```c
#include "cmd.h"
bool do_blocks(cmd *c) {

  if (c->argc != 2) {
    printf("Usage: blocks <filename>\n");
    return false;
  }

  char buf1[BLKSIZE_1024], buf2[BLKSIZE_1024], buf3[BLKSIZE_1024];
  int *fs_p1, *fs_p2, *fs_p3;
  path in_path;
  minode *mip;

  parse_path(c->argv[1], &in_path);
  if (!(mip = search_path(in_path)))
    return false;

  printf("\ndirect blocks:\n");
  for (int i = 0; i < 12 && mip->inode.i_block[i]; i++)
    printf(" %-4u ", mip->inode.i_block[i]);

  get_block(mip->dev, mip->inode.i_block[12], buf1);
  if (mip->inode.i_block[12])
    printf("\nindirect blocks:\n[%u] :\n", mip->inode.i_block[12]);
  fs_p1 = (int *)buf1;
  while (*fs_p1 && ((char *)fs_p1 < buf1 + BLKSIZE_1024))
    printf(" %-4u ", *fs_p1++);

  get_block(mip->dev, mip->inode.i_block[13], buf1);
  if (mip->inode.i_block[13])
    printf("\ndouble indirect blocks:\n[%u] :\n", mip->inode.i_block[13]);
  fs_p1 = (int *)buf1;
  while (*fs_p1 && ((char *)fs_p1 < buf1 + BLKSIZE_1024)) {
    get_block(mip->dev, *fs_p1, buf2);
    printf("[[%u]] :\n", *fs_p1);
    fs_p2 = (int *)buf2;
    while (*fs_p2 && ((char *)fs_p2 < buf2 + BLKSIZE_1024))
      printf(" %-4u ", *fs_p2++);
    printf("\n");
    fs_p1++;
  }

  get_block(mip->dev, mip->inode.i_block[14], buf1);
  if (mip->inode.i_block[14])
    printf("triple indirect blocks:\n[%u] :\n", mip->inode.i_block[14]);
  fs_p1 = (int *)buf1;
  while (*fs_p1 && ((char *)fs_p1 < buf1 + BLKSIZE_1024)) {
    get_block(mip->dev, *fs_p1, buf2);
    printf("[[%u]] :\n", *fs_p1);
    fs_p2 = (int *)buf2;
    while (*fs_p2 && ((char *)fs_p2 < buf2 + BLKSIZE_1024)) {
      get_block(mip->dev, *fs_p2, buf3);
      printf("[[[%u]]] :\n", *fs_p2);
      fs_p3 = (int *)buf3;
      while (*fs_p3 && ((char *)fs_p3 < buf3 + BLKSIZE_1024))
        printf(" %-4u ", *fs_p3++);
      printf("\n");
      fs_p2++;
    }
    printf("\n");
    fs_p1++;
  }
  printf("\n");

  put_minode(mip);
  return true;
}
```

```
#include "cmd.h"

bool do_cat(cmd *c) {

  if (c->argc != 2) {
    printf("Usage: cat <filename>\n");
    return false;
  }

  char mybuf[1024], dummy = 0; // a null char at end of mybuf[ ]
  int n, fd;

  if (fd = open_file(c->argv[1], 0) == -1) {
    printf("can't open file for read\n");
    return false;
  }
  while ((n = read_file(fd, mybuf, 1024))) {
    mybuf[n] = 0;        // as a null terminated string
    printf("%s", mybuf); // <=== THIS works but not good
  }
  printf("\n");
  if (close_file(fd) < 0) {
    printf("fail to close file\n");
    return false;
  }
  return true;
}
```

```c
#include "cmd.h"

bool do_cd(cmd *c) {
  if (c->argc < 2)
    return _cd("/");
  else
    return _cd(c->argv[1]);
}

int _cd(char *dest) {
  path in_path;
  minode *mip;
  parse_path(dest, &in_path);
  if (!(mip = search_path(in_path))) {
    printf("path not found\n");
    return 0;
  }
  // if we got back a symlink
  if (S_ISLNK(mip->inode.i_mode)) {
    path sym_path;
    parse_path((char *)mip->inode.i_block, &sym_path);
    if (!sym_path.is_absolute) {
      // replace link name with link contents and search again
      memcpy(&(in_path.argv[--(in_path.argc)]), sym_path.argv,
             sizeof(char *) * sym_path.argc);
      in_path.argc += sym_path.argc;
      mip = search_path(in_path);
    } else // search absolute path
      mip = search_path(sym_path);
  }
  if (!S_ISDIR(mip->inode.i_mode)) {
    printf("cannot cd to non-dir\n");
    return 0;
  }

  if (mip == NULL) {
    printf("path not found\n");
  }

  put_minode(running->cwd);
  running->cwd = mip;
  DEBUG_PRINT("cwd is now %d\n", mip->ino);
  return mip->ino;
}
```

```c
#include "cmd.h"

bool do_chmod(cmd *c) {
  path in_path;
  if (c->argc != 3) {
    printf("usage: chmod <mode> <filename>\n");
    return false;
  }
  if (!parse_path(c->argv[2], &in_path)) {
    printf("bad path");
    return false;
  }
  minode *mip = search_path(in_path);
  // long int strtol (const char* str, char** endptr, int base);
  // if given base == 0 then base is determined by +, -,0, 0X/0x prefix
  unsigned int mode = strtol(c->argv[1], NULL, 0);
  mip->inode.i_mode |= (__u16)mode;
  mip->dirty = true;
  put_minode(mip);
  return true;
}
```

```c
#include "cmd.h"

bool do_close(cmd *c) {
  if (c->argc != 2) {
    printf("Usage: close <filename>\n");
    return false;
  }

  int fd;
  struct path p;
  parse_path(c->argv[1], &p);
  minode *mip = NULL;
  if (!(mip = search_path(p)) || !S_ISREG(mip->inode.i_mode)) {
    put_minode(mip);
    return 0;
  }

  for (fd = 0; fd < NUM_OFT_PER && !(running->oft_arr[fd] == NULL); fd++) {
    if (running->oft_arr[fd]->minode->ino == mip->ino) {
      put_minode(mip);
      return close_file(fd);
    }
  }

  return true;
}
```

```c
#include "cmd.h"

bool do_cp(cmd *c) {
  if (c->argc != 3) {
    printf("Usage: cp <src filename> <dest filename>\n");
    return false;
  }
  return _cp(c->argv[1], c->argv[2]);
}

int _cp(char *src, char *dest) {
  minode *src_mip, *dest_mip;
  int src_fd, dest_fd, n, copied = 0;
  char buf[BLKSIZE_1024];
  // open src for READ;
  if ((src_fd = open_file(src, 0)) < 0) {
    printf("failed to open src for read\n");
    return 0;
  }
  // creat dst if not exist
  if (_creat(dest))
    DEBUG_PRINT("creat %s", dest);
  // open dst for WR;
  if ((dest_fd = open_file(dest, 2)) < 0) {
    printf("failed to open dest for read\n");
    return 0;
  }
  // copy data
  while (n = read_file(src_fd, buf, BLKSIZE_1024)) {
    int written = write_file(dest_fd, buf, n);
    DEBUG_PRINT("wrote %d\n", written);
    copied += n;
  }

  // close src;
  if ((close_file(src_fd)) < 0) {
    printf("failed to close src\n");
    return 0;
  }
  // close dest;
  if ((close_file(dest_fd)) < 0) {
    printf("failed to close dest\n");
    return 0;
  }
  return copied;
}
```

```c
#include "cmd.h"

bool do_creat(cmd *c) {

  if (c->argc != 2) {
    printf("Usage: creat <filename>\n");
    return false;
  }
  if (!_creat(c->argv[1])) {
    printf("fail to creat %s\n", c->argv[1]);
    return false;
  }
  return true;
}

int _creat(char *dest) {
  path in_path;
  minode *exists;
  parse_path(dest, &in_path);
  char *bname = in_path.argv[in_path.argc - 1];
  if ((exists = search_path(in_path))) {
    printf("%s already exists\n", dest);
    put_minode(exists);
    return 0;
  }
  in_path.argc--;
  minode *parent = search_path(in_path);
  if (!S_ISDIR(parent->inode.i_mode)) {
    printf("Can't add file to non-directory\n");
    return 0;
  }
  int ino = alloc_inode(parent->dev);

  minode *child = get_minode(parent->dev, ino);
