를 확인하고, 이를 출력합니다.

해당 시스템 콜 함수를 통해 ssualloc() 시스템 콜 실행 전 후 사용자가 가상 주소 공간에 할당받은 가상 주소 페이지 개수를 확인 할 수 있습니다.

sysproc.c()내 sys\_getvp() 함수는 proc.c 내 int get\_vpage\_count() 함수를 호출 합니다.

2-1-2. int sys\_getpp(void); , int getpp(void);

텍스트, 도표, 스크린샷이(가) 표시된 사진

자동 생성된 설명

getpp() 시스템 콜 함수를 호출하는 콜 그래프 입니다. 해당 콜 그래프의 결과를 유저 프로그램인

ssualloc\_test에서 리턴받은 결과를 확인하고, 이를 출력합니다.

해당 시스템 콜 함수를 통해 사용자가 현재 가상 페이지와 매핑되는 물리 페이지를 몇 개 할당 받았는지 개수를 확인할 수 있습니다.

sysproc.c()내 sys\_getpp() 함수는 proc.c 내 int get\_ppage\_count() 함수를 호출 합니다.

2-1-3. int sys\_ssualloc(void); , int ssualloc(int vmem\_size);

텍스트, 도표, 스크린샷, 평행이(가) 표시된 사진

자동 생성된 설명

getvp() 함수나 getpp() 함수는 프로세스가 할당받은 가상 페이지 개수 , 물리 페이지 개수를 리턴 받습니다.

ssualloc() 함수는 유저 프로그램에 가상 주소 공간을 추가 하였는지 성공 여부를 반환합니다.

ssualloc() 함수는 인자를 1개 받는데, 이는 페이지의 배수가 되는 양수 인자입니다.

만약, 양수가 아닌 인자 거나, 페이지의 배수가 되지 않는 (1234)와 같은 인자를 받을 경우 주소 공간을 추가 하지 못하였기 때문에 -1을 반환합니다.

해당 함수는 가상 주소 공간을 추가하지만, 이에 매핑이 되는 물리 페이지를 같이 할당 하지 않기 때문에

늘어난 가상 페이지 수가 매핑된 물리 페이지가 메모리에 없는 경우 이에 따라,

trap() 함수를 호출하며 페이지 폴트에 대한 인터럽트를 해결하여야합니다.

2-1-4. trap.c 내 trap(struct trapframe \*tf) 및 pagefault(struct trapframe \*tf)

텍스트, 도표, 스크린샷, 직사각형이(가) 표시된 사진

자동 생성된 설명

ssualloc() 시스템 콜 추가를 통해 유저 프로그램에서 가상 페이지를 추가로 할당 받을 수 있었다.

이에 따라, 물리 페이지를 같이 할당 받지 못했기 때문에 페이지 폴트라는 인터럽트가 추가로 발생 가능 해졌다.

늘어난 가상 페이지를 접근 할 경우, 물리 페이지가 메모리에 할당 되지 않은 경우 페이지 폴트 인터럽트가 발생하며

이에 따라 trap() 함수를 호출 pagefault() 함수를 호출하여 인터럽트를 해결합니다.

CR2 레지스터 값(페이지 폴트 발생시 발생한 주소가 저장되는 레지스터)를 통해 해당 주소에 물리 메모리를 할당합니다.

kalloc() 이라는 함수를 활용합니다. // 물리 메모리 할당

페이지 폴트 실패시, 프로세스 종료 killed 비트를 1로 설정하여 프로세스를 종료시킵니다.

2-2. multi-level 파일 시스템 구현

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

비교를 위해 교수님께서 제공하신 과제 명세서내에 inode 구조체에서 addr 배열 부분을 인용하였습니다.

해당 과제 구현을 위해 param.h내에 대용량 파일 생성을 위한 FSSIZE를 수정하였습니다 1000 -> 250,0000

기존 XV6 파일 시스템에서 저장공간으로 사용할 수 잇는 데이터 블록은 1 \* 12 + (128) \*1 == 140 개 였습니다.

새로운 XV6 multi-level 파일 시스템에서는 1 \* 6 + 128 \* 4 + 128 \* 128 \* 2 + 128 \* 128 \* 128 총 2,130,438 개의 데이터 블록을 저장공간으로 사용할 수 있게 구현합니다.

fs.h 내에 dinode 구조체와 file.h내에 inode 구조체에 addrs 배열을 추가로 수정하였습니다.

bmap() 함수와 itrunc() 함수를 추가로 수정하였습니다.

bmap() 함수는 할당받은 디스크 블록을 addrs 배열에 인덱스 블록들이 가리키도록 인덱싱하여 가리킵니다.

itrunc() 함수는 파일이 삭제 unlink 되는 등 할경우 addrs 배열에 인덱스 블록들이 가리키는 디스크 블록이 있을 경우 이를 할당 해제 하기 위한 함수입니다.

