컴퓨터학부, 20202925, 강준현

1. 개요

xv6의 파일시스템의 동작 원리를 이해하고, .xv6에 CS기반의 새로운 파일 시스템을 추가한다. 새로운 파일 시스템은 direct블록만 이용하며 이 안에 (블락번호, 연속된 수)를 각각 int의 상위3b 하위1b를 이용하여 표현한다. 이에 따라 최대로 저장 가능한 파일의 크기 등이 변화한다. 이를 통해 파일 시스템의 보다 깊은 이해를 한다.

2. 상세설계

xv6 filesystem의 file을 새로 만드는 상황분석

open에서 O\_CREATE 플래그를 인식하며 create를 호출한다. ialloc을 이용해 해당 inode구조체의 타입 등 정보를 저장한다.

xv6 filesystem의 file에 wirte하는 상황 분석

우선 시트템콜wirte를 사용자가 호출한다. 이 시스템콜은 filewrite라는 함수를 부르고 이 함수는 writei함수를 부른다. writei에서 bmap함수를 호출해 inode의 블록에 balloc을 이용해 데이터블록의 주소를 저장해준다.

CS파일 시스템을 추가하기위해...

open에서 O\_CS를 인식하여 create에type을 T\_CS로 넘긴다.

write상황에서는 writei함수 와 bmap에서 내부에서 type이 T\_CS인 상황을 분리해 따로 처리해 기존 파일 시스템은 유지하며 T\_CS인 파일만 다르게 처리한다.

itrunc에서 도 T\_CS인 상황을 나누어 (블락번호, 연속된 수)를 각각 int의 상위3b 하위1b 규칙을 이용하여 블록을 해제한다.

printinfo함수는 시스템콜로 구현하였다. sysfile.c에 구현하였다.

3. 구현방법

open시스템콜함수에서

if(omode & O\_CS){

    ip = create(path, T\_CS, 0, 0);

omode에 O\_CS가 들어온 경우 create에 type을 T\_CS로 넘긴다.

write는 filewrite함수에서 잘 쪼개어 writei로 전달되는 것들을 갖고 writei에서 기존의 최대파일 크기를 0x000000ff\*512\*12 로 변경하였다. 이는 CS파일시스템을 이용 시 최대 파일 크기가 변하기 때문이다. 기존의 부분에 if else 문을 이용하여 T\_CS인 경우를 나눴다. 그리고 전역변수로 c, check를 만들고 이는 T\_CS파일 중간이 일반 파일 입력이 이루어질 경우를 인식하고 이에 따라 다음 bn 으로 조정을 위해 사용된다. check는 bmap에서 중간에 일반파일이 들어오는지 체크한다. 중간에 들어올경우 2값을 부여하고 이를 writei에서 이용하여

이렇게 중간에 들어올 경우 bn값을 조정해준다. b는 중간에 일반파일이 입력된 후 처음은 다음 블록으로 건너 뛰어야하기 때문에 사용한 변수이다. bmap에서는T\_CS인 경우 0x000000ff와 &연산으로 연속된 횟수를 찾고 , 배정된 블록넘버는 <<8을 통해 상위 3b만 이용할 수 있게 처리하였다. 0x000000ff와 &연산 값이 <0x000000ff인 경우에는 현재 아이노드의 direct블락에 +1을 해주었다.

int c=0;

int

writei(struct inode \*ip, char \*src, uint off, uint n)

{

  if(off + n > 1566720){//CS 최대크기로 변경

    return -1;

  }

  uint b=0,b\_n=0;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){

  if(ip->type == T\_CS){//cs일때 20202925

    if(check==2){

    if(c==0){

      b\_n=off/BSIZE/255+1;

      b=off/BSIZE/255;

      c=1;

    }

    else

      b\_n=b/BSIZE/255+1;

    }

    else

    b\_n=off/BSIZE/255;

    uint addr=bmap(ip, b\_n);

    bp = bread(ip->dev, addr); //addrs 배정

    m = min(n - tot, BSIZE - off%BSIZE);

    memmove(bp->data + off%BSIZE, src, m);

    log\_write(bp);

      brelse(bp);

    if(check==2)

    b=b+m;

  }

  else{

    if(check==1)

      check=2;

itrunc 에서도 비트 연산을 이용해 연속된 횟수와 해당 데이터블록의 시작주소를 추출해 연속된 수만큼 해제해준다.

  if(ip->type==T\_CS){ //cs일때 삭제

    if(ip->addrs[i]){

    uint n = ip->addrs[i] & 0x000000ff;

      uint b = ip->addrs[i] >> 8;

    for(k=0;k<n;k++) // n즉 하위1b의 값만큼 연속되어있으므로 반복으로 삭제

      bfree(ip->dev, b++);

        ip->addrs[i] = 0;

    }

  }

printinfo 함수는 sysfile.c에 시스템콜로 추가하여 파일포인터가 가리키는 아이노드를 이용하여 정보를 출력하였다.