  child->inode.i_mode = 0x81A4;      // OR 0100644: REG type and permissions
  child->inode.i_uid = running->uid; // Owner uid
  child->inode.i_gid = running->gid; // Group Id
  child->inode.i_size = 0;           // Size in bytes nothing in file
  child->inode.i_links_count = 1;    // Links count=1 because REG
  child->inode.i_atime = child->inode.i_ctime = child->inode.i_mtime = time(0L);
  child->inode.i_blocks = 0; // LINUX: Blocks count in 512-byte chunks
  for (int i = 0; i < 15; i++)
    child->inode.i_block[i] = 0;
  child->dirty = true;

  // add child to parent
  dir_entry pcd, *parent_child_dir = &pcd;
  parent_child_dir->inode = child->ino;
  strcpy(parent_child_dir->name, bname);
  parent_child_dir->name_len = strlen(parent_child_dir->name);
  add_dir_entry(parent, parent_child_dir);

  DEBUG_PRINT("creat file with ino %d\n", child->ino);
  // write back to disk / put
  put_minode(parent);
  put_minode(child);
  return ino;
}
```

```c
#pragma once

#include "../fs/fs.h"
#include <stdbool.h>
#include <stdio.h>
#include <time.h>

typedef struct cmd {
  int argc;       // count of strings
  char *argv[64]; // array of strings
} cmd;

// command handlers
bool do_blocks(cmd *);
bool do_cat(cmd *);
bool do_cd(cmd *);
bool do_chmod(cmd *);
bool do_close(cmd *);
bool do_cp(cmd *);
bool do_creat(cmd *);
bool do_link(cmd *);
bool do_ls(cmd *);
bool do_lseek(cmd *);
bool do_mkdir(cmd *);
bool do_mount(cmd *);
bool do_mv(cmd *);
bool do_open(cmd *);
bool do_pwd(cmd *);
bool do_read(cmd *);
bool do_rmdir(cmd *);
bool do_stat(cmd *);
bool do_su(cmd *c);
bool do_symlink(cmd *);
bool do_touch(cmd *);
bool do_umount(cmd *);
bool do_unlink(cmd *);
bool do_write(cmd *);

// command implementations
int _cd(char *);
int _cp(char *, char *);
int _creat(char *);
int _link(char *, char *);
int _ls_file(minode *, char *);
int _mkdir(char *);
int _mount(char *, char *);
int _pwd(minode *);
int _rmdir(char *);
int _symlink(char *, char *);
int _umount(char *);
int _unlink(char *);

// utility
bool do_cmd(cmd *c);
int parse_cmd(char *, cmd *);
```

```c
#include "cmd.h"

bool do_link(cmd *c) {
  if (c->argc != 3) {
    printf("Usage: link <src filename> <dest filename>\n");
    return false;
  }
  return _link(c->argv[1], c->argv[2]);
}

int _link(char *src, char *dest) {
  path src_path, dest_path;
  parse_path(src, &src_path);
  parse_path(dest, &dest_path);

  char *bname = dest_path.argv[dest_path.argc - 1];

  dest_path.argc--;
  minode *dest_parent = search_path(dest_path);
  minode *mip = search_path(src_path);
  if (!mip || !dest_parent) {
    printf("bad path\n");
    return 0;
  }

  if (!(S_ISREG(mip->inode.i_mode) || S_ISLNK(mip->inode.i_mode))) {
    printf("cannot link this type of file\n");
    return false;
  }

  // add child to parent
  dir_entry de, *dep = &de;
  dep->inode = mip->ino;
  strcpy(dep->name, bname);
  dep->name_len = strlen(dep->name);
  add_dir_entry(dest_parent, dep);
  mip->inode.i_links_count++;

  DEBUG_PRINT("ino %d link count %d\n", mip->ino, mip->inode.i_links_count);
  // write back to disk / put
  mip->dirty = true;
  put_minode(mip);
  dest_parent->dirty = true;
  put_minode(dest_parent);
  return mip->inode.i_links_count;
}
```

```c
#include "cmd.h"

bool do_ls(cmd *c) {
  dir_entry dep[4096];
  minode *cur_dir = NULL;
  int entryc;

  if (c->argc < 2) {
    cur_dir = running->cwd;
    cur_dir->ref_count++;
  } else {
    path in_path;
    parse_path(c->argv[1], &in_path);
    cur_dir = search_path(in_path);
    if (!cur_dir) {
      printf("invalid path\n");
      return false;
    }
  }
  entryc = list_dir(cur_dir, dep);
  for (int i = 0; i < entryc; i++) {
    minode *file = get_minode(cur_dir->dev, dep[i].inode);
    // printf("%s\n", dep[i].name);
    char filename[256] = {0};
    strncpy(filename, dep[i].name, dep[i].name_len);
    _ls_file(file, filename);

    put_minode(file);
  }
  put_minode(cur_dir);
  return true;
}

int _ls_file(minode *file, char *fname) {
  char *t1 = "xwrxwrxwr-------";
  char *t2 = "----------------";
  char ftime[256], buf[256] = {0};
  int r, i;
  if (S_ISREG(file->inode.i_mode))
    printf("-");
  else if (S_ISDIR(file->inode.i_mode))
    printf("d");
  else if (S_ISLNK(file->inode.i_mode))
    printf("l");
  for (i = 8; i >= 0; i--) {
    if ((file->inode.i_mode & (1 << i))) // print r|w|x
      printf("%c", t1[i]);
    else
      printf("%c", t2[i]);
  }
  printf("%4d %4d %4d %8d ", (int)file->inode.i_links_count, file->inode.i_gid,
         file->inode.i_uid, (int)file->inode.i_size);
  printf("%s %s", strtok(ctime((long *)&file->inode.i_ctime), "\n"), fname);
  if ((file->inode.i_mode & 0xF000) == 0xA000) {
    // use readlink() to read linkname
    char linkname[256] = {0};
    readlink(fname, linkname, 256);
    printf(" -> %s", linkname);
  }
  printf("\n");
  return 0;
}
```

```c
#include "cmd.h"

bool do_lseek(cmd *c) {

  if (c->argc != 4) {
    printf("Usage: lseek <fd> <offset> <position 0|1|2 = "
           "SEET_SET|SEET_CUR|SEEK_END>\n");
    return false;
  }
  return lseek_file(atoi(c->argv[1]), atoi(c->argv[2]), atoi(c->argv[3])) == -1
             ? false
             : true;
}
```

```c
#include "cmd.h"

bool do_mkdir(cmd *c) {

  if (c->argc != 2) {
    printf("Usage: mkdir <dest>\n");
    return false;
  }

  return _mkdir(c->argv[1]);
}

int _mkdir(char *dest) {
  minode *exists;
  path in_path;
  parse_path(dest, &in_path);
  char *bname = in_path.argv[in_path.argc - 1];
  if ((exists = search_path(in_path))) {
    printf("%s already exists\n", dest);
    put_minode(exists);
    return 0;
  }
  in_path.argc--;
  minode *parent = search_path(in_path);
  if (!S_ISDIR(parent->inode.i_mode)) {
    printf("Can't add file to non-directory\n");
    return 0;
  }
  int ino = alloc_inode(parent->dev);

  minode *child = get_minode(parent->dev, ino);
  child->inode.i_mode = 0x41ED;       // OR 040755: DIR type and permissions
  child->inode.i_uid = running->uid;  // Owner uid
  child->inode.i_gid = running->gid;  // Group Id
  child->inode.i_size = BLKSIZE_1024; // Size in bytes
  child->inode.i_links_count = 0;     // incremented in add_dir_entry
  child->inode.i_atime = child->inode.i_ctime = child->inode.i_mtime = time(0L);
  child->inode.i_blocks = 2; // LINUX: Blocks count in 512-byte chunks
  for (int i = 0; i < 15; i++)
    child->inode.i_block[i] = 0;

  child->dirty = true;

  // make .
  dir_entry cd, *child_dir = &cd;
  child_dir->inode = child->ino;
  strcpy(child_dir->name, ".");
  child_dir->name_len = strlen(child_dir->name);
  add_dir_entry(child, child_dir);
  // make ..