2-2-1. bmap() 함수 콜 그래프

bmap() 함수는 ssufs\_test.c 테스트 파일에서 본것과 같이

read() , write() 함수를 호출할 때 호출됩니다.

텍스트, 도표, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명

유저 프로그램 ssufs\_test.c 내에 read() write() 함수를 사용하는 것을 통해 bmap() 함수가 호출 되는 것을 볼 수 있습니다.

read() write() 시스템 콜 함수를 호출하면 , 여타 다른 시스템 콜 함수 와 같이 인터럽트 발생 및 trap() 호출

시스템 콜 핸들러 함수가 호출되고, sys\_read() , sys\_write() 그 후, 해당 함수 에서 fileread()

file.c 내 fileread() 나 filewrite() 함수를 호출합니다. 해당 함수에서 inode에 데이터를 작성하기 위해

fs.c 내 readi() / writei() 함수를 호출합니다. 해당 readi() 함수나 writei() 함수는 bmap() 함수를 호출합니다.

fileread() , filewrite()가 readi() , writei()를 호출하는 이유는 파일 시스템에 접근하기 위함이고,

readi() 나 writei()는 실제로 inode를 읽거나 쓰기 위해 bmap() 함수를 호출합니다.

bmap() 함수는 파일의 데이터 블록 매핑을 수행합니다.

inode와 dinode 구조체를 multi-level file system으로 재 구현하면서 bmap() 함수를 새로 구현해야 하는 이유입니다.

2-2-2. itrunc() 함수 콜 그래프

멀티 레벨 파일 시스템 구현을 위해 재 구현해야 할 함수 중 2번째 itrunc() 함수입니다.

해당 함수는 read() 나 write()를 통해 bmap() 함수가 파일의 데이터 블록 매핑을 수행했다면

unlink() 함수를 통해서 파일을 제거할 때 파일의 데이터 블록을 해제하고 inode를 정리합니다.

itrunc() 함수는 inode에서 관리하는 데이터 블록을 해제합니다.

도표, 텍스트, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명

unlink() 후 링크가 더 이상 없는 파일은 inode를 해제 시키고, 이를 위해 itrunc() 함수를 호출합니다.

이 때문에 기존 xv6 파일 시스템에서 직접 블록 , 간접 블록 0~11 인덱스 12 인덱스를 사용하던 것에서

블록의 종류 2중 ,3중 간접 블록 사용 및 인덱스 마다 블록에 사용이 달라졌기 때문에 수정하여야합니다.

3. 실행 결과

3-1. ssualloc\_test

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3-2. ssufs\_test

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

4. 소스 코드

<https://github.com/mit-pdos/xv6-public> 을 원본으로 한 소스코드에서

해당 파일에 수정된 코드 부분에 대해서만 추가하였습니다.

4-1. user.h

// 추가 시스템 콜

int ssualloc(int);   // 가상 메모리 할당을 위한 시스템 콜 함수

int getvp(void);     // 호출한 가상 메모리 페이지 개수 반환 시스템 콜 함수

int getpp(void);     // 호출한 물리 메모리 페이지 개수 반환 시스템 콜 함수

4-2. usys.S

SYSCALL(ssualloc)

SYSCALL(getvp)

SYSCALL(getpp)

4-3. syscall.c

extern int sys\_ssualloc(void);

extern int sys\_getvp(void);

extern int sys\_getpp(void);

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

[SYS\_fork]    sys\_fork,

[SYS\_exit]    sys\_exit,

[SYS\_wait]    sys\_wait,

[SYS\_pipe]    sys\_pipe,

[SYS\_read]    sys\_read,

[SYS\_kill]    sys\_kill,

[SYS\_exec]    sys\_exec,

[SYS\_fstat]   sys\_fstat,

[SYS\_chdir]   sys\_chdir,

[SYS\_dup]     sys\_dup,

[SYS\_getpid]  sys\_getpid,

[SYS\_sbrk]    sys\_sbrk,

[SYS\_sleep]   sys\_sleep,

[SYS\_uptime]  sys\_uptime,

[SYS\_open]    sys\_open,

[SYS\_write]   sys\_write,

[SYS\_mknod]   sys\_mknod,

[SYS\_unlink]  sys\_unlink,

[SYS\_link]    sys\_link,

[SYS\_mkdir]   sys\_mkdir,

[SYS\_close]   sys\_close,

[SYS\_ssualloc] sys\_ssualloc,

[SYS\_getvp]   sys\_getvp,

[SYS\_getpp]   sys\_getpp,

};

4-4. syscall.h

#define SYS\_ssualloc 22

#define SYS\_getvp  23

#define SYS\_getpp  24

4-5. sysproc.c

// 설계4 관련 시스템 콜 호출 구현

// sys\_ssualloc(), sys\_getvp(), sys\_getpp()

int

sys\_ssualloc(void)