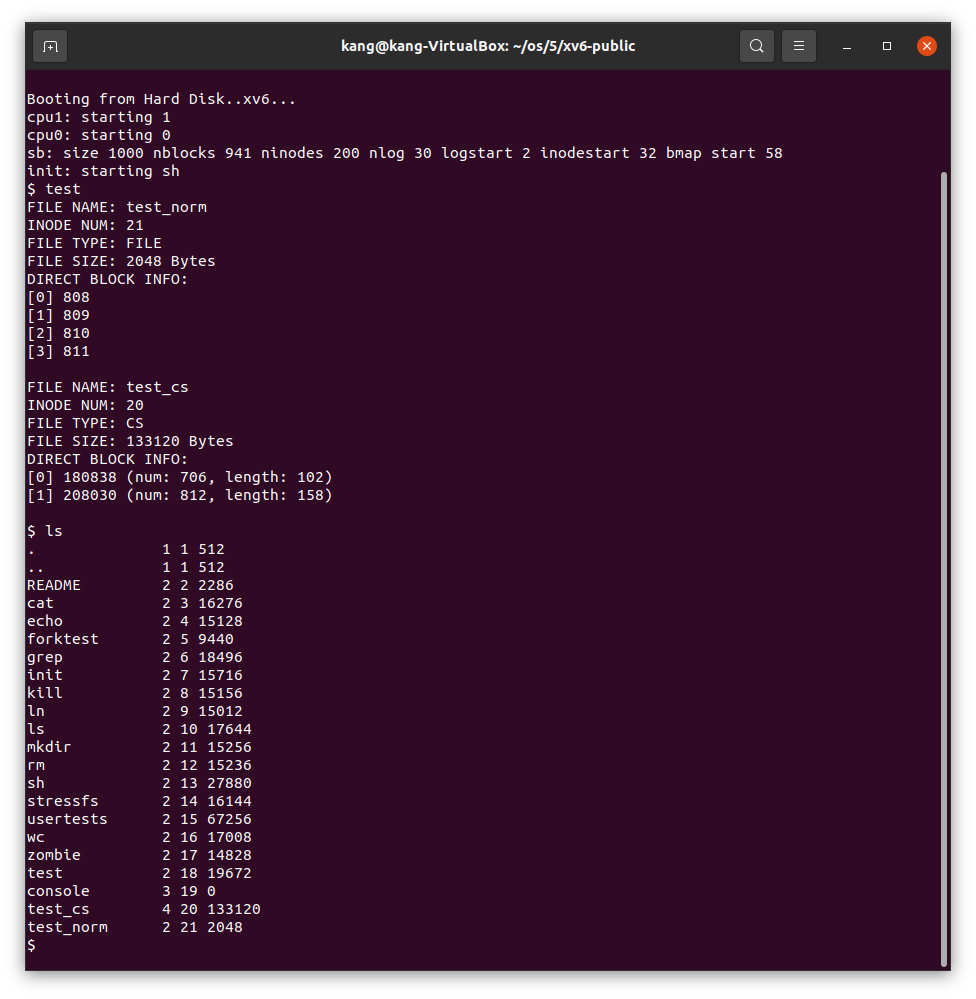
4.결과

test.c원본실행

텍스트이(가) 표시된 사진

자동 생성된 설명

test.c 주석 해제 실행 후 ls명령어까지



5. 소스코드

fcntl.h

#define O\_RDONLY  0x000

#define O\_WRONLY  0x001

#define O\_RDWR    0x002

#define O\_CREATE  0x200

#define O\_CS    0x020

stat.h

#define T\_DIR  1   // Directory

#define T\_FILE 2   // File

#define T\_DEV  3   // Device

#define T\_CS 4  // Continuous Sector based File

struct stat {

  short type;  // Type of file

  int dev;     // File system's disk device

  uint ino;    // Inode number

  short nlink; // Number of links to file

  uint size;   // Size of file in bytes

};

fs.c

// File system implementation.  Five layers:

//   + Blocks: allocator for raw disk blocks.

//   + Log: crash recovery for multi-step updates.

//   + Files: inode allocator, reading, writing, metadata.

//   + Directories: inode with special contents (list of other inodes!)

//   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.

//

// This file contains the low-level file system manipulation

// routines.  The (higher-level) system call implementations

// are in sysfile.c.

#include "types.h"

#include "defs.h"

#include "param.h"

#include "stat.h"

#include "mmu.h"

#include "proc.h"

#include "spinlock.h"

#include "sleeplock.h"

#include "fs.h"

#include "buf.h"

#include "file.h"

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

static void itrunc(struct inode\*);

// there should be one superblock per disk device, but we run with

// only one device

struct superblock sb;

// Read the super block.

void

readsb(int dev, struct superblock \*sb)

{

  struct buf \*bp;

  bp = bread(dev, 1);

  memmove(sb, bp->data, sizeof(\*sb));

  brelse(bp);

}

// Zero a block.

static void

bzero(int dev, int bno)

{

  struct buf \*bp;

  bp = bread(dev, bno);

  memset(bp->data, 0, BSIZE);

  log\_write(bp);

  brelse(bp);

}

// Blocks.

// 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);

        return b + bi;

      }

    }

    brelse(bp);

  }

  panic("balloc: out of blocks");

}

// Free a disk block.

static void

bfree(int dev, uint b)

{

  struct buf \*bp;

  int bi, m;

  bp = bread(dev, BBLOCK(b, sb));

  bi = b % BPB;

  m = 1 << (bi % 8);

  if((bp->data[bi/8] & m) == 0)

    panic("freeing free block");

  bp->data[bi/8] &= ~m;

  log\_write(bp);

  brelse(bp);

}

// Inodes.

//

// An inode describes a single unnamed file.

// The inode disk structure holds metadata: the file's type,

// its size, the number of links referring to it, and the

// list of blocks holding the file's content.

//

// The inodes are laid out sequentially on disk at

// sb.startinode. Each inode has a number, indicating its

// position on the disk.

//

// The kernel keeps a cache of in-use inodes in memory

// to provide a place for synchronizing access

// to inodes used by multiple processes. The cached

// inodes include book-keeping information that is

// not stored on disk: ip->ref and ip->valid.

//

// An inode and its in-memory representation go through a

// sequence of states before they can be used by the

// rest of the file system code.

//

// \* Allocation: an inode is allocated if its type (on disk)

//   is non-zero. ialloc() allocates, and iput() frees if

//   the reference and link counts have fallen to zero.

//

// \* Referencing in cache: an entry in the inode cache

//   is free if ip->ref is zero. Otherwise ip->ref tracks

//   the number of in-memory pointers to the entry (open

//   files and current directories). iget() finds or

//   creates a cache entry and increments its ref; iput()

//   decrements ref.

//

// \* Valid: the information (type, size, &c) in an inode

//   cache entry is only correct when ip->valid is 1.

//   ilock() reads the inode from

//   the disk and sets ip->valid, while iput() clears

//   ip->valid if ip->ref has fallen to zero.

//

// \* Locked: file system code may only examine and modify

//   the information in an inode and its content if it

//   has first locked the inode.

//

// Thus a typical sequence is:

//   ip = iget(dev, inum)

//   ilock(ip)

//   ... examine and modify ip->xxx ...

//   iunlock(ip)

//   iput(ip)

//

// ilock() is separate from iget() so that system calls can

// get a long-term reference to an inode (as for an open file)

// and only lock it for short periods (e.g., in read()).

// The separation also helps avoid deadlock and races during

// pathname lookup. iget() increments ip->ref so that the inode

// stays cached and pointers to it remain valid.

//

// Many internal file system functions expect the caller to

// have locked the inodes involved; this lets callers create

// multi-step atomic operations.

//

// The icache.lock spin-lock protects the allocation of icache

// entries. Since ip->ref indicates whether an entry is free,

// and ip->dev and ip->inum indicate which i-node an entry

// holds, one must hold icache.lock while using any of those fields.

//

// An ip->lock sleep-lock protects all ip-> fields other than ref,

// dev, and inum.  One must hold ip->lock in order to

// read or write that inode's ip->valid, ip->size, ip->type, &c.

struct {

  struct spinlock lock;

  struct inode inode[NINODE];

} icache;

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);

  cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\

 inodestart %d bmap start %d\n", sb.size, sb.nblocks,

          sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,

          sb.bmapstart);

}

static struct inode\* iget(uint dev, uint inum);

//PAGEBREAK!

// Allocate an inode on device dev.

// Mark it as allocated by  giving it type type.

// Returns an unlocked but allocated and referenced inode.

struct inode\*

ialloc(uint dev, short type)

{

  int inum;

  struct buf \*bp;

  struct dinode \*dip;

  for(inum = 1; inum < sb.ninodes; inum++){

    bp = bread(dev, IBLOCK(inum, sb));

    dip = (struct dinode\*)bp->data + inum%IPB;

    if(dip->type == 0){  // a free inode

      memset(dip, 0, sizeof(\*dip));

      dip->type = type;

      log\_write(bp);   // mark it allocated on the disk

      brelse(bp);

      return iget(dev, inum);

    }

    brelse(bp);

  }

  panic("ialloc: no inodes");

}

// Copy a modified in-memory inode to disk.

// Must be called after every change to an ip->xxx field

// that lives on disk, since i-node cache is write-through.