  dir_entry pd, *parent_dir = &pd;
  parent_dir->inode = parent->ino;
  strcpy(parent_dir->name, "..");
  parent_dir->name_len = strlen(parent_dir->name);
  add_dir_entry(child, parent_dir);
  // add child to parent
  dir_entry pcd, *parent_child_dir = &pcd;
  parent_child_dir->inode = child->ino;
  strcpy(parent_child_dir->name, bname);
  parent_child_dir->name_len = strlen(parent_child_dir->name);
  add_dir_entry(parent, parent_child_dir);
  // write back to disk / put
  put_minode(parent);
  put_minode(child);
  return ino;
}
```

```c
#include "cmd.h"

bool do_mount(cmd *c) {
  if (c->argc == 1) {
    for (int i = 0; i < NUM_MOUNT_ENTRIES; i++) {
      mount_entry *me = &mount_entry_arr[i];
      if (me->fd)
        printf("me: %s at %s with fd %d\n", me->dev_path, me->mnt_path, me->fd);
    }
    return true;
  } else if (c->argc == 3) {
    return _mount(c->argv[1], c->argv[2]);
  } else {
    printf("Usage: mount <device> <dir>\n");
    return false;
  }
}

int _mount(char *dev, char *dir) {

  // check if already mounted
  for (int i = 0; i < NUM_MOUNT_ENTRIES; i++) {
    if (mount_entry_arr[i].fd && (!strcmp(dev, mount_entry_arr[i].dev_path))) {
      printf("cannot mount, you already mount! why?\n");
      return 0;
    }
  }

  // make a directory at dir
  _mkdir(dir);

  // search for the directory we just made
  minode *mnt_dir;
  path dir_path;
  parse_path(dir, &dir_path);
  if (!(mnt_dir = search_path(dir_path)) || !S_ISDIR(mnt_dir->inode.i_mode)) {
    printf("bad path given for mount point\n");
    put_minode(mnt_dir);
    return false;
  }

  // make mount entry for device
  mount_entry *me = make_me(dev, dir);

  // set device mount point to minode
  me->mnt_pnt = mnt_dir;

  // shadow new dir's ino with mnt point
  mnt_dir->mnt = me;
  mnt_dir->dirty = false;

  return mnt_dir->ino;
}
```

```c
#include "cmd.h"

bool do_mv(cmd *c) {
  if (c->argc != 3) {
    printf("Usage: mv <src> <dest>\n");
    return false;
  }
  path src_path, dest_path;
  parse_path(c->argv[1], &src_path);
  parse_path(c->argv[2], &dest_path);

  char *bname = dest_path.argv[dest_path.argc - 1];

  dest_path.argc--;
  minode *dest_parent = search_path(dest_path);
  minode *mip = search_path(src_path);
  if (!mip || !dest_parent) {
    printf("bad path\n");
    return 0;
  }
  if (dest_parent->dev != mip->dev)
    return _cp(c->argv[1], c->argv[2]);
  else {
    return (_link(c->argv[1], c->argv[2]) && _unlink(c->argv[1]));
  }
}
```

```c
#include "cmd.h"

bool do_open(cmd *c) {
  if (c->argc < 3) {
    printf("Usage: open <path/to/file> <mode: (0|1|2|3 or R|W|RW|APPEND)>\n");
    return false;
  }
  int mode;
  if (!strcmp(c->argv[2], "R") || !strcmp(c->argv[2], "0"))
    mode = 0;
  else if (!strcmp(c->argv[2], "W") || !strcmp(c->argv[2], "1")) {
    mode = 1;
  } else if (!strcmp(c->argv[2], "RW") || !strcmp(c->argv[2], "2"))
    mode = 2;
  else if (!strcmp(c->argv[2], "APPEND") || !strcmp(c->argv[2], "3")) {
    mode = 3;
  } else {
    printf("Usage: open <path/to/file> <mode: (0|1|2|3 or R|W|RW|APPEND)>\n");
    return false;
  }

  int fd = open_file(c->argv[1], mode);
  if (fd == -1)
    printf("Usage: open <path/to/file> <mode: (0|1|2|3 or R|W|RW|APPEND)>\n");
  return (bool)fd;
}
```

```c
#include "cmd.h"

bool do_pwd(cmd *c) {
  if (running->cwd == global_root_inode)
    printf("/\n");
  else {
    _pwd(running->cwd);
    printf("\n");
  }
  return true;
}

int _pwd(minode *mip) {
  if (mip == global_root_inode)
    ;
  else {
    char buf1[BLKSIZE_1024], *buf1p;
    char buf2[BLKSIZE_1024], *buf2p;
    char name[256] = {0};
    dir_entry *this_dir, *parent_dir, *dirp;

    get_block(mip->dev, mip->inode.i_block[0], buf1);
    this_dir = (dir_entry *)buf1;
    parent_dir = (dir_entry *)(buf1 + this_dir->rec_len);
    minode *pip = get_minode(mip->dev, parent_dir->inode);

    if (mip->ino == pip->ino) {
      minode *newpip = mip->dev->mnt_pnt;
      put_minode(pip);
      pip = newpip;
    } else {

      for (int i = 0; i < 12 && !(*name); i++) { // search direct blocks only
        if (pip->inode.i_block[i] == 0)
          break;
        get_block(pip->dev, pip->inode.i_block[i], buf2);
        dirp = (dir_entry *)buf2;
        buf2p = buf2;
        while (buf2p < buf2 + BLKSIZE_1024) {
          dirp = (dir_entry *)buf2p;
          buf2p += dirp->rec_len;
          if (dirp->inode == this_dir->inode) {
            strncpy(name, dirp->name, dirp->name_len);
            name[dirp->name_len] = '\0';
            break;
          }
        }
      }
      put_minode(pip);
    }
    _pwd(pip); // recursive call
    printf("/%s", name);
  }
}
```

```c
#include "cmd.h"

bool do_read(cmd *c) {
  char buf[4096] = {0};
  if (c->argc < 3) {
    printf("Usage: read <fd> <bytes <= 4096>");
  }
  read_file(atoi(c->argv[1]), buf, atoi(c->argv[2]));
  printf("%s\n", buf);
  return true;
}
```

```c
#include "cmd.h"

bool do_rmdir(cmd *c) {

  if (c->argc != 2) {
    printf("Usage: rmdir <path>\n");
    return false;
  }
  return _rmdir(c->argv[1]);
}

int _rmdir(char *dest) {
  path in_path;
  parse_path(dest, &in_path);
  minode *mip = search_path(in_path);
  if (!S_ISDIR(mip->inode.i_mode)) {
    printf("Can't rm non-directory\n");
    put_minode(mip);
    return 0;
  }
  char *bname = in_path.argv[in_path.argc - 1];
  in_path.argc--;
  minode *parent = search_path(in_path);

  // Just like in real filesystems
  if (running->uid != 0 && mip->inode.i_uid != running->uid) {
    printf("you do not have permission\n");
    put_minode(mip);
    return 0;
  }
  if (mip->ref_count > 1) {
    printf("DIR is in use\n");
    put_minode(mip);
    return 0;
  }
  if (!(mip->inode.i_links_count > 2))
    if (count_dir(mip) > 2) {
      printf("DIR is not empty\n");
      put_minode(mip);
      return 0;
    }
  int ino = mip->ino;
  // Deallocate its block and inode
  for (int i = 0; i < 12; i++) {
    if (mip->inode.i_block[i] == 0)
      continue;
    free_block(mip->dev, mip->inode.i_block[i]);
  }
  free_minode(mip);
  put_minode(mip);
  rm_dir_entry(parent, bname);
  put_minode(parent);
  return ino;
}
```

```c
#include "cmd.h"

bool do_stat(cmd *c) {
  path in_path;
  minode *mip;
  if (c->argc < 2) {
    printf("stat requires: stat filename\n");
    return false;
  }
  if (!parse_path(c->argv[1], &in_path) || !(mip = search_path(in_path))) {
    printf("bad path");
    return false;
  }
  inode *i = &mip->inode;
  printf("File: %s\n"
         "Size: %u\t Blocks: %u\t Mode: %o\n"
         "Device: %s\t Ino: %u\t Links: %u\t \n"
         "Uid: %u\t Gid: %u\t \n"
         "Access: %s"
         "Modify: %s"
         "Change: %s",
         c->argv[1], i->i_size, i->i_blocks, i->i_mode, mip->dev->dev_path,
         mip->ino, i->i_links_count, i->i_uid, i->i_gid,
         ctime((long *)&i->i_atime), ctime((long *)&i->i_mtime),
         ctime((long *)&i->i_ctime));

  return true;
}
```

```
#include "cmd.h"

bool do_su(cmd *c) {
  // check enough args
  if (c->argc != 2) {
    printf("Usage: su <uid>\n");
    return false;
  }
  running->uid = atoi(c->argv[1]);
  return true;
}
```

```c
#include "cmd.h"

bool do_symlink(cmd *c) {
  // check enough args
  if (c->argc != 3) {
    printf("Usage: symlink <src> <dest>\n");
    return false;
  }
  return _symlink(c->argv[1], c->argv[2]);