{

  int vmem\_size;  // 할당할 가상페이지 크기

  // 인자 오류

  if(argint(0, &vmem\_size) < 0)

    return -1;

  // 추가 할당받을 가상 메모리 크기가 양수가 아니거나

  // 페이지 크기의 배수가 아닐 경우 -1을 반환한다.

  if(vmem\_size <= 0 || vmem\_size % PGSIZE)

  return -1;

  struct proc \*p = myproc();

  uint oldsz = p->sz;

  uint newsz = p->sz + vmem\_size; // 새롭게 할당할 가상 주소 공간

  // 커널 침범 overflow 여부 확인 및 새로운 크기가 예전 크기보다 작은지 확인

  if(newsz >= KERNBASE || newsz < oldsz)

    return -1;

  p-> sz = newsz; // 늘어난 가상 주소 공간으로 초기화

  return oldsz;

}

int

sys\_getvp(void)

{

  return get\_vpage\_count();

}

int

sys\_getpp(void)

{

  return get\_ppage\_count();

}

4-6. Makefile

UPROGS=\

    \_cat\

    \_echo\

    \_forktest\

    \_grep\

    \_init\

    \_kill\

    \_ln\

    \_ls\

    \_mkdir\

    \_rm\

    \_sh\

    \_stressfs\

    \_usertests\

    \_wc\

    \_zombie\

    \_ssualloc\_test\

    \_ssufs\_test\

EXTRA=\

    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\

    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\

    printf.c umalloc.c ssualloc\_test.c ssufs\_test.c\

    README dot-bochsrc \*.pl toc.\* runoff runoff1 runoff.list\

    .gdbinit.tmpl gdbutil\

4-7. param.h

#define FSSIZE       2500000  // size of file system in blocks

4-8. proc.c

int

get\_vpage\_count(void)

{

  struct proc \*p = myproc();

  if(!p) return -1;

  // 할당받은 가상 페이지 개수 확인을 위해 할당 받은 가상 주소 공간 크기를 페이지 크기로 나누어 반환

  return ( p->sz / PGSIZE );

}

int

get\_ppage\_count(void)

{

  struct proc \*p = myproc();

  if(!p) return -1;

  int count = 0,i,j;

  pde\_t \*pgdir = p->pgdir; // 현재 프로세스의 페이지 디렉토리 주소

  for(i = 0; i < PDX(KERNBASE); i++)

  {

    pde\_t pde = pgdir[i];

    if(pde & PTE\_P)  // 현재 페이지 디렉토리 항목의 P비트를 통해 유효한지 확인

    {

      pte\_t \*pgtable = (pte\_t \*)P2V(PTE\_ADDR(pde));

      for(j = 0; j < NPTENTRIES; j++)

      {

        pte\_t pte = pgtable[j]; // 페이지 디렉토리 에서 j번째 페이지 테이블 엔트리를 가져오는것.

        if(pte & PTE\_P) count++; // 현재 페이지 테이블 엔트리의 P비트를 통해 유효한지 확인 및 U비트를 통해 유저 공간 물리페이지이면, 물리 페이지 개수 추가

      }

    }

  }

  return count;

}

4-9. defs.h

// proc.c

int             cpuid(void);

void            exit(void);

int             fork(void);

int             growproc(int);

int             kill(int);

struct cpu\*     mycpu(void);

struct proc\*    myproc();

void            pinit(void);

void            procdump(void);

void            scheduler(void) \_\_attribute\_\_((noreturn));

void            sched(void);

void            setproc(struct proc\*);

void            sleep(void\*, struct spinlock\*);

void            userinit(void);

int             wait(void);

void            wakeup(void\*);

void            yield(void);

int             get\_vpage\_count(void); // 할당된 가상 페이지 개수 반환

int             get\_ppage\_count(void); // 할당된 물리 페이지 개수 반환

4-10. vm.c

#include "param.h"

#include "types.h"

#include "defs.h"

#include "x86.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

#include "elf.h"

extern char data[];  // defined by kernel.ld

pde\_t \*kpgdir;  // for use in scheduler()

// Set up CPU's kernel segment descriptors.

// Run once on entry on each CPU.

void

seginit(void)

{

  struct cpu \*c;

  // Map "logical" addresses to virtual addresses using identity map.