// Caller must hold ip->lock.

void

iupdate(struct inode \*ip)

{

  struct buf \*bp;

  struct dinode \*dip;

  bp = bread(ip->dev, IBLOCK(ip->inum, sb));

  dip = (struct dinode\*)bp->data + ip->inum%IPB;

  dip->type = ip->type;

  dip->major = ip->major;

  dip->minor = ip->minor;

  dip->nlink = ip->nlink;

  dip->size = ip->size;

  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));

  log\_write(bp);

  brelse(bp);

}

// Find the inode with number inum on device dev

// and return the in-memory copy. Does not lock

// the inode and does not read it from disk.

static struct inode\*

iget(uint dev, uint inum)

{

  struct inode \*ip, \*empty;

  acquire(&icache.lock);

  // Is the inode already cached?

  empty = 0;

  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){

    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){

      ip->ref++;

      release(&icache.lock);

      return ip;

    }

    if(empty == 0 && ip->ref == 0)    // Remember empty slot.

      empty = ip;

  }

  // Recycle an inode cache entry.

  if(empty == 0)

    panic("iget: no inodes");

  ip = empty;

  ip->dev = dev;

  ip->inum = inum;

  ip->ref = 1;

  ip->valid = 0;

  release(&icache.lock);

  return ip;

}

// Increment reference count for ip.

// Returns ip to enable ip = idup(ip1) idiom.

struct inode\*

idup(struct inode \*ip)

{

  acquire(&icache.lock);

  ip->ref++;

  release(&icache.lock);

  return ip;

}

// Lock the given inode.

// Reads the inode from disk if necessary.

void

ilock(struct inode \*ip)

{

  struct buf \*bp;

  struct dinode \*dip;

  if(ip == 0 || ip->ref < 1)

    panic("ilock");

  acquiresleep(&ip->lock);

  if(ip->valid == 0){

    bp = bread(ip->dev, IBLOCK(ip->inum, sb));

    dip = (struct dinode\*)bp->data + ip->inum%IPB;

    ip->type = dip->type;

    ip->major = dip->major;

    ip->minor = dip->minor;

    ip->nlink = dip->nlink;

    ip->size = dip->size;

    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));

    brelse(bp);

    ip->valid = 1;

    if(ip->type == 0)

      panic("ilock: no type");

  }

}

// Unlock the given inode.

void

iunlock(struct inode \*ip)

{

  if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)

    panic("iunlock");

  releasesleep(&ip->lock);

}

// Drop a reference to an in-memory inode.

// If that was the last reference, the inode cache entry can

// be recycled.

// If that was the last reference and the inode has no links

// to it, free the inode (and its content) on disk.

// All calls to iput() must be inside a transaction in

// case it has to free the inode.

void

iput(struct inode \*ip)

{

  acquiresleep(&ip->lock);

  if(ip->valid && ip->nlink == 0){

    acquire(&icache.lock);

    int r = ip->ref;

    release(&icache.lock);

    if(r == 1){

      // inode has no links and no other references: truncate and free.

      itrunc(ip);

      ip->type = 0;

      iupdate(ip);

      ip->valid = 0;

    }

  }

  releasesleep(&ip->lock);

  acquire(&icache.lock);

  ip->ref--;

  release(&icache.lock);

}

// Common idiom: unlock, then put.

void

iunlockput(struct inode \*ip)

{

  iunlock(ip);

  iput(ip);

}

//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.

uint check=0;

int new=0;

static uint

bmap(struct inode \*ip, uint bn)

{

  uint addr, \*a;

  struct buf \*bp;

  //20202925\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  // 0x00000011로 &연산 < 0x00000011 일 때 배정

  if(ip->type==T\_CS){//cs로 플래그 있을 때 다른 배정

  if((addr = ip->addrs[bn])  == 0){//비었으면

    addr= balloc(ip->dev);

    ip->addrs[bn] =(addr<<8)+(addr&0x00000000)+1;//해당 주소로 할당 상위 3b이용 1b는 할당 데이터블록 개수

    new=1;

  }

  else if(((addr&0x000000ff)<0x000000ff) && new==1){

    //최대 할당 가능 블록수 보다 작을 때

  //  cprintf("aa\n");

    ip->addrs[bn] =addr+1;//해당 주소로 할당 상위 3b이용, 하위 1b연속수

    addr=balloc(ip->dev);

  }

  else{

    if(new==0){

      addr=addr>>8;

      }

  }

    //20202925\*\*\*\*\*\*

  if(bn>=NDIRECT)

    panic("bimap: CS out of range");

  if(check==0)

    check=1;

  return addr;

  }

  else if(bn < NDIRECT){

  if((addr = ip->addrs[bn]) == 0)//비었으면

      ip->addrs[bn] = addr = balloc(ip->dev);//해당 주소로 할당

    return addr;

  }

  //cprintf(" ---- %d\n",bn);

  bn -= NDIRECT;

  if(bn < NINDIRECT){

    // Load indirect block, allocating if necessary.

    if((addr = ip->addrs[NDIRECT]) == 0)

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

    bp = bread(ip->dev, addr);

    a = (uint\*)bp->data;

    if((addr = a[bn]) == 0){

      a[bn] = addr = balloc(ip->dev);

      log\_write(bp);

    }

    brelse(bp);

    return addr;

  }

  panic("bmap: out of range");

}

// 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).

//삭제 20202925

static void

itrunc(struct inode \*ip)

{

  int i, j,k;

  struct buf \*bp;

  uint \*a;

  uint b, n;

  for(i = 0; i < NDIRECT; i++){

  if(ip->type==T\_CS){ //cs일때 삭제

    if(ip->addrs[i]){

    n = ip->addrs[i] & 0x000000ff;

      b = ip->addrs[i] >> 8;

//    cprintf("b: %d, n: %d\n",b,n);

    for(k=0;k<n;k++) // n즉 하위1b의 값만큼 연속되어있으므로 반복으로 삭제

      bfree(ip->dev, b++);

        ip->addrs[i] = 0;

    }

  }

  else{

//    cprintf("?????\n");

    if(ip->addrs[i]){

        bfree(ip->dev, ip->addrs[i]);

        ip->addrs[i] = 0;

      }

  }

  }

  if(ip->addrs[NDIRECT]){

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

    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]);

    ip->addrs[NDIRECT] = 0;

  }

  ip->size = 0;

  iupdate(ip);

}

// Copy stat information from inode.

// Caller must hold ip->lock.

void

stati(struct inode \*ip, struct stat \*st)

{

  st->dev = ip->dev;

  st->ino = ip->inum;

  st->type = ip->type;

  st->nlink = ip->nlink;

  st->size = ip->size;

}

//PAGEBREAK!

// Read data from inode.

// Caller must hold ip->lock.

int

readi(struct inode \*ip, char \*dst, 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].read)

      return -1;

    return devsw[ip->major].read(ip, dst, n);

  }

  if(off > ip->size || off + n < off)

    return -1;

  if(off + n > ip->size)

    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){

    bp = bread(ip->dev, bmap(ip, off/BSIZE));

    m = min(n - tot, BSIZE - off%BSIZE);

    memmove(dst, bp->data + off%BSIZE, m);

    brelse(bp);

  }

  return n;

}

// PAGEBREAK!