}

int _symlink(char *src, char *dest) {

  path src_path, dest_path;
  parse_path(src, &src_path);
  parse_path(dest, &dest_path);

  char *bname = dest_path.argv[dest_path.argc - 1];

  dest_path.argc--;
  minode *dest_parent = search_path(dest_path);
  minode *mip = search_path(src_path);
  if (!mip || !dest_parent) {
    printf("bad path\n");
    return 0;
  }

  if (S_ISLNK(mip->inode.i_mode)) {
    printf("cannot link a link because then you link to the link to the "
           "link...\n");
    return 0;
  }
  int ino = alloc_inode(dest_parent->dev);
  minode *child = get_minode(dest_parent->dev, ino);

  child->inode.i_mode = 0120000;      // LNK type and permissions
  child->inode.i_uid = running->uid; // Owner uid
  child->inode.i_gid = running->gid; // Group Id
  child->inode.i_size = 0;           // Size in bytes nothing
  child->inode.i_links_count = 1;    // Links count=1
  child->inode.i_atime = child->inode.i_ctime = child->inode.i_mtime = time(0L);
  child->inode.i_blocks = 0; // LINUX: Blocks count in 512-byte chunks
  // copy path into i_block
  strcpy((char *)child->inode.i_block, src);
  child->dirty = true;

  // add child to parent
  dir_entry pcd, *parent_child_dir = &pcd;
  parent_child_dir->inode = child->ino;
  strcpy(parent_child_dir->name, bname);
  parent_child_dir->name_len = strlen(parent_child_dir->name);
  add_dir_entry(dest_parent, parent_child_dir);

  DEBUG_PRINT("symlink file with ino %d\n", child->ino);
  // write back to disk / put
  put_minode(dest_parent);
  put_minode(child);
  return ino;
}
```

```c
#include "cmd.h"

bool do_touch(cmd *c) {
  path in_path;
  if (c->argc != 2) {
    printf("Usage: touch <filename>\n");
    return false;
  }
  parse_path(c->argv[1], &in_path);
  minode *mip = search_path(in_path);
  if (!mip)
    return _creat(c->argv[1]);
  else
    return !!(mip->inode.i_atime = time(0L));
}
```

```
#include "cmd.h"

bool do_umount(cmd *c) {
  if (c->argc != 2) {
    printf("Usage: umount <dir>\n");
    return false;
  }
  return _umount(c->argv[1]);
}

int _umount(char *dir) {
  // find mount minode
  path mnt_path;
  parse_path(dir, &mnt_path);
  minode *mip;
  if (!(mip = search_path(mnt_path)) || !(mip->ino == 2)) {
    printf("not a mount point\n");
    return 0;
  }

  minode *newmip = mip->dev->mnt_pnt;
  put_minode(mip);
  mip = newmip;

  // get mount entry
  mount_entry *me = mip->mnt;

  // check if dev in use
  for (int i = 0; i < NUM_MINODES; i++) {
    if ((minode_arr[i].ref_count) && (minode_arr[i].dev == me)) {
      printf("cannot umount, device in use");
      return 0;
    }
  }

  // free me
  for (int i = 0; i < NUM_MOUNT_ENTRIES + 1; i++) {
    if (i == NUM_MOUNT_ENTRIES) {
      printf("could not locate mount entry\n");
      return 0;
    }
    if (&mount_entry_arr[i] == me) {
      DEBUG_PRINT("closing fd %d\n", mount_entry_arr[i].fd);
      close(mount_entry_arr[i].fd);
      mount_entry_arr[i].fd = 0;
      break;
    }
  }

  // set ref count zero and not dirty
  mip->ref_count = 1;
  mip->dirty = false;
  // put
  put_minode(mip);
  return 0;
}
```

```c
#include "cmd.h"

bool do_unlink(cmd *c) {
  if (c->argc != 2) {
    printf("Usage: unlink <filename>\n");
    return false;
  }
  return _unlink(c->argv[1]);
}

int _unlink(char *dest) {
  path in_path;
  minode *mip, *parent;
  if (!parse_path(dest, &in_path) || !(mip = search_path(in_path))) {
    printf("bad path");
    return false;
  }

  if (running->uid != 0 && mip->inode.i_uid != running->uid) {
    printf("you do not have permission\n");
    put_minode(mip);
    return 0;
  }

  if (S_ISDIR(mip->inode.i_mode)) {
    printf("Can't unlink directory\n");
    put_minode(mip);
    return false;
  }
  char *bname = in_path.argv[in_path.argc - 1];
  in_path.argc--;
  parent = search_path(in_path);
  mip->inode.i_links_count--;
  if (!mip->inode.i_links_count) {
    DEBUG_PRINT("freeing ino %d\n", mip->ino);
    free_i_block(mip);
    free_inode(mip->dev, mip->ino);
  } else {
    DEBUG_PRINT("ino %d link count now %d\n", mip->ino,
                mip->inode.i_links_count);
  }
  rm_dir_entry(parent, bname);
  mip->dirty = true;
  put_minode(mip);
  parent->dirty = true;
  put_minode(parent);
}
```

```c
#include "../debug/debug.h"
#include "cmd.h"
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/statvfs.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

bool do_cmd(cmd *c) {
  if (!c)
    DEBUG_PRINT("cmd was null\n");
  if (!strcmp(c->argv[0], "blocks")) {
    do_blocks(c);
  } else if (!strcmp(c->argv[0], "cd")) {
    do_cd(c);
  } else if (!strcmp(c->argv[0], "cat")) {
    do_cat(c);
  } else if (!strcmp(c->argv[0], "chmod")) {
    do_chmod(c);
  } else if (!strcmp(c->argv[0], "close")) {
    do_close(c);
  } else if (!strcmp(c->argv[0], "cp")) {
    do_cp(c);
  } else if (!strcmp(c->argv[0], "creat")) {
    do_creat(c);
  } else if (!strcmp(c->argv[0], "link")) {
    do_link(c);
  } else if (!strcmp(c->argv[0], "ls")) {
    do_ls(c);
  } else if (!strcmp(c->argv[0], "lseek")) {
    do_lseek(c);
  } else if (!strcmp(c->argv[0], "mkdir")) {
    do_mkdir(c);
  } else if (!strcmp(c->argv[0], "mount")) {
    do_mount(c);
  } else if (!strcmp(c->argv[0], "mv")) {
    do_mv(c);
  } else if (!strcmp(c->argv[0], "open")) {
    do_open(c);
  } else if (!strcmp(c->argv[0], "pwd")) {
    do_pwd(c);
  } else if (!strcmp(c->argv[0], "read")) {
    do_read(c);
  } else if (!strcmp(c->argv[0], "rmdir")) {
    do_rmdir(c);
  } else if (!strcmp(c->argv[0], "stat")) {
    do_stat(c);
  } else if (!strcmp(c->argv[0], "su")) {
    do_su(c);
  } else if (!strcmp(c->argv[0], "symlink")) {
    do_symlink(c);
  } else if (!strcmp(c->argv[0], "touch")) {
    do_touch(c);
  } else if (!strcmp(c->argv[0], "umount")) {
    do_umount(c);
  } else if (!strcmp(c->argv[0], "unlink")) {
    do_unlink(c);
  } else if (!strcmp(c->argv[0], "write")) {
    do_write(c);
  } else if (!strcmp(c->argv[0], "quit")) {
    for (int i = 0; i < NUM_MINODES; i++) {
      if (minode_arr[i].ref_count) {
        minode_arr[i].ref_count = 1;
        put_minode(&minode_arr[i]);
      }
    }
    for (int i = 0; i < NUM_MOUNT_ENTRIES; i++) {
      if (mount_entry_arr[i].fd)
        _umount(mount_entry_arr[i].mnt_path);
    }
    exit(0);
  } else {
    printf("command not recognized: %s\n", c->argv[0]);
  }
  return 0;
}

int parse_cmd(char *line, cmd *c) {
  // split by whitespace into cmd struct
  int i = 0;
  char *s = strtok(line, " ");
  for (; s; i++) {
    c->argv[i] = s;
    s = strtok(NULL, " ");
  }
  c->argc = i;
  // NULL terminate argv
  c->argv[i] = NULL;
  return i;
}
```

```c
#include "cmd.h"

bool do_write(cmd *c) {
  char buf[4096];
  int wrote;
  if (c->argc < 3) {
    printf("Usage: write <fd> <message>\n");
  }
  strcpy(buf, c->argv[2]);
  wrote = write_file(atoi(c->argv[1]), buf, strlen(buf));
  printf("wrote %d bytes\n", wrote);
  return true;
}
```

```
#pragma once

//// DEBUG ////

// Uncommment for debug mode
#define DEBUG_MODE 1

// debug messages
#if defined DEBUG_MODE
#define DEBUG_PRINT(fmt, args...)                                              \
  fprintf(stderr, "DEBUG: %s:%d:%s(): " fmt, __FILE__, __LINE__, __func__,     \
          ##args)
#else
// Don't do anything in release builds
#define DEBUG_PRINT(fmt, args...)