  // Cannot share a CODE descriptor for both kernel and user

  // because it would have to have DPL\_USR, but the CPU forbids

  // an interrupt from CPL=0 to DPL=3.

  c = &cpus[cpuid()];

  c->gdt[SEG\_KCODE] = SEG(STA\_X|STA\_R, 0, 0xffffffff, 0);

  c->gdt[SEG\_KDATA] = SEG(STA\_W, 0, 0xffffffff, 0);

  c->gdt[SEG\_UCODE] = SEG(STA\_X|STA\_R, 0, 0xffffffff, DPL\_USER);

  c->gdt[SEG\_UDATA] = SEG(STA\_W, 0, 0xffffffff, DPL\_USER);

  lgdt(c->gdt, sizeof(c->gdt));

}

// 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;

  pte\_t \*pgtab;

  pde = &pgdir[PDX(va)];

  if(\*pde & PTE\_P){

    pgtab = (pte\_t\*)P2V(PTE\_ADDR(\*pde));

  } else {

    if(!alloc || (pgtab = (pte\_t\*)kalloc()) == 0)

      return 0;

    // Make sure all those PTE\_P bits are zero.

    memset(pgtab, 0, PGSIZE);

    // The permissions here are overly generous, but they can

    // be further restricted by the permissions in the page table

    // entries, if necessary.

    \*pde = V2P(pgtab) | PTE\_P | PTE\_W | PTE\_U;

  }

  return &pgtab[PTX(va)];

}

// 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;

  pte\_t \*pte;

  a = (char\*)PGROUNDDOWN((uint)va);

  last = (char\*)PGROUNDDOWN(((uint)va) + size - 1);

  for(;;){

    if((pte = walkpgdir(pgdir, a, 1)) == 0)

      return -1;

    if(\*pte & PTE\_P)

      panic("remap");

    \*pte = pa | perm | PTE\_P;

    if(a == last)

      break;

    a += PGSIZE;

    pa += PGSIZE;

  }

  return 0;

}

// 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;

// page protection bits prevent user code from using the kernel's

// mappings.

//

// setupkvm() and exec() set up every page table like this:

//

//   0..KERNBASE: user memory (text+data+stack+heap), mapped to

//                phys memory allocated by the kernel

//   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)

//   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)

//                for the kernel's instructions and r/o data

//   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,

//                                  rw data + free physical memory

//   0xfe000000..0: mapped direct (devices such as ioapic)

//

// The kernel allocates physical memory for its heap and for user memory

// between V2P(end) and the end of physical memory (PHYSTOP)

// (directly addressable from end..P2V(PHYSTOP)).

// This table defines the kernel's mappings, which are present in

// every process's page table.

static struct kmap {

  void \*virt;

  uint phys\_start;

  uint phys\_end;

  int perm;

} kmap[] = {

 { (void\*)KERNBASE, 0,             EXTMEM,    PTE\_W}, // I/O space

 { (void\*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata

 { (void\*)data,     V2P(data),     PHYSTOP,   PTE\_W}, // kern data+memory

 { (void\*)DEVSPACE, DEVSPACE,      0,         PTE\_W}, // more devices

};

// Set up kernel part of a page table.

pde\_t\*

setupkvm(void)

{

  pde\_t \*pgdir;

  struct kmap \*k;

  if((pgdir = (pde\_t\*)kalloc()) == 0)

    return 0;

  memset(pgdir, 0, PGSIZE);

  if (P2V(PHYSTOP) > (void\*)DEVSPACE)

    panic("PHYSTOP too high");

  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)

    if(mappages(pgdir, k->virt, k->phys\_end - k->phys\_start,

                (uint)k->phys\_start, k->perm) < 0) {

      freevm(pgdir);

      return 0;

    }

  return pgdir;

}

// Allocate one page table for the machine for the kernel address

// space for scheduler processes.

void

kvmalloc(void)

{

  kpgdir = setupkvm();

  switchkvm();

}

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

// for when no process is running.

void

switchkvm(void)

{

  lcr3(V2P(kpgdir));   // switch to the kernel page table

}

// Switch TSS and h/w page table to correspond to process p.

void

switchuvm(struct proc \*p)

{

  if(p == 0)

    panic("switchuvm: no process");

  if(p->kstack == 0)

    panic("switchuvm: no kstack");

  if(p->pgdir == 0)

    panic("switchuvm: no pgdir");

  pushcli();

  mycpu()->gdt[SEG\_TSS] = SEG16(STS\_T32A, &mycpu()->ts,

                                sizeof(mycpu()->ts)-1, 0);

  mycpu()->gdt[SEG\_TSS].s = 0;

  mycpu()->ts.ss0 = SEG\_KDATA << 3;

  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;

  // setting IOPL=0 in eflags \*and\* iomb beyond the tss segment limit

  // forbids I/O instructions (e.g., inb and outb) from user space

  mycpu()->ts.iomb = (ushort) 0xFFFF;

  ltr(SEG\_TSS << 3);

  lcr3(V2P(p->pgdir));  // switch to process's address space

  popcli();

}

// Load the initcode into address 0 of pgdir.