// Write data to inode.

// Caller must hold ip->lock.

uint p\_off=0; //중간에 파일 작성시 다음 블록 할당위해 기존 off저장

int c=0;

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);

  }

  if(off > ip->size || off + n < off){

    return -1;

  }

  if(off + n > 1566720){//CS 최대크기로 변경

    return -1;

  }

  uint b=0,b\_n=0;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){

  if(ip->type == T\_CS){//cs일때 20202925

    if(check==2){

    if(c==0){

      b\_n=off/BSIZE/255+1;

      b=off/BSIZE/255;

      c=1;

    }

    else

      b\_n=b/BSIZE/255+1;

    }

    else

    b\_n=off/BSIZE/255;

    uint addr=bmap(ip, b\_n);

    bp = bread(ip->dev, addr); //addrs 배정

    m = min(n - tot, BSIZE - off%BSIZE);

    memmove(bp->data + off%BSIZE, src, m);

    log\_write(bp);

      brelse(bp);

    if(check==2)

    b=b+m;

  }

  else{

    if(check==1)

      check=2;

    bp = bread(ip->dev, bmap(ip, off/BSIZE)); //addrs 배정

    m = min(n - tot, BSIZE - off%BSIZE);

    memmove(bp->data + off%BSIZE, src, m);

      log\_write(bp);

      brelse(bp);

  }

  }

  if(n > 0 && off > ip->size){

    ip->size = off;

    iupdate(ip);

  }

  return n;

}

//PAGEBREAK!

// Directories

int

namecmp(const char \*s, const char \*t)

{

  return strncmp(s, t, DIRSIZ);

}

// Look for a directory entry in a directory.

// If found, set \*poff to byte offset of entry.

struct inode\*

dirlookup(struct inode \*dp, char \*name, uint \*poff)

{

  uint off, inum;

  struct dirent de;

  if(dp->type != T\_DIR)

    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){

    if(readi(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

      panic("dirlookup read");

    if(de.inum == 0)

      continue;

    if(namecmp(name, de.name) == 0){

      // entry matches path element

      if(poff)

        \*poff = off;

      inum = de.inum;

      return iget(dp->dev, inum);

    }

  }

  return 0;

}

// Write a new directory entry (name, inum) into the directory dp.

int

dirlink(struct inode \*dp, char \*name, uint inum)

{

  int off;

  struct dirent de;

  struct inode \*ip;

  // Check that name is not present.

  if((ip = dirlookup(dp, name, 0)) != 0){

    iput(ip);

    return -1;

  }

  // Look for an empty dirent.

  for(off = 0; off < dp->size; off += sizeof(de)){

    if(readi(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

      panic("dirlink read");

    if(de.inum == 0)

      break;

  }

  strncpy(de.name, name, DIRSIZ);

  de.inum = inum;

  if(writei(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

    panic("dirlink");

  return 0;

}

//PAGEBREAK!

// Paths

// Copy the next path element from path into name.

// Return a pointer to the element following the copied one.

// The returned path has no leading slashes,

// so the caller can check \*path=='\0' to see if the name is the last one.

// If no name to remove, return 0.

//

// Examples:

//   skipelem("a/bb/c", name) = "bb/c", setting name = "a"

//   skipelem("///a//bb", name) = "bb", setting name = "a"

//   skipelem("a", name) = "", setting name = "a"

//   skipelem("", name) = skipelem("////", name) = 0

//

static char\*

skipelem(char \*path, char \*name)

{

  char \*s;

  int len;

  while(\*path == '/')

    path++;

  if(\*path == 0)

    return 0;

  s = path;

  while(\*path != '/' && \*path != 0)

    path++;

  len = path - s;

  if(len >= DIRSIZ)

    memmove(name, s, DIRSIZ);

  else {

    memmove(name, s, len);

    name[len] = 0;

  }

  while(\*path == '/')

    path++;

  return path;

}

// Look up and return the inode for a path name.

// If parent != 0, return the inode for the parent and copy the final

// path element into name, which must have room for DIRSIZ bytes.

// Must be called inside a transaction since it calls iput().

static struct inode\*

namex(char \*path, int nameiparent, char \*name)

{

  struct inode \*ip, \*next;

  if(\*path == '/')

    ip = iget(ROOTDEV, ROOTINO);

  else

    ip = idup(myproc()->cwd);

  while((path = skipelem(path, name)) != 0){

    ilock(ip);

    if(ip->type != T\_DIR){

      iunlockput(ip);

      return 0;

    }

    if(nameiparent && \*path == '\0'){

      // Stop one level early.

      iunlock(ip);

      return ip;

    }

    if((next = dirlookup(ip, name, 0)) == 0){

      iunlockput(ip);

      return 0;

    }

    iunlockput(ip);

    ip = next;

  }

  if(nameiparent){

    iput(ip);

    return 0;

  }

  return ip;

}

struct inode\*

namei(char \*path)

{

  char name[DIRSIZ];

  return namex(path, 0, name);

}

struct inode\*

nameiparent(char \*path, char \*name)

{

  return namex(path, 1, name);

}

syscall.c

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

#include "x86.h"

#include "syscall.h"

// User code makes a system call with INT T\_SYSCALL.

// System call number in %eax.

// Arguments on the stack, from the user call to the C

// library system call function. The saved user %esp points

// to a saved program counter, and then the first argument.

// Fetch the int at addr from the current process.

int

fetchint(uint addr, int \*ip)

{

  struct proc \*curproc = myproc();

  if(addr >= curproc->sz || addr+4 > curproc->sz)

    return -1;

  \*ip = \*(int\*)(addr);

  return 0;

}

// Fetch the nul-terminated string at addr from the current process.

// Doesn't actually copy the string - just sets \*pp to point at it.

// Returns length of string, not including nul.

int

fetchstr(uint addr, char \*\*pp)

{

  char \*s, \*ep;

  struct proc \*curproc = myproc();

  if(addr >= curproc->sz)

    return -1;

  \*pp = (char\*)addr;

  ep = (char\*)curproc->sz;

  for(s = \*pp; s < ep; s++){

    if(\*s == 0)

      return s - \*pp;

  }

  return -1;

}

// Fetch the nth 32-bit system call argument.

int

argint(int n, int \*ip)

{

  return fetchint((myproc()->tf->esp) + 4 + 4\*n, ip);

}

// Fetch the nth word-sized system call argument as a pointer

// to a block of memory of size bytes.  Check that the pointer

// lies within the process address space.

int

argptr(int n, char \*\*pp, int size)

{

  int i;

  struct proc \*curproc = myproc();

  if(argint(n, &i) < 0)

    return -1;

  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)

    return -1;

  \*pp = (char\*)i;

  return 0;

}

// Fetch the nth word-sized system call argument as a string pointer.

// Check that the pointer is valid and the string is nul-terminated.