#endif
```

```c
#ifndef _CPTS360_FS_H
#define _CPTS360_FS_H

#include "../debug/debug.h"
#include <ext2fs/ext2_fs.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

//// TYPEDEF ////

// define shorter TYPES for convenience
typedef struct ext2_group_desc group_desc;
typedef struct ext2_super_block super_block;
typedef struct ext2_inode inode;
typedef struct ext2_dir_entry_2 dir_entry;

//// CONST ////

// Block size
#define BLKSIZE_1024 1024

// Inode numbers of EXT2 as defined in ext2fs.h
// EXT2_BAD_INO 1         Bad blocks inode
// EXT2_ROOT_INO 2        Root inode
// EXT4_USR_QUOTA_INO 3   User quota inode
// EXT4_GRP_QUOTA_INO 4   Group quota inode
// EXT2_BOOT_LOADER_INO 5 Boot loader inode
// EXT2_UNDEL_DIR_INO 6   Undelete directory inode
// EXT2_RESIZE_INO 7      Reserved group descriptors inode
// EXT2_JOURNAL_INO 8     Journal inode
// EXT2_EXCLUDE_INO 9     The "exclude" inode, for snapshots
// EXT4_REPLICA_INO 10    Used by non-upstream feature

// File types
// #define __S_IFDIR 0040000  /* Directory.  */
// #define __S_IFCHR 0020000  /* Character device.  */
// #define __S_IFBLK 0060000  /* Block device.  */
// #define __S_IFREG 0100000  /* Regular file.  */
// #define __S_IFIFO 0010000  /* FIFO.  */
// #define __S_IFLNK 0120000  /* Symbolic link.  */
// #define __S_IFSOCK 0140000 /* Socket.  */

// Proc status
#define PROC_FREE 0
#define PROC_BUSY 1

// file system table sizes
#define NUM_MINODES 100
#define NUM_MOUNT_ENTRIES 10
#define NUM_PROCS 2
#define NUM_OFT_PER 10
#define NUM_OFT 40

//// STRUCTS ////
```

```
// used to iterate over memory blocks of an inode
typedef struct blk_iter {
  struct minode *mip;
  // buf contains the nth block
  unsigned int lbkno;
  // direct block (buf), indirection block(map1),
  // double indirection(map2), triple indirection(map3);
  int map1[BLKSIZE_1024 / sizeof(int)], map2[BLKSIZE_1024 / sizeof(int)],
      map3[BLKSIZE_1024 / sizeof(int)];
  // block numbers of maps for writing
  int map1_bno, map2_bno, map3_bno;
} blk_iter;

// for parsing paths into
typedef struct path {
  char argv[256][64]; // count of strings
  int argc;           // array of strings
  bool is_absolute;
} path;

// In-memory inodes structure
typedef struct minode {
  // disk inode
  inode inode;
  // inode number
  int ino;
  // use count
  int ref_count;
  // modified flag
  bool dirty;
  // mount point transition
  struct mount_entry *mnt;
  // device containing inode
  struct mount_entry *dev;
  // ignored for simple FS
  // int lock;
} minode;

// Open file Table AKA opened file instance
typedef struct oft {
  // file mode
  int mode;
  // number of PROCs sharing this instance
  int ref_count;
  // pointer to minode of file
  minode *minode;
  // byte offset for R|W
  int offset;
  // for caching
  blk_iter it;
} oft;

// PROC structure
typedef struct proc {
  struct proc *next;
  int pid;
  int uid;
  int gid;
  int ppid;
  int status;
  minode *cwd;
  oft *oft_arr[NUM_OFT_PER];
```

```c
} proc;

// Mount Entry structure
typedef struct mount_entry {
  // device file descriptor
  int fd;
  // device root inode
  minode *mnt_pnt;
  // device path ex: ~/project/exampledisk
  char dev_path[64];
  // mount path ex: / for root, /A or /B or /C ... for non-root
  char mnt_path[64];
  // superblock
  super_block super_block;
  // group_desc
  group_desc group_desc;
} mount_entry;

// bmap == dev_gd->bg_block_bitmap;
// imap == dev_gd->bg_inode_bitmap;
// iblock == dev_gd->bg_inode_table;

//// VAR ////

// in memory  inodes
minode minode_arr[NUM_MINODES];

// root mounted inode
minode *global_root_inode;

// mount tables
mount_entry mount_entry_arr[NUM_MOUNT_ENTRIES];

mount_entry *global_root_mount;

// Opened file instance
oft oft_arr[NUM_OFT];

// PROC structures
proc proc_arr[NUM_PROCS];

// current executing PROC
proc *running;

//// FUNC ////

// fs_io
oft *alloc_oft();
bool free_oft(oft *);
int *get_lbk(blk_iter *, int);
int open_file(char *, int);
int lseek_file(int, int, int);
int close_file(int);
int read_file(int, void *, unsigned int);
int write_file(int, void *, unsigned int);

// fs_minode
minode *alloc_minode();
bool free_minode(minode *);
minode *get_minode(mount_entry *, int);
bool put_minode(minode *);
```

```
// fs_mount
int fs_init();
mount_entry *make_me(char *, char *);

// fs_path
int parse_path(char *, path *);
int search_dir(minode *, char *);
minode *search_path(path);
int list_dir(minode *, dir_entry *);
int count_dir(minode *);

// fs_util
int alloc_inode(mount_entry *);
int free_inode(mount_entry *, int);
int alloc_block(mount_entry *);
int free_block(mount_entry *, int);
int get_block(mount_entry *, int, char *);
int put_block(mount_entry *, int, char *);
int tst_bit(char *, int);
int set_bit(char *, int);
int clr_bit(char *, int);
int add_dir_entry(minode *, dir_entry *);
int rm_dir_entry(minode *, char *);
int free_i_block(minode *);
#endif
```

```c
#include "fs.h"

// allocate a free oft for use
oft *alloc_oft() {
  for (int i = 0; i < NUM_OFT; i++) {
    oft *op = &oft_arr[i];
    if (op->ref_count == 0) {
      op->ref_count = 1;
      return op;
    }
  }
  printf("panic:out of fd's\n");
  return NULL;
}

// release a used oft.