// sz must be less than a page.

void

inituvm(pde\_t \*pgdir, char \*init, uint sz)

{

  char \*mem;

  if(sz >= PGSIZE)

    panic("inituvm: more than a page");

  mem = kalloc();

  memset(mem, 0, PGSIZE);

  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE\_W|PTE\_U);

  memmove(mem, init, sz);

}

// Load a program segment into pgdir.  addr must be page-aligned

// and the pages from addr to addr+sz must already be mapped.

int

loaduvm(pde\_t \*pgdir, char \*addr, struct inode \*ip, uint offset, uint sz)

{

  uint i, pa, n;

  pte\_t \*pte;

  if((uint) addr % PGSIZE != 0)

    panic("loaduvm: addr must be page aligned");

  for(i = 0; i < sz; i += PGSIZE){

    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)

      panic("loaduvm: address should exist");

    pa = PTE\_ADDR(\*pte);

    if(sz - i < PGSIZE)

      n = sz - i;

    else

      n = PGSIZE;

    if(readi(ip, P2V(pa), offset+i, n) != n)

      return -1;

  }

  return 0;

}

// Allocate page tables and physical memory to grow process from oldsz to

// newsz, which need not be page aligned.  Returns new size or 0 on error.

int

allocuvm(pde\_t \*pgdir, uint oldsz, uint newsz)

{

  char \*mem;

  uint a;

  if(newsz >= KERNBASE)

    return 0;

  if(newsz < oldsz)

    return oldsz;

  a = PGROUNDUP(oldsz);

  for(; a < newsz; a += PGSIZE){

    mem = kalloc();

    if(mem == 0){

      cprintf("allocuvm out of memory\n");

      deallocuvm(pgdir, newsz, oldsz);

      return 0;

    }

    memset(mem, 0, PGSIZE);

    if(mappages(pgdir, (char\*)a, PGSIZE, V2P(mem), PTE\_W|PTE\_U) < 0){

      cprintf("allocuvm out of memory (2)\n");

      deallocuvm(pgdir, newsz, oldsz);

      kfree(mem);

      return 0;

    }

  }

  return newsz;

}

// Deallocate user pages to bring the process size from oldsz to

// newsz.  oldsz and newsz need not be page-aligned, nor does newsz

// need to be less than oldsz.  oldsz can be larger than the actual

// process size.  Returns the new process size.

int

deallocuvm(pde\_t \*pgdir, uint oldsz, uint newsz)

{

  pte\_t \*pte;

  uint a, pa;

  if(newsz >= oldsz)

    return oldsz;

  a = PGROUNDUP(newsz);

  for(; a  < oldsz; a += PGSIZE){

    pte = walkpgdir(pgdir, (char\*)a, 0);

    if(!pte)

      a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;

    else if((\*pte & PTE\_P) != 0){

      pa = PTE\_ADDR(\*pte);

      if(pa == 0)

        panic("kfree");

      char \*v = P2V(pa);

      kfree(v);

      \*pte = 0;

    }

  }

  return newsz;

}

// Free a page table and all the physical memory pages

// in the user part.

void

freevm(pde\_t \*pgdir)

{

  uint i;

  if(pgdir == 0)

    panic("freevm: no pgdir");

  deallocuvm(pgdir, KERNBASE, 0);

  for(i = 0; i < NPDENTRIES; i++){

    if(pgdir[i] & PTE\_P){

      char \* v = P2V(PTE\_ADDR(pgdir[i]));

      kfree(v);

    }

  }

  kfree((char\*)pgdir);

}

// Clear PTE\_U on a page. Used to create an inaccessible

// page beneath the user stack.

void

clearpteu(pde\_t \*pgdir, char \*uva)

{

  pte\_t \*pte;

  pte = walkpgdir(pgdir, uva, 0);

  if(pte == 0)

    panic("clearpteu");

  \*pte &= ~PTE\_U;

}

// Given a parent process's page table, create a copy

// of it for a child.

pde\_t\*

copyuvm(pde\_t \*pgdir, uint sz)

{

  pde\_t \*d;

  pte\_t \*pte;

  uint pa, i, flags;

  char \*mem;

  if((d = setupkvm()) == 0)

    return 0;

  for(i = 0; i < sz; i += PGSIZE){

    if((pte = walkpgdir(pgdir, (void \*) i, 0)) == 0)

      panic("copyuvm: pte should exist");

    if(!(\*pte & PTE\_P))

      panic("copyuvm: page not present");

    pa = PTE\_ADDR(\*pte);

    flags = PTE\_FLAGS(\*pte);

    if((mem = kalloc()) == 0)

      goto bad;

    memmove(mem, (char\*)P2V(pa), PGSIZE);

    if(mappages(d, (void\*)i, PGSIZE, V2P(mem), flags) < 0) {

      kfree(mem);

      goto bad;

    }

  }

  return d;

bad:

  freevm(d);

  return 0;

}

//PAGEBREAK!