// (There is no shared writable memory, so the string can't change

// between this check and being used by the kernel.)

int

argstr(int n, char \*\*pp)

{

  int addr;

  if(argint(n, &addr) < 0)

    return -1;

  return fetchstr(addr, pp);

}

extern int sys\_chdir(void);

extern int sys\_close(void);

extern int sys\_dup(void);

extern int sys\_exec(void);

extern int sys\_exit(void);

extern int sys\_fork(void);

extern int sys\_fstat(void);

extern int sys\_getpid(void);

extern int sys\_kill(void);

extern int sys\_link(void);

extern int sys\_mkdir(void);

extern int sys\_mknod(void);

extern int sys\_open(void);

extern int sys\_pipe(void);

extern int sys\_read(void);

extern int sys\_sbrk(void);

extern int sys\_sleep(void);

extern int sys\_unlink(void);

extern int sys\_wait(void);

extern int sys\_write(void);

extern int sys\_uptime(void);

extern int sys\_printinfo(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\_printinfo]   sys\_printinfo,

};

void

syscall(void)

{

  int num;

  struct proc \*curproc = myproc();

  num = curproc->tf->eax;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    curproc->tf->eax = syscalls[num]();

  } else {

    cprintf("%d %s: unknown sys call %d\n",

            curproc->pid, curproc->name, num);

    curproc->tf->eax = -1;

  }

}

syscall.h

// System call numbers

#define SYS\_fork    1

#define SYS\_exit    2

#define SYS\_wait    3

#define SYS\_pipe    4

#define SYS\_read    5

#define SYS\_kill    6

#define SYS\_exec    7

#define SYS\_fstat   8

#define SYS\_chdir   9

#define SYS\_dup    10

#define SYS\_getpid 11

#define SYS\_sbrk   12

#define SYS\_sleep  13

#define SYS\_uptime 14

#define SYS\_open   15

#define SYS\_write  16

#define SYS\_mknod  17

#define SYS\_unlink 18

#define SYS\_link   19

#define SYS\_mkdir  20

#define SYS\_close  21

#define SYS\_printinfo 22

sysfile.c

//

// File-system system calls.

// Mostly argument checking, since we don't trust

// user code, and calls into file.c and fs.c.

//

#include "types.h"

#include "defs.h"

#include "param.h"

#include "stat.h"

#include "mmu.h"

#include "proc.h"

#include "fs.h"

#include "spinlock.h"

#include "sleeplock.h"

#include "file.h"

#include "fcntl.h"

// Fetch the nth word-sized system call argument as a file descriptor

// and return both the descriptor and the corresponding struct file.

static int

argfd(int n, int \*pfd, struct file \*\*pf)

{

  int fd;

  struct file \*f;

  if(argint(n, &fd) < 0)

    return -1;

  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)

    return -1;

  if(pfd)

    \*pfd = fd;

  if(pf)

    \*pf = f;

  return 0;

}

// Allocate a file descriptor for the given file.

// Takes over file reference from caller on success.

static int

fdalloc(struct file \*f)

{

  int fd;

  struct proc \*curproc = myproc();

  for(fd = 0; fd < NOFILE; fd++){

    if(curproc->ofile[fd] == 0){

      curproc->ofile[fd] = f;

      return fd;

    }

  }

  return -1;

}

int

sys\_dup(void)

{

  struct file \*f;

  int fd;

  if(argfd(0, 0, &f) < 0)

    return -1;

  if((fd=fdalloc(f)) < 0)

    return -1;

  filedup(f);

  return fd;

}

int

sys\_read(void)

{

  struct file \*f;

  int n;

  char \*p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)

    return -1;

  return fileread(f, p, n);

}

int

sys\_write(void)

{

  struct file \*f;

  int n;

  char \*p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)

    return -1;

  return filewrite(f, p, n);

}

int

sys\_close(void)

{

  int fd;

  struct file \*f;

  if(argfd(0, &fd, &f) < 0)

    return -1;

  myproc()->ofile[fd] = 0;

  fileclose(f);

  return 0;

}

int

sys\_fstat(void)

{

  struct file \*f;

  struct stat \*st;

  if(argfd(0, 0, &f) < 0 || argptr(1, (void\*)&st, sizeof(\*st)) < 0)

    return -1;

  return filestat(f, st);

}

// Create the path new as a link to the same inode as old.

int

sys\_link(void)

{

  char name[DIRSIZ], \*new, \*old;

  struct inode \*dp, \*ip;

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

    return -1;

  begin\_op();

  if((ip = namei(old)) == 0){

    end\_op();

    return -1;

  }

  ilock(ip);

  if(ip->type == T\_DIR){

    iunlockput(ip);

    end\_op();

    return -1;

  }

  ip->nlink++;

  iupdate(ip);

  iunlock(ip);

  if((dp = nameiparent(new, name)) == 0)

    goto bad;

  ilock(dp);

  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){

    iunlockput(dp);

    goto bad;

  }

  iunlockput(dp);

  iput(ip);

  end\_op();

  return 0;

bad:

  ilock(ip);

  ip->nlink--;

  iupdate(ip);

  iunlockput(ip);

  end\_op();

  return -1;

}

// Is the directory dp empty except for "." and ".." ?

static int

isdirempty(struct inode \*dp)

{

  int off;

  struct dirent de;

  for(off=2\*sizeof(de); off<dp->size; off+=sizeof(de)){

    if(readi(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

      panic("isdirempty: readi");

    if(de.inum != 0)

      return 0;

  }

  return 1;

}

//PAGEBREAK!

int

sys\_unlink(void)

{

  struct inode \*ip, \*dp;

  struct dirent de;

  char name[DIRSIZ], \*path;

  uint off;

  if(argstr(0, &path) < 0)

    return -1;

  begin\_op();

  if((dp = nameiparent(path, name)) == 0){

    end\_op();

    return -1;

  }

  ilock(dp);

  // Cannot unlink "." or "..".

  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)

    goto bad;

  if((ip = dirlookup(dp, name, &off)) == 0)

    goto bad;

  ilock(ip);

  if(ip->nlink < 1)

    panic("unlink: nlink < 1");

  if(ip->type == T\_DIR && !isdirempty(ip)){

    iunlockput(ip);

    goto bad;

  }

  memset(&de, 0, sizeof(de));

  if(writei(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

    panic("unlink: writei");

  if(ip->type == T\_DIR){

    dp->nlink--;

    iupdate(dp);

  }

  iunlockput(dp);

  ip->nlink--;

  iupdate(ip);

  iunlockput(ip);

  end\_op();

  return 0;

bad:

  iunlockput(dp);

  end\_op();

  return -1;

}

static struct inode\*

create(char \*path, short type, short major, short minor)

{

  struct inode \*ip, \*dp;

  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)

    return 0;

  ilock(dp);

  if((ip = dirlookup(dp, name, 0)) != 0){

//  cprintf("1111\n");

    iunlockput(dp);

//  cprintf("22222\n");

    ilock(ip);

//  cprintf("333\n");

    if((type == T\_FILE && ip->type == T\_FILE )|| (type==T\_CS && ip->type==T\_CS))//20202925

  {

      return ip;

  }

//  cprintf("5555\n");

    iunlockput(ip);

    return 0;

  }

  if((ip = ialloc(dp->dev, type)) == 0)//이부분

    panic("create: ialloc");

  ilock(ip);

  ip->major = major;

  ip->minor = minor;

  ip->nlink = 1;

  iupdate(ip);

  if(type == T\_DIR){  // Create . and .. entries.

    dp->nlink++;  // for ".."

    iupdate(dp);

    // No ip->nlink++ for ".": avoid cyclic ref count.