bool free_oft(oft *op) {
  op->ref_count = 0;
  return true;
}
int *add_lbk(blk_iter *it) {

  mount_entry *me = it->mip->dev;
  int last_bno = it->mip->inode.i_size / BLKSIZE_1024;
  get_lbk(it, last_bno);

  if (it->map1_bno) {
    for (int i = 0; i < BLKSIZE_1024 / sizeof(int); i++) {
      if (!it->map1[i]) {
        it->map1[i] = alloc_block(me);
        if (i + 1 < BLKSIZE_1024 / sizeof(int))
          it->map1[i + 1] = 0;
        put_block(me, it->map1_bno, (char *)it->map1);
        return &it->map1[i];
      }
    }
  }
  if (it->map2_bno) {
    for (int i = 0; i < BLKSIZE_1024 / sizeof(int); i++) {
      if (!it->map2[i]) {
        it->map2[i] = alloc_block(me);
        put_block(me, it->map2_bno, (char *)it->map2);
        it->map1_bno = it->map2[i];
        get_block(me, it->map1_bno, (char *)it->map1);
        it->map1[0] = alloc_block(me);
        it->map1[1] = 0;
        put_block(me, it->map1_bno, (char *)it->map1);
        return &it->map1[0];
      }
    }
  }
  if (it->map3_bno) {
    for (int i = 0; i < BLKSIZE_1024 / sizeof(int); i++) {
      if (!it->map3[i]) {
        it->map3[i] = alloc_block(me);
        put_block(me, it->map3_bno, (char *)it->map3);
        it->map2_bno = it->map3[i];
        get_block(me, it->map2_bno, (char *)it->map2);
        it->map2[0] = alloc_block(me);
        it->map2[1] = 0;
        put_block(me, it->map2_bno, (char *)it->map2);
        it->map1_bno = it->map2[0];
        get_block(me, it->map1_bno, (char *)it->map1);
        it->map1[0] = alloc_block(me);
        it->map1[1] = 0;
        put_block(me, it->map1_bno, (char *)it->map1);
        return &it->map1[0];
      }
    }
  }

  inode *ip = &it->mip->inode;
  for (int i = 0; i < 15; i++) {
    if (!ip->i_block[i]) {
      ip->i_block[i] = alloc_block(me); // always need one
      if (i == 12) {                    // one more blocks
        get_block(me, ip->i_block[12], (char *)it->map1);
        it->map1_bno = ip->i_block[12];
        it->map1[0] = alloc_block(me);
        it->map1[1] = 0;
        put_block(me, ip->i_block[12], (char *)it->map1);
        return &it->map1[0];
      } else if (i == 13) { // two more blocks
        get_block(me, ip->i_block[13], (char *)it->map2);
        it->map2_bno = ip->i_block[13];
        it->map2[0] = alloc_block(me);
        it->map2[1] = 0;
        put_block(me, ip->i_block[13], (char *)it->map2);
        get_block(me, it->map2[0], (char *)it->map1);
        it->map1_bno = it->map2[0];
        it->map1[0] = alloc_block(me);
        it->map1[1] = 0;
        put_block(me, it->map2[0], (char *)it->map1);
        return &it->map1[0];
      } else if (i == 14) { // three more blocks
        get_block(me, ip->i_block[14], (char *)it->map3);
        it->map3_bno = ip->i_block[14];
        it->map3[0] = alloc_block(me);
        it->map3[1] = 0;
        put_block(me, ip->i_block[14], (char *)it->map3);
        get_block(me, it->map3[0], (char *)it->map2);
        it->map2_bno = it->map3[0];
        it->map2[0] = alloc_block(me);
        it->map2[1] = 0;
        put_block(me, it->map3[0], (char *)it->map2);
        get_block(me, it->map2[0], (char *)it->map1);
        it->map1_bno = it->map2[0];
        it->map1[0] = alloc_block(me);
        it->map1[1] = 0;
        put_block(me, it->map2[0], (char *)it->map1);
        return &it->map1[0];
      }
    }
  }
}

// returns block number of logical block
// 0 on failure (nothing more to read)
// start from lbkno = -1
int *get_lbk(blk_iter *it, int target) {
  // calculations for convience, could be macros
  int blks_per = BLKSIZE_1024 / sizeof(int), *bno;
  int direct_start = 0, direct_end = 12, indirect_start = direct_end,
      indirect_end = direct_end + blks_per, double_start = indirect_end,
```

```c
      double_end = indirect_end + blks_per * blks_per,
      triple_start = double_end,
      triple_end = double_end + blks_per * blks_per * blks_per;
  // pointers for shorter names
  unsigned int *i_block = it->mip->inode.i_block;
  mount_entry *me = it->mip->dev;
  // null check
  if (!it || !it->mip)
    return 0;

  // get blocks based on target

  // get direct block
  if (target < direct_end) {
    it->map1_bno = it->map2_bno = it->map3_bno = 0;
    bno = &i_block[target];
  }
  // get indirect block
  else if (target < indirect_end) {
    it->map2_bno = it->map3_bno = 0;
    if (!(it->lbkno >= indirect_start && it->lbkno < indirect_end))
      // check if map1 cached
      get_block(me, it->map1_bno = i_block[12], (char *)it->map1);
    bno = &it->map1[target - indirect_start];
  }
  // get double indirect block
  else if (target < double_end) {
    it->map3_bno = 0;
    if (!(it->lbkno >= double_start && it->lbkno < double_end))
      // check if map2 cached
      get_block(me, it->map2_bno = i_block[13], (char *)it->map2);
    if (!((target - double_start) / blks_per ==
          (it->lbkno - double_start) / blks_per))
      // check if map1 cached
      get_block(me, it->map1_bno = it->map2[(target - double_start) / blks_per],
                (char *)it->map1);
    bno = &it->map1[(target - double_start) % blks_per];
  }
  // triple  indirect blocks
  else if (target < triple_end) {
    if (!(it->lbkno >= triple_start && it->lbkno < triple_end))
      // check if map3 cached
      get_block(me, it->map3_bno = i_block[14], (char *)it->map3);
    if (!((target - triple_start) / (blks_per * blks_per) ==
          (it->lbkno - triple_start) / (blks_per * blks_per)))
      // check if map2 cached
      get_block(me,
                it->map2_bno =
                    it->map3[(target - triple_start) / (blks_per * blks_per)],
                (char *)it->map2);
    if (!((target - triple_start) / blks_per ==
          (it->lbkno - triple_start) / blks_per))
      // check if map1 cached
      get_block(me, it->map1_bno = it->map2[(target - triple_start) / blks_per],
                (char *)it->map1);
    bno = &it->map1[(target - triple_start) % blks_per];
  }

  it->lbkno = target;
  return bno;
}
```

```c
// returns fd or -1 for fail
int open_file(char *path, int mode) {
  int fd;
  struct path p;
  parse_path(path, &p);
  minode *mip = NULL;
  if (!(mip = search_path(p)) || !S_ISREG(mip->inode.i_mode)) {
    put_minode(mip);
    return 0;
  }

  oft *oftp = alloc_oft();
  for (fd = 0; fd < NUM_OFT_PER; fd++) {
    if (running->oft_arr[fd] == NULL) {
      running->oft_arr[fd] = oftp;
      break;
    }
  }
  oftp->minode = mip;
  oftp->offset = 0;

  oftp->it.lbkno = -1;
  oftp->it.map1_bno = -1;
  oftp->it.map2_bno = -1;
  oftp->it.map3_bno = -1;
  oftp->it.mip = mip;

  // mode = 0|1|2|3 for R|W|RW|APPEND
  if (mode == 0) {
    oftp->minode->inode.i_atime = time(0L);
    oftp->mode = 0;
  } else if (mode == 1) {
    oftp->minode->inode.i_atime = oftp->minode->inode.i_mtime = time(0L);
    oftp->mode = 1;
    free_i_block(mip);
  } else if (mode == 2) {
    oftp->minode->inode.i_atime = oftp->minode->inode.i_mtime = time(0L);
    oftp->mode = 2;
  } else if (mode == 3) {
    oftp->minode->inode.i_atime = oftp->minode->inode.i_mtime = time(0L);
    oftp->mode = 3;
    oftp->offset = mip->inode.i_size;
  } else {
    put_minode(mip);
    free_oft(oftp);
    printf("Invalid file mode given\n");
    return -1;
  }

  for (int i = 0; i < NUM_OFT; i++) {
    if (oft_arr[i].ref_count && oft_arr[i].minode->ino == mip->ino &&
        &oft_arr[i] != oftp && (oft_arr[i].mode || oftp->mode)) {
      printf("Only allowed to write to unopened file\n");
      put_minode(mip);
      free_oft(oftp);
      return -1;
    }
  }
  mip->dirty = true;
  return fd;
}
```

```c
// return final offset or -1 for failure
int lseek_file(int fd, int offset, int whence) {
  oft *oftp;