// Map user virtual address to kernel address.

char\*

uva2ka(pde\_t \*pgdir, char \*uva)

{

  pte\_t \*pte;

  pte = walkpgdir(pgdir, uva, 0);

  if((\*pte & PTE\_P) == 0)

    return 0;

  if((\*pte & PTE\_U) == 0)

    return 0;

  return (char\*)P2V(PTE\_ADDR(\*pte));

}

// Copy len bytes from p to user address va in page table pgdir.

// Most useful when pgdir is not the current page table.

// uva2ka ensures this only works for PTE\_U pages.

int

copyout(pde\_t \*pgdir, uint va, void \*p, uint len)

{

  char \*buf, \*pa0;

  uint n, va0;

  buf = (char\*)p;

  while(len > 0){

    va0 = (uint)PGROUNDDOWN(va);

    pa0 = uva2ka(pgdir, (char\*)va0);

    if(pa0 == 0)

      return -1;

    n = PGSIZE - (va - va0);

    if(n > len)

      n = len;

    memmove(pa0 + (va - va0), buf, n);

    len -= n;

    buf += n;

    va = va0 + PGSIZE;

  }

  return 0;

}

//PAGEBREAK!

// Blank page.

//PAGEBREAK!

// Blank page.

//PAGEBREAK!

// Blank page.

4-11. trap.c

void

pagefault(struct trapframe \*tf)

{

  // CR2 레지스터 값 반환

  // CR2 레지스터란? 페이지 폴트 발생시 발생한 주소가 저장되는 레지스터

  // rcr2() 함수를 통해 폴트난 가상 주소를 받는다.

  uint fault\_va = rcr2();

  struct proc \* p = myproc();

  char \*mem = kalloc(); // 물리 메모리 할당

  if(mem == 0)

  {

    // 페이지 폴트 실패시, 프로세스 종료 killed 비트를 켬으로써

    p->killed = 1;

    return;

  }

  // memset() 함수를 통해 할당받은

  // 페이지 크기 만큼의 물리 메모리 초기화

  memset(mem, 0, PGSIZE);

  if(mappages(p->pgdir, (char \*)PGROUNDDOWN(fault\_va), PGSIZE, V2P(mem), PTE\_W | PTE\_U) < 0)

  {

    // 페이지 폴트 실패시, 프로세스 종료 killed 비트를 켬으로써

    // kalloc()은 이루어졌었으니, kfree()를 통해 물리 메모리 할당 해제

    kfree(mem);

    p->killed = 1;

    return;

  }

  return;

}

//PAGEBREAK: 41

void

trap(struct trapframe \*tf)

{

  if(tf->trapno == T\_SYSCALL){

    if(myproc()->killed)

      exit();

    myproc()->tf = tf;

    syscall();

    if(myproc()->killed)

      exit();

    return;

  }

  switch(tf->trapno){

  case T\_IRQ0 + IRQ\_TIMER:

    if(cpuid() == 0){

      acquire(&tickslock);

      ticks++;

      wakeup(&ticks);

      release(&tickslock);

    }

    lapiceoi();

    break;

  case T\_IRQ0 + IRQ\_IDE:

    ideintr();

    lapiceoi();

    break;

  case T\_IRQ0 + IRQ\_IDE+1:

    // Bochs generates spurious IDE1 interrupts.

    break;

  case T\_IRQ0 + IRQ\_KBD:

    kbdintr();

    lapiceoi();

    break;

  case T\_IRQ0 + IRQ\_COM1:

    uartintr();

    lapiceoi();

    break;

  case T\_IRQ0 + 7:

  case T\_IRQ0 + IRQ\_SPURIOUS:

    cprintf("cpu%d: spurious interrupt at %x:%x\n",

            cpuid(), tf->cs, tf->eip);

    lapiceoi();

    break;

  case T\_PGFLT:

  // 가상 주소 공간을 할당받고, 그에 따른 물리 페이지는 나중에 할당받게 되어

  // 페이지 폴트가 발생할수 있게 됨에 따라

  // trap() 함수내에서 페이지 폴트를 처리해야한다.

  pagefault(tf);

  break;

  //PAGEBREAK: 13

  default:

    if(myproc() == 0 || (tf->cs&3) == 0){

      // In kernel, it must be our mistake.

      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",

              tf->trapno, cpuid(), tf->eip, rcr2());

      panic("trap");

    }

    // In user space, assume process misbehaved.

    cprintf("pid %d %s: trap %d err %d on cpu %d "

            "eip 0x%x addr 0x%x--kill proc\n",

            myproc()->pid, myproc()->name, tf->trapno,

            tf->err, cpuid(), tf->eip, rcr2());

    myproc()->killed = 1;

  }

  // Force process exit if it has been killed and is in user space.