    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)

      panic("create dots");

  }

  if(dirlink(dp, name, ip->inum) < 0)

    panic("create: dirlink");

  iunlockput(dp);

  return ip;

}

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;

  begin\_op();

  //20202925 수정 플래그 받는 부분

//  if(omode & O\_CS){

//    ip = create(path, T\_CS, 0, 0);

//    if(ip==0){

//      end\_op();

//      return -1;

//    }

  //}

  if(omode & O\_CREATE){

  if(omode & O\_CS)

    ip=create(path,T\_CS,0,0);

  else

    ip = create(path, T\_FILE, 0, 0);

    if(ip == 0){

      end\_op();

//    cprintf("aaa\n");

      return -1;

    }

  }

  else {

    if((ip = namei(path)) == 0){

      end\_op();

      return -1;

    }

    ilock(ip);

    if(ip->type == T\_DIR && omode != O\_RDONLY){

      iunlockput(ip);

      end\_op();

      return -1;

    }

  }

 // cprintf("!!!!!!\n");

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){

    if(f)

      fileclose(f);

    iunlockput(ip);

    end\_op();

    return -1;

  }

  iunlock(ip);

  end\_op();

  //cprintf("size %d\n",ip->size);

  f->type = FD\_INODE;

  f->ip = ip;

  f->off = 0;

  f->readable = !(omode & O\_WRONLY);

  f->writable = (omode & O\_WRONLY) || (omode & O\_RDWR);

  return fd;

}

int

sys\_mkdir(void)

{

  char \*path;

  struct inode \*ip;

  begin\_op();

  if(argstr(0, &path) < 0 || (ip = create(path, T\_DIR, 0, 0)) == 0){

    end\_op();

    return -1;

  }

  iunlockput(ip);

  end\_op();

  return 0;

}

int

sys\_mknod(void)

{

  struct inode \*ip;

  char \*path;

  int major, minor;

  begin\_op();

  if((argstr(0, &path)) < 0 ||

     argint(1, &major) < 0 ||

     argint(2, &minor) < 0 ||

     (ip = create(path, T\_DEV, major, minor)) == 0){

    end\_op();

    return -1;

  }

  iunlockput(ip);

  end\_op();

  return 0;

}

int

sys\_chdir(void)

{

  char \*path;

  struct inode \*ip;

  struct proc \*curproc = myproc();

  begin\_op();

  if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){

    end\_op();

    return -1;

  }

  ilock(ip);

  if(ip->type != T\_DIR){

    iunlockput(ip);

    end\_op();

    return -1;

  }

  iunlock(ip);

  iput(curproc->cwd);

  end\_op();

  curproc->cwd = ip;

  return 0;

}

int

sys\_exec(void)

{

  char \*path, \*argv[MAXARG];

  int i;

  uint uargv, uarg;

  if(argstr(0, &path) < 0 || argint(1, (int\*)&uargv) < 0){

    return -1;

  }

  memset(argv, 0, sizeof(argv));

  for(i=0;; i++){

    if(i >= NELEM(argv))

      return -1;

    if(fetchint(uargv+4\*i, (int\*)&uarg) < 0)

      return -1;

    if(uarg == 0){

      argv[i] = 0;

      break;

    }

    if(fetchstr(uarg, &argv[i]) < 0)

      return -1;

  }

  return exec(path, argv);

}

int

sys\_pipe(void)

{

  int \*fd;

  struct file \*rf, \*wf;

  int fd0, fd1;

  if(argptr(0, (void\*)&fd, 2\*sizeof(fd[0])) < 0)

    return -1;

  if(pipealloc(&rf, &wf) < 0)

    return -1;

  fd0 = -1;

  if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){

    if(fd0 >= 0)

      myproc()->ofile[fd0] = 0;

    fileclose(rf);

    fileclose(wf);

    return -1;

  }

  fd[0] = fd0;

  fd[1] = fd1;

  return 0;

}

int

sys\_printinfo(void)

{

  struct file \*f;

  char \*fname;

  int i;

  if(argfd(0, 0, &f) < 0 || argptr(1, &fname, sizeof(fname)) < 0)

    return -1;

  cprintf("FILE NAME: %s\n",fname);

  cprintf("INODE NUM: %d\n", f->ip->inum);

  if(f->ip->type==T\_CS){

    cprintf("FILE TYPE: CS\n");

    cprintf("FILE SIZE: %d Bytes\n", f->ip->size);

    cprintf("DIRECT BLOCK INFO:\n");

    for(i=0;i<NDIRECT;i++){

    if(f->ip->addrs[i]>0)

      cprintf("[%d] %d (num: %d, length: %d)\n",i, f->ip->addrs[i],  f->ip->addrs[i] >> 8, f->ip->addrs[i] & 0x000000ff);

    }

  }

  else{

    cprintf("FILE TYPE: FILE\n");

      cprintf("FILE SIZE: %d Bytes\n", f->ip->size);

      cprintf("DIRECT BLOCK INFO:\n");

    for(i=0;i<NDIRECT;i++){

    if(f->ip->addrs[i]>0)

      cprintf("[%d] %d\n",i, f->ip->addrs[i]);

    }

  }

  cprintf("\n");

  return 0;

}

user.h

struct stat;

struct rtcdate;

// system calls

int fork(void);

int exit(void) \_\_attribute\_\_((noreturn));

int wait(void);

int pipe(int\*);

int write(int, const void\*, int);

int read(int, void\*, int);

int close(int);

int kill(int);

int exec(char\*, char\*\*);

int open(const char\*, int);

int mknod(const char\*, short, short);

int unlink(const char\*);

int fstat(int fd, struct stat\*);

int link(const char\*, const char\*);

int mkdir(const char\*);

int chdir(const char\*);

int dup(int);

int getpid(void);

char\* sbrk(int);

int sleep(int);

int uptime(void);

void printinfo(int, char\*);

// ulib.c

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

char\* strcpy(char\*, const char\*);

void \*memmove(void\*, const void\*, int);

char\* strchr(const char\*, char c);

int strcmp(const char\*, const char\*);

void printf(int, const char\*, ...);

char\* gets(char\*, int max);

uint strlen(const char\*);

void\* memset(void\*, int, uint);

void\* malloc(uint);

void free(void\*);

int atoi(const char\*);

**usys.S**

#include "syscall.h"

#include "traps.h"

#define SYSCALL(name) \

.globl name; \

name: \

movl $SYS\_ ## name, %eax; \

int $T\_SYSCALL; \

ret

SYSCALL(fork)

SYSCALL(exit)

SYSCALL(wait)

SYSCALL(pipe)

SYSCALL(read)

SYSCALL(write)

SYSCALL(close)