  int og_off;
  if (fd < 0 || fd > NUM_OFT_PER)
    return -1;
  if ((oftp = running->oft_arr[fd])) {
    og_off = oftp->offset;
    // 0        SEEK_SET
    if (whence == 0)
      oftp->offset = offset;
    // 1        SEEK_CUR
    else if (whence == 1)
      oftp->offset += offset;
    // 2        SEEK_END
    else if (whence == 2)
      oftp->offset = oftp->minode->inode.i_size + offset;
    else
      return -1;
  }
  if (oftp->offset > oftp->minode->inode.i_size || oftp->offset < 0) {
    oftp->offset = og_off;
    return -1;
  }
  return oftp->offset;
}

// returns fd close or -1 for fail
int close_file(int fd) {
  if (fd > NUM_OFT_PER)
    return -1;
  if (running->oft_arr[fd]) {
    put_minode(running->oft_arr[fd]->minode);
    free_oft(running->oft_arr[fd]);
    running->oft_arr[fd] = NULL;
    return fd;
  }
  printf("fd not open\n");
  return -1;
}

int read_file(int fd, void *buf, unsigned int count) {
  char *dest = (char *)buf;
  char blk_buf[BLKSIZE_1024] = {0};
  oft *oftp = running->oft_arr[fd];
  if (!oftp || !(oftp->mode == 0 || oftp->mode == 2))
    return 0;
  int tar_lbk, avil, tar_byte, bno, to_copy, remain = 0, mid;
  avil = oftp->minode->inode.i_size - oftp->offset;
  while (count && avil) {
    // find logical block
    tar_lbk = oftp->offset / BLKSIZE_1024;
    // find offset from start of block
    tar_byte = oftp->offset % BLKSIZE_1024;
    // find offset from end of block
    remain = BLKSIZE_1024 - tar_byte;
    // get bno
    if (!(bno = *get_lbk(&oftp->it, tar_lbk)))
      return 0;
    // get full ass block
    get_block(oftp->minode->dev, bno, blk_buf);
    // figure out how much of block to copy
    to_copy = ((mid = (count > avil ? avil : count)) > remain) ? remain : mid;

    // copy that much of block
    memcpy(dest, blk_buf + tar_byte, to_copy);
    // increment buf pointer  by amount copied
    dest += to_copy;
    // increment offset by amount copied
    oftp->offset += to_copy;
    // decrement count by amount copied
    count -= to_copy;
    // decrement avil by amount copied
    avil -= to_copy;
  }
  return dest - (char *)buf;
}

int write_file(int fd, void *buf, unsigned int count) {
  char *src = (char *)buf;
  char blk_buf[BLKSIZE_1024] = {0};
  oft *oftp = running->oft_arr[fd];
  if (!oftp || !(oftp->mode == 1 || oftp->mode == 2))
    return 0;
  int tar_lbk, tar_byte, *bnop, to_copy, remain = 0, mid;
  while (count) {
    // find logical block
    tar_lbk = oftp->offset / BLKSIZE_1024;
    // find offset from start of block
    tar_byte = oftp->offset % BLKSIZE_1024;
    // find offset from end of block
    remain = BLKSIZE_1024 - tar_byte;
    // get bno and alloc
    bnop = get_lbk(&oftp->it, tar_lbk);
    if (!*bnop) {
      bnop = add_lbk(&oftp->it);
    }
    // figure out how much of block to copy
    to_copy = (count > remain) ? remain : count;

    // read full ass block if needed
    if (to_copy < BLKSIZE_1024)
      get_block(oftp->minode->dev, *bnop, blk_buf);

    // copy that much to block
    memcpy(blk_buf + tar_byte, src, to_copy);
    put_block(oftp->minode->dev, *bnop, blk_buf);
    // increment buf pointer  by amount copied
    src += to_copy;
    // increment offset by amount copied
    oftp->offset += to_copy;
    // decrement count by amount copied
    count -= to_copy;
    // increase file size
    oftp->minode->inode.i_size = ((oftp->offset > oftp->minode->inode.i_size)
                                      ? oftp->offset
                                      : oftp->minode->inode.i_size);
  }
  return src - (char *)buf;
}
```

```c
#include "fs.h"

// allocate a free minode for use
minode *alloc_minode() {
  for (int i = 0; i < NUM_MINODES; i++) {
    minode *mp = &minode_arr[i];
    if (mp->ref_count == 0) {
      mp->ref_count = 1;
      return mp;
    }
  }
  printf("panic:out of minodes\n");
  return NULL;
}

// release a used minode. However, root is always referenced.
bool free_minode(minode *mip) {
  mip->ref_count = mip == global_root_inode ? 1 : 0;
  return true;
}

// Returns a pointer to the in-memory minode containing the INODE of (me, ino).
minode *get_minode(mount_entry *dev, int ino) {
  minode *mip;
  inode *ip;
  int i, block, offset;
  char buf[BLKSIZE_1024];
  // search in-memory minodes first
  for (i = 0; i < NUM_MINODES; i++) {
    minode *mip = &minode_arr[i];
    if (mip->ref_count && (mip->dev == dev) && (mip->ino == ino)) {
      mip->ref_count++;
      return mip;
    }
  }
  mip = alloc_minode();
  block = (ino - 1) / 8 + dev->group_desc.bg_inode_table;
  offset = (ino - 1) % 8;
  get_block(dev, block, buf);
  ip = (inode *)buf + offset;
  // initialize minode
  *mip = (minode){
      .inode = *ip,
      .ino = ino,
      .ref_count = 1,
      .dirty = 0,
      .mnt = NULL,
      .dev = dev,
  };

  return mip;
}

// Decrements the ref_count by 1. If the refCount is zero then
// inode is written back to disk if it is modified (dirty).
bool put_minode(minode *mip) {
  inode *ip;
  int i, block, offset;
  char buf[BLKSIZE_1024];
  if (mip == 0)
    return false;
  mip->ref_count--;
  if (mip->ref_count > 0)
    return false;
  // ROOT NEVER FREE. ROOT STAY FOREVER
  if (global_root_inode->ref_count < 1)
    global_root_inode->ref_count = 1;
  if (mip->dirty == 0)
    return false;
  block = (mip->ino - 1) / 8 + mip->dev->group_desc.bg_inode_table;
  offset = (mip->ino - 1) % 8;
  // get block containing this inode
  get_block(mip->dev, block, buf);
  ip = (inode *)buf + offset;
  *ip = mip->inode;
  put_block(mip->dev, block, buf);
  free_minode(mip);
  return true;
}
```

```c
#include "fs.h"

int fs_init() {
  int i, j;
  // initialize all minodes
  for (i = 0; i < NUM_MINODES; i++)
    minode_arr[i].ref_count = 0;
  // initialize mount entries
  for (i = 0; i < NUM_MOUNT_ENTRIES; i++)
    mount_entry_arr[i].fd = 0;
  // initialize PROCs
  for (i = 0; i < NUM_PROCS; i++) {
    proc_arr[i].status = PROC_FREE;
    proc_arr[i].pid = i;
    // P0 is a superuser process
    proc_arr[i].uid = i;
    // initialize PROC file descriptors to NULL
    for (j = 0; j < NUM_OFT_PER; j++)
      proc_arr[i].oft_arr[j] = 0;
    proc_arr[i].next = &proc_arr[i + 1];
  }
  // circular list
  proc_arr[NUM_PROCS - 1].next = &proc_arr[0];
  // P0 runs first
  running = &proc_arr[0];
  return 0;
}

// returns a globally allocated mount_entry pointer
// null on failure
// mount_entry->mnt_pnt is set to root minode by default
mount_entry *make_me(char *dev_path, char *mnt_path) {

  // open 'device'
  int dev = open(dev_path, O_RDWR);
  if (dev < 0) {
    printf("panic : can’t open device\n");
    return NULL;
  }

  // alloc mount entry
  int meno;
  mount_entry *me;
  for (meno = 0; meno < NUM_MOUNT_ENTRIES + 1; meno++) {
    if (meno == NUM_MOUNT_ENTRIES) {
      printf("panic: cannot mount");
      return NULL;
    }
    me = &mount_entry_arr[meno];
    if (!me->fd)
      break;
  }

  // set fd and names
  me->fd = dev;
  strcpy(me->dev_path, dev_path);
  strcpy(me->mnt_path, mnt_path);

  // get super block to me
  char buf[BLKSIZE_1024];
  get_block(me, 1, buf);
  me->super_block = *(super_block *)buf;
  // check magic number
  if (me->super_block.s_magic != EXT2_SUPER_MAGIC) {
    printf("not an EXT2 filesystem please umount\n");
    exit(0);
  }

  // get group descriptor to me
  get_block(me, 2, buf);
  me->group_desc = *(group_desc *)buf;

  // init mnt pnt to NULL
  me->mnt_pnt = NULL;

  DEBUG_PRINT("mounted %s to %s with fd %d\n", me->dev_path, me->mnt_path,
              me->fd);
  return me;
}
```

```c
#include "fs.h"

// set if path relative/absolute/root in buf
// split path_name on "/" into argv and argc of buf
// return argc
int parse_path(char *path_name, path *buf_path) {