  // (If it is still executing in the kernel, let it keep running

  // until it gets to the regular system call return.)

  if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

    exit();

  // Force process to give up CPU on clock tick.

  // If interrupts were on while locks held, would need to check nlock.

  if(myproc() && myproc()->state == RUNNING &&

     tf->trapno == T\_IRQ0+IRQ\_TIMER)

    yield();

  // Check if the process has been killed since we yielded

  if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

    exit();

}

4-12. fs.h

// multi-level File System

// multi-level INDIRECT 블록의 개수

#define   DIRECT\_CNT        6

#define INDIRECT\_CNT        4

#define INDIRECT\_CNT\_DOUBLE 2

#define INDIRECT\_CNT\_TRIPLE 1

#define NDIRECT DIRECT\_CNT

#define NINDIRECT (BSIZE / sizeof(uint))

#define NINDIRECT\_DOUBLE (NINDIRECT \* NINDIRECT)

#define NINDIRECT\_TRIPLE (NINDIRECT \* NINDIRECT \* NINDIRECT)

// 하나의 파일이 가질 수 있는 최대 데이터 블록의 개수

#define MAXFILE (NDIRECT + NINDIRECT \* INDIRECT\_CNT + NINDIRECT\_DOUBLE \* INDIRECT\_CNT\_DOUBLE + NINDIRECT\_TRIPLE \* INDIRECT\_CNT\_TRIPLE)

// 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[NDIRECT + INDIRECT\_CNT + INDIRECT\_CNT\_DOUBLE + INDIRECT\_CNT\_TRIPLE]; // Data block addresses

};

4-13. fs.c

static uint

bmap(struct inode \*ip, uint bn)

{

  // 새로운 파일 시스템을 위해  블록을 할당하기 위한 함수 수정

  // addrs 0~5 인덱스 DIRECT 블록, addrs 6~9 인덱스 INDIRECT 블록

  // addrs 10,11 인덱스 DOULBE-INDIRECT 블록 , addrs 12 인덱스 TRIPLE-INDIRECT 불록

  // bn에 인자는 0 ~ 할당할 블록 개수 -1 까지 bmap() 을 할당할 블록 개수만큼 호출합니다.

  uint addr, \*a;

  struct buf \*bp;

  int i;

  // DIRECT 블록 (1) \* 6

  if(bn < NDIRECT)

  {

    if((addr = ip->addrs[bn]) == 0)

      ip->addrs[bn] = addr = balloc(ip->dev);

    return addr;

  }

  bn -= NDIRECT; // 인덱스 계산을 위해 앞에 직접 블록 개수만큼 블록 번호 차감

  // INDIRECT 블록 (128) \* 4

  // addrs 6,7,8,9 할당

  if(bn < NINDIRECT \* INDIRECT\_CNT)

  {

  i = DIRECT\_CNT + (bn / NINDIRECT); // i는 6~9 사이 값

  while(bn >= NINDIRECT)

    bn  -= NINDIRECT; // 인덱스 계산을 위한 블록 번호 차감

    // Load indirect block, allocating if necessaray

  if((addr = ip->addrs[i]) == 0) {

    ip->addrs[i] = addr = balloc(ip->dev); // 현재 addrs 할당된 블록 없다면 할당

  }

  bp = bread(ip->dev, addr);

  a = (uint\*)bp->data;

  if((addr = a[bn]) == 0)

  {

    a[bn] = addr = balloc(ip->dev); // 1단계 블록

    log\_write(bp);

  }

  brelse(bp);

  return addr;

  }

  bn -= NINDIRECT \* INDIRECT\_CNT; // 앞에 직접 블록 및 SINGLE-INDIRECT 블록 만큼 차감

  if(bn < NINDIRECT\_DOUBLE \* INDIRECT\_CNT\_DOUBLE)

  {

  // DOULBE-INDIRECT 블록 (128 \* 128) \*2

  // addrs 10,11 할당

  i = DIRECT\_CNT + INDIRECT\_CNT + ( bn / NINDIRECT\_DOUBLE); // i는 10 , 11 중 하나

  if(bn >= NINDIRECT\_DOUBLE)

    bn -= NINDIRECT\_DOUBLE; // 인덱스 계산을 위한 블록 번호 차감

  if((addr = ip->addrs[i]) == 0)

    ip->addrs[i] = 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);

  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 -= NINDIRECT\_DOUBLE \* INDIRECT\_CNT\_DOUBLE; // 앞에 이중 간접 블록에 할당된 블록 개수 만큼 차감

  if(bn < NINDIRECT\_TRIPLE)

  {

  // TRIPLE-INDIRECT 블록 할당

  // 인덱스는 12 고정 (13개의 addrs중 맨 마지막 블록은 3중 간접 블록)

  if((addr = ip->addrs[12]) == 0)

    ip->addrs[12] = addr = balloc(ip->dev);

  bp = bread(ip->dev, addr);

  a = (uint\*)bp->data;