SYSCALL(kill)

SYSCALL(exec)

SYSCALL(open)

SYSCALL(mknod)

SYSCALL(unlink)

SYSCALL(fstat)

SYSCALL(link)

SYSCALL(mkdir)

SYSCALL(chdir)

SYSCALL(dup)

SYSCALL(getpid)

SYSCALL(sbrk)

SYSCALL(sleep)

SYSCALL(uptime)

SYSCALL(printinfo)

**Makefile**

OBJS = \

    bio.o\

    console.o\

    exec.o\

    file.o\

    fs.o\

    ide.o\

    ioapic.o\

    kalloc.o\

    kbd.o\

    lapic.o\

    log.o\

    main.o\

    mp.o\

    picirq.o\

    pipe.o\

    proc.o\

    sleeplock.o\

    spinlock.o\

    string.o\

    swtch.o\

    syscall.o\

    sysfile.o\

    sysproc.o\

    trapasm.o\

    trap.o\

    uart.o\

    vectors.o\

    vm.o\

# Cross-compiling (e.g., on Mac OS X)

# TOOLPREFIX = i386-jos-elf

# Using native tools (e.g., on X86 Linux)

#TOOLPREFIX =

# Try to infer the correct TOOLPREFIX if not set

ifndef TOOLPREFIX

TOOLPREFIX := $(shell if i386-jos-elf-objdump -i 2>&1 | grep '^elf32-i386$$' >/dev/null 2>&1; \

    then echo 'i386-jos-elf-'; \

    elif objdump -i 2>&1 | grep 'elf32-i386' >/dev/null 2>&1; \

    then echo ''; \

    else echo "\*\*\*" 1>&2; \

    echo "\*\*\* Error: Couldn't find an i386-\*-elf version of GCC/binutils." 1>&2; \

    echo "\*\*\* Is the directory with i386-jos-elf-gcc in your PATH?" 1>&2; \

    echo "\*\*\* If your i386-\*-elf toolchain is installed with a command" 1>&2; \

    echo "\*\*\* prefix other than 'i386-jos-elf-', set your TOOLPREFIX" 1>&2; \

    echo "\*\*\* environment variable to that prefix and run 'make' again." 1>&2; \

    echo "\*\*\* To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \

    echo "\*\*\*" 1>&2; exit 1; fi)

endif

# If the makefile can't find QEMU, specify its path here

# QEMU = qemu-system-i386

# Try to infer the correct QEMU

ifndef QEMU

QEMU = $(shell if which qemu > /dev/null; \

    then echo qemu; exit; \

    elif which qemu-system-i386 > /dev/null; \

    then echo qemu-system-i386; exit; \

    elif which qemu-system-x86\_64 > /dev/null; \

    then echo qemu-system-x86\_64; exit; \

    else \

    qemu=/Applications/Q.app/Contents/MacOS/i386-softmmu.app/Contents/MacOS/i386-softmmu; \

    if test -x $$qemu; then echo $$qemu; exit; fi; fi; \

    echo "\*\*\*" 1>&2; \

    echo "\*\*\* Error: Couldn't find a working QEMU executable." 1>&2; \

    echo "\*\*\* Is the directory containing the qemu binary in your PATH" 1>&2; \

    echo "\*\*\* or have you tried setting the QEMU variable in Makefile?" 1>&2; \

    echo "\*\*\*" 1>&2; exit 1)

endif

CC = $(TOOLPREFIX)gcc

AS = $(TOOLPREFIX)gas

LD = $(TOOLPREFIX)ld

OBJCOPY = $(TOOLPREFIX)objcopy

OBJDUMP = $(TOOLPREFIX)objdump

CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer

CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)

ASFLAGS = -m32 -gdwarf-2 -Wa,-divide

# FreeBSD ld wants ``elf\_i386\_fbsd''

LDFLAGS += -m $(shell $(LD) -V | grep elf\_i386 2>/dev/null | head -n 1)

# Disable PIE when possible (for Ubuntu 16.10 toolchain)

ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]no-pie'),)

CFLAGS += -fno-pie -no-pie

endif

ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]nopie'),)

CFLAGS += -fno-pie -nopie

endif

xv6.img: bootblock kernel

    dd if=/dev/zero of=xv6.img count=10000

    dd if=bootblock of=xv6.img conv=notrunc

    dd if=kernel of=xv6.img seek=1 conv=notrunc

xv6memfs.img: bootblock kernelmemfs

    dd if=/dev/zero of=xv6memfs.img count=10000

    dd if=bootblock of=xv6memfs.img conv=notrunc

    dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc

bootblock: bootasm.S bootmain.c

    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c

    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o

    $(OBJDUMP) -S bootblock.o > bootblock.asm

    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock

    ./sign.pl bootblock

entryother: entryother.S

    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o

    $(OBJCOPY) -S -O binary -j .text bootblockother.o entryother

    $(OBJDUMP) -S bootblockother.o > entryother.asm

initcode: initcode.S

    $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o

    $(OBJCOPY) -S -O binary initcode.out initcode

    $(OBJDUMP) -S initcode.o > initcode.asm

kernel: $(OBJS) entry.o entryother initcode kernel.ld

    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother

    $(OBJDUMP) -S kernel > kernel.asm

    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > kernel.sym

# kernelmemfs is a copy of kernel that maintains the

# disk image in memory instead of writing to a disk.

# This is not so useful for testing persistent storage or

# exploring disk buffering implementations, but it is

# great for testing the kernel on real hardware without

# needing a scratch disk.

MEMFSOBJS = $(filter-out ide.o,$(OBJS)) memide.o

kernelmemfs: $(MEMFSOBJS) entry.o entryother initcode kernel.ld fs.img

    $(LD) $(LDFLAGS) -T kernel.ld -o kernelmemfs entry.o  $(MEMFSOBJS) -b binary initcode entryother fs.img

    $(OBJDUMP) -S kernelmemfs > kernelmemfs.asm

    $(OBJDUMP) -t kernelmemfs | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > kernelmemfs.sym

tags: $(OBJS) entryother.S \_init

    etags \*.S \*.c

vectors.S: vectors.pl

    ./vectors.pl > vectors.S

ULIB = ulib.o usys.o printf.o umalloc.o

\_%: %.o $(ULIB)

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^

    $(OBJDUMP) -S $@ > $\*.asm

    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $\*.sym

\_forktest: forktest.o $(ULIB)

    # forktest has less library code linked in - needs to be small

    # in order to be able to max out the proc table.

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o \_forktest forktest.o ulib.o usys.o

    $(OBJDUMP) -S \_forktest > forktest.asm

mkfs: mkfs.c fs.h

    gcc -Werror -Wall -o mkfs mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so

# that disk image changes after first build are persistent until clean.  More

# details:

# http://www.gnu.org/software/make/manual/html\_node/Chained-Rules.html

.PRECIOUS: %.o

UPROGS=\

    \_cat\

    \_echo\

    \_forktest\

    \_grep\

    \_init\

    \_kill\

    \_ln\

    \_ls\

    \_mkdir\

    \_rm\

    \_sh\

    \_stressfs\

    \_usertests\

    \_wc\

    \_zombie\

    \_test\

fs.img: mkfs README $(UPROGS)

    ./mkfs fs.img README $(UPROGS)

-include \*.d

clean:

    rm -f \*.tex \*.dvi \*.idx \*.aux \*.log \*.ind \*.ilg \

    \*.o \*.d \*.asm \*.sym vectors.S bootblock entryother \

    initcode initcode.out kernel xv6.img fs.img kernelmemfs \

    xv6memfs.img mkfs .gdbinit \

    $(UPROGS)

# make a printout

FILES = $(shell grep -v '^\#' runoff.list)

PRINT = runoff.list runoff.spec README toc.hdr toc.ftr $(FILES)

xv6.pdf: $(PRINT)

    ./runoff

    ls -l xv6.pdf

print: xv6.pdf

# run in emulators

bochs : fs.img xv6.img

    if [ ! -e .bochsrc ]; then ln -s dot-bochsrc .bochsrc; fi

    bochs -q

# try to generate a unique GDB port

GDBPORT = $(shell expr `id -u` % 5000 + 25000)

# QEMU's gdb stub command line changed in 0.11

QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \

    then echo "-gdb tcp::$(GDBPORT)"; \

    else echo "-s -p $(GDBPORT)"; fi)

ifndef CPUS

CPUS := 2

endif

QEMUOPTS = -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp $(CPUS) -m 512 $(QEMUEXTRA)

qemu: fs.img xv6.img

    $(QEMU) -serial mon:stdio $(QEMUOPTS)

qemu-memfs: xv6memfs.img

    $(QEMU) -drive file=xv6memfs.img,index=0,media=disk,format=raw -smp $(CPUS) -m 256

qemu-nox: fs.img xv6.img

    $(QEMU) -nographic $(QEMUOPTS)

.gdbinit: .gdbinit.tmpl

    sed "s/localhost:1234/localhost:$(GDBPORT)/" < $^ > $@

qemu-gdb: fs.img xv6.img .gdbinit

    @echo "\*\*\* Now run 'gdb'." 1>&2

    $(QEMU) -serial mon:stdio $(QEMUOPTS) -S $(QEMUGDB)

qemu-nox-gdb: fs.img xv6.img .gdbinit

    @echo "\*\*\* Now run 'gdb'." 1>&2

    $(QEMU) -nographic $(QEMUOPTS) -S $(QEMUGDB)

# CUT HERE

# prepare dist for students

# after running make dist, probably want to

# rename it to rev0 or rev1 or so on and then

# check in that version.

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 test.c\

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

    .gdbinit.tmpl gdbutil\

dist:

    rm -rf dist

    mkdir dist

    for i in $(FILES); \

    do \

        grep -v PAGEBREAK $$i >dist/$$i; \

    done

    sed '/CUT HERE/,$$d' Makefile >dist/Makefile

    echo >dist/runoff.spec

    cp $(EXTRA) dist

dist-test:

    rm -rf dist

    make dist

    rm -rf dist-test

    mkdir dist-test

    cp dist/\* dist-test

    cd dist-test; $(MAKE) print

    cd dist-test; $(MAKE) bochs || true

    cd dist-test; $(MAKE) qemu

# update this rule (change rev#) when it is time to

# make a new revision.

tar:

    rm -rf /tmp/xv6

    mkdir -p /tmp/xv6

    cp dist/\* dist/.gdbinit.tmpl /tmp/xv6

    (cd /tmp; tar cf - xv6) | gzip >xv6-rev10.tar.gz  # the next one will be 10 (9/17)

.PHONY: dist-test dist

**file.c**

//

// File descriptors

//

#include "types.h"

#include "defs.h"

#include "param.h"

#include "fs.h"

#include "spinlock.h"

#include "sleeplock.h"

#include "file.h"

extern int new;

struct devsw devsw[NDEV];

struct {

  struct spinlock lock;

  struct file file[NFILE];

} ftable;

void

fileinit(void)

{

  initlock(&ftable.lock, "ftable");

}

// Allocate a file structure.

struct file\*

filealloc(void)

{

  struct file \*f;

  acquire(&ftable.lock);

  for(f = ftable.file; f < ftable.file + NFILE; f++){

    if(f->ref == 0){

      f->ref = 1;

      release(&ftable.lock);

      return f;

    }

  }

  release(&ftable.lock);

  return 0;

}

// Increment ref count for file f.

struct file\*

filedup(struct file \*f)

{

  acquire(&ftable.lock);

  if(f->ref < 1)

    panic("filedup");

  f->ref++;

  release(&ftable.lock);

  return f;

}

// Close file f.  (Decrement ref count, close when reaches 0.)

void

fileclose(struct file \*f)

{

  struct file ff;

  acquire(&ftable.lock);

  if(f->ref < 1)

    panic("fileclose");

  if(--f->ref > 0){

    release(&ftable.lock);

    return;

  }

  ff = \*f;

  f->ref = 0;

  f->type = FD\_NONE;

  release(&ftable.lock);

  if(ff.type == FD\_PIPE)

    pipeclose(ff.pipe, ff.writable);

  else if(ff.type == FD\_INODE){

    begin\_op();

    iput(ff.ip);

    end\_op();

  }

}

// Get metadata about file f.

int

filestat(struct file \*f, struct stat \*st)

{

  if(f->type == FD\_INODE){

    ilock(f->ip);

    stati(f->ip, st);

    iunlock(f->ip);

    return 0;

  }

  return -1;

}

// Read from file f.

int

fileread(struct file \*f, char \*addr, int n)

{

  int r;

  if(f->readable == 0)

    return -1;

  if(f->type == FD\_PIPE)

    return piperead(f->pipe, addr, n);

  if(f->type == FD\_INODE){

    ilock(f->ip);

    if((r = readi(f->ip, addr, f->off, n)) > 0)

      f->off += r;

    iunlock(f->ip);

    return r;

  }

  panic("fileread");

}

//PAGEBREAK!

// Write to file f.

//addr 은 문자열 n이 사이즈 20202925

int

filewrite(struct file \*f, char \*addr, int n)

{

  int r;

  if(f->writable == 0)

    return -1;

  if(f->type == FD\_PIPE)

    return pipewrite(f->pipe, addr, n);

  if(f->type == FD\_INODE){

    // write a few blocks at a time to avoid exceeding

    // the maximum log transaction size, including

    // i-node, indirect block, allocation blocks,

    // and 2 blocks of slop for non-aligned writes.

    // this really belongs lower down, since writei()

    // might be writing a device like the console.

    int max = ((MAXOPBLOCKS-1-1-2) / 2) \* 512;

    int i = 0;

  if(f->off==0)

    new=0;

    while(i < n){

      int n1 = n - i;

      if(n1 > max)

        n1 = max;

   // cprintf("%d",n1);

      begin\_op();

      ilock(f->ip);

      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)

        f->off += r;

      iunlock(f->ip);

      end\_op();

      if(r < 0){

        break;

    }

      if(r != n1)

        panic("short filewrite");

      i += r;

    }

    return i == n ? n : -1;

  }

  panic("filewrite");

}