  // 1차 단계

  if((addr = a[bn / NINDIRECT\_DOUBLE]) == 0)

  {

    a[bn / NINDIRECT\_DOUBLE] = addr = balloc(ip->dev);

    log\_write(bp);

  }

  brelse(bp);

  bn = bn % NINDIRECT\_DOUBLE;

  bp = bread(ip->dev, addr);

  a = (uint \*)bp->data;

  // 2차 단계

  if((addr = a[bn / NINDIRECT]) == 0)

  {

    a[(bn / NINDIRECT)] = addr = balloc(ip->dev);

    log\_write(bp);

  }

  brelse(bp);

  bp = bread(ip->dev, addr);

  a = (uint \*)bp->data;

  // 마지막 단계

  if((addr = a[bn % NINDIRECT]) == 0)

  {

    a[bn % NINDIRECT] = addr = balloc(ip->dev);

    log\_write(bp);

  }

  brelse(bp);

  return addr;

  }

  panic("bmap: out of range"); // 3중 간접 블록으로도 할당을 못할정도로 블록개수가 많음. -> 실패

}

// Truncate inode (discard contents).

// Only called when the inode has no links

// to it (no directory entries referring to it)

// and has no in-memory reference to it (is

// not an open file or current directory).

static void

itrunc(struct inode \*ip)

{

  int i, j,start,end;

  struct buf \*bp;

  struct buf \*bp2;

  struct buf \*bp3;

  uint \*a,\*a2,\*a3;

  // 0 ~ 5 까지 인덱스의 직접 블록 할당 해제

  for(i = 0; i < NDIRECT; i++){

    if(ip->addrs[i]){

      bfree(ip->dev, ip->addrs[i]);

      ip->addrs[i] = 0;

    }

  }

  start = NDIRECT;

  end = NDIRECT + INDIRECT\_CNT;

  // start : 6 ~ end : 9 까지 인덱스의 싱글 간접 블록 할당 해제

  for(i = start; i < end; i++)

  {

  if(ip->addrs[i]){

    bp = bread(ip->dev, ip->addrs[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[i]);

    ip->addrs[i] = 0;

  }

  }

  // start :10 ~ end : 11 까지 인덱스의 더블 간접 블록 할당 해제

  start = NDIRECT + INDIRECT\_CNT;

  end   = start + INDIRECT\_CNT\_DOUBLE;

  for(i = start; i < end; i++)

  {

  if(ip->addrs[i])

  {

    bp = bread(ip->dev, ip->addrs[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(int y = 0; y < NINDIRECT ; y++)

        {

          if(a2[y])

            bfree(ip->dev, a2[y]);

        }

        brelse(bp2);

        bfree(ip->dev, a[j]);

      }

    }

    brelse(bp);

    bfree(ip->dev, ip->addrs[i]);

    ip->addrs[i] = 0;

  }

  }

  // 마지막 인덱스 12 의 삼중 간접 블록 할당 해제

  // 1 -> 2 -> 3 단계 간접 블록을 타고 들어가서 할당된 블록이 있으면 할당 해제

  if(ip->addrs[12])

  {

  bp = bread(ip->dev, ip->addrs[12]);

  a = (uint \*)bp->data;

  for(j = 0; j < 2; j++)

  {

    if(a[j])

    {

      bp2 = bread(ip->dev, a[j]);

      a2 = (uint \*)bp2->data;

      for(int y = 0; y < NINDIRECT; y++)

      {

        if(a2[y])

        {

          bp3 = bread(ip->dev, a2[y]);

          a3 = (uint \*)bp3->data;

          for(int z = 0; z < NINDIRECT; z++)

          {

            if(a3[z])

              bfree(ip->dev, a3[z]);

          }

          brelse(bp3);

          bfree(ip->dev, a2[y]);

        }

      }

      brelse(bp2);

      bfree(ip->dev, a[j]);

    }

  }

  brelse(bp);

  bfree(ip->dev, ip->addrs[12]);

  ip->addrs[12] = 0;

  }

  ip->size = 0;

  iupdate(ip);

}

4-14. file.h

// in-memory copy of an inode

struct inode {

  uint dev;           // Device number

  uint inum;          // Inode number

  int ref;            // Reference count

  struct sleeplock lock; // protects everything below here

  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode

  short major;

  short minor;

  short nlink;

  uint size;

  uint addrs[NDIRECT + INDIRECT\_CNT + INDIRECT\_CNT\_DOUBLE + INDIRECT\_CNT\_TRIPLE];

};

4-15. ssualloc\_test.c (수정 불가)

#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();

}

4-16. ssufs\_test.c (수정 불가)

#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();

}