  char *s, safe_name[256];
  strcpy(safe_name, path_name);
  buf_path->argc = 0;

  // check if absolute or relative
  if (safe_name[0] == '/')
    buf_path->is_absolute = true;
  else
    buf_path->is_absolute = false;

  // split into components
  s = strtok(safe_name, "/");
  while (s) {
    strcpy(buf_path->argv[buf_path->argc++], s);
    s = strtok(NULL, "/");
  }
  buf_path->argv[buf_path->argc][0] = 0;
  return buf_path->argc;
}

// iterates through i_block of mip
// return ino of dir with dir_name on success
// return 0 on failure
int search_dir(minode *mip, char *dir_name) {
  int i;
  char *fs_p, temp[256], buf[BLKSIZE_1024] = {0}, *b = buf;
  dir_entry *dep;
  DEBUG_PRINT("search for %s\n", dir_name);
  if (!S_ISDIR(mip->inode.i_mode)) {
    DEBUG_PRINT("search fail %s is not a dir\n", dir_name);
    return 0;
  }
  // search dir_entry direct blocks only
  for (i = 0; i < 12; i++) {
    // if direct block is null stap
    if (mip->inode.i_block[i] == 0)
      return 0;
    // get next direct block
    get_block(mip->dev, mip->inode.i_block[i], buf);
    dep = (dir_entry *)buf;
    fs_p = buf;
    while (fs_p < buf + BLKSIZE_1024) {
      snprintf(temp, dep->name_len + 1, "%s", dep->name);
      DEBUG_PRINT("ino:%d rec_len:%d name_len:%u name:%s\n", dep->inode,
                  dep->rec_len, dep->name_len, temp);
      if (strcmp(dir_name, temp) == 0) {
        DEBUG_PRINT("found %s : inumber = %d\n", dir_name, dep->inode);
        return dep->inode;
      }
      fs_p += dep->rec_len;
      dep = (dir_entry *)fs_p;
    }
  }
  return 0;
}
```

```c
// iterate through i_block of mip and store in dir_arr
// return dirc on success, return 0 on failure
int list_dir(minode *mip, dir_entry *dir_arr) {
  char *fs_p, buf[BLKSIZE_1024];
  dir_entry *dirp;
  int dirc = 0;
  if (!S_ISDIR(mip->inode.i_mode)) {
    DEBUG_PRINT("list fail ino %d is not a dir\n", mip->ino);
    return 0;
  }
  for (int i = 0; i < 12; i++) { // search direct blocks only
    if (mip->inode.i_block[i] == 0)
      return dirc;
    get_block(mip->dev, mip->inode.i_block[i], buf);
    dirp = (dir_entry *)buf;
    fs_p = buf;
    while (fs_p < buf + BLKSIZE_1024) {
      dirp = (dir_entry *)fs_p;
      dir_arr[dirc] = *dirp;
      dirc++;
      fs_p += dirp->rec_len;
    }
  }
  return dirc;
}

// returns count of dir entries in mip on success
// returns 0 on failure
int count_dir(minode *mip) {
  char buf[BLKSIZE_1024], *bufp = buf, temp[256];
  dir_entry *dirp;
  int dirc = 0;
  if (!S_ISDIR(mip->inode.i_mode))
    return 0;
  for (int i = 0; i < 12; i++) { // search direct blocks only
    if (mip->inode.i_block[i] == 0)
      return dirc;
    get_block(mip->dev, mip->inode.i_block[i], buf);
    dirp = (dir_entry *)buf;
    bufp = buf;
    while (bufp < buf + BLKSIZE_1024) {
      snprintf(temp, dirp->name_len + 1, "%s", dirp->name);
      DEBUG_PRINT("ino:%d rec_len:%d name_len:%u name:%s\n", dirp->inode,
                  dirp->rec_len, dirp->name_len, temp);
      dirc++;
      bufp += dirp->rec_len;
      dirp = (dir_entry *)bufp;
    }
  }
  return dirc;
}

// returns minode of path on success
// returns NULL on failure
// does not put found minode
minode *search_path(path target_path) {
  minode *prev_mip, *mip = global_root_inode;
  int ino;
  if (!target_path.is_absolute)
    mip = running->cwd; // if relative
  mip->ref_count++;
  // search for each token string
  for (int i = 0; i < target_path.argc; i++) {

    // find component
    ino = search_dir(mip, target_path.argv[i]);

    // special case
    // traverse up to mnt point from mnt device
    if (ino == 2 && mip->ino == 2) {
      minode *newmip = mip->dev->mnt_pnt;
      put_minode(mip);
      mip = newmip;
    }

    // bad path
    if (!ino) {
      DEBUG_PRINT("no such component name %s\n", target_path.argv[i]);
      put_minode(mip);
      return NULL;
    }

    minode *prev_mip = mip;
    mip = get_minode(mip->dev, ino);

    // traverse down to mnt device
    if (mip->mnt) {
      minode *newmip = get_minode(mip->mnt, 2);
      mip = newmip;
    }

    if (S_ISLNK(mip->inode.i_mode)) { // handle symlink
      if (i == target_path.argc - 1)  // if last entry return symlink
        return mip;
      path sym_path;
      parse_path((char *)mip->inode.i_block, &sym_path);
      if (sym_path.is_absolute) { // recurse and continue iteration on new mip
        put_minode(mip);
        mip = search_path(sym_path);
      } else // append sym_path to target_path and continue iteration
      {
        memcpy(&target_path.argv[i + sym_path.argc], &target_path.argv[i + 1],
               sizeof(char *) * sym_path.argc);
        memcpy(&target_path.argv[i], sym_path.argv,
               sizeof(char *) * sym_path.argc);
        target_path.argc += sym_path.argc - 1;
        i--;
        put_minode(mip);
        mip = prev_mip;
        continue;
      }
    }
    put_minode(prev_mip);
  }
  return mip;
}
```

```c
#include "fs.h"
#include "string.h"
#include <unistd.h>

//// ALLOC AND FREE

// returns the ino of the next available inode in inode_bitmap
// returns 0 if no more inodes, modifies inode_bitmap
int alloc_inode(mount_entry *me) {
  char buf[BLKSIZE_1024];
  get_block(me, me->group_desc.bg_inode_bitmap, buf);
  for (int i = 0; i < me->super_block.s_inodes_count; i++) {
    if (tst_bit(buf, i) == 0) {
      set_bit(buf, i);
      me->group_desc.bg_free_inodes_count--;
      put_block(me, me->group_desc.bg_inode_bitmap, buf);
      return i + 1;
    }
  }
  return 0;
}

// Marks the given ino in inode_bitmap as available
// returns 1
int free_inode(mount_entry *me, int ino) {
  char buf[BLKSIZE_1024];
  get_block(me, me->group_desc.bg_inode_bitmap, buf);
  clr_bit(buf, ino - 1);
  me->group_desc.bg_free_inodes_count++;
  put_block(me, me->group_desc.bg_inode_bitmap, buf);
  return 1;
}

// returns bno of next available block in block_bitmap
// returns 0 if out of blocks, modifies block_bitmap
int alloc_block(mount_entry *me) {
  char buf[BLKSIZE_1024];

  // read block_bitmap block
  get_block(me, me->group_desc.bg_block_bitmap, buf);

  for (int i = 0; i < me->super_block.s_blocks_count; i++) {
    if (tst_bit(buf, i) == 0) {
      set_bit(buf, i);
      me->group_desc.bg_free_blocks_count--;
      put_block(me, me->group_desc.bg_block_bitmap, buf);
      return i + 1;
    }
  }
  return 0;
}

// Marks the given bno in block_bitmap as available
int free_block(mount_entry *me, int bno) {
  char buf[BLKSIZE_1024];
  get_block(me, me->group_desc.bg_block_bitmap, buf);
  clr_bit(buf, bno - 1);
  me->group_desc.bg_free_blocks_count++;
  put_block(me, me->group_desc.bg_block_bitmap, buf);
  return 1;
}
```

```c
// GET PUT BLOCK

// read block to buf from disk
// return 1 on success, 0 on failure
int get_block(mount_entry *me, int bno, char *buf) {
  lseek(me->fd, bno * BLKSIZE_1024, SEEK_SET);
  int n = read(me->fd, buf, BLKSIZE_1024);
  if (n < 0) {
    printf("get_block[% d % d] error \n", me->fd, bno);
    return 0;
  }
  return 1;
}

// write block from buf to disk
// return 1 on success, 0 on failure
int put_block(mount_entry *me, int bno, char *buf) {
  lseek(me->fd, bno * BLKSIZE_1024, SEEK_SET);
  int n = write(me->fd, buf, BLKSIZE_1024);
  if (n != BLKSIZE_1024) {
    printf("put_block [%d %d] error\n", me->fd, bno);
    return 0;
  }
  return 1;
}

// BIT OPERATIONS

// tests the nth bit of buf
// return value of bit
int tst_bit(char *buf, int bit) {
  int i, j;
  i = bit / 8;
  j = bit % 8;
  if (buf[i] & (1 << j))
    return 1;
  return 0;
}

// sets the nth bit of buf to 1
// returns 1
int set_bit(char *buf, int bit) {
  int i, j;
  i = bit / 8;
  j = bit % 8;
  buf[i] |= (1 << j);
  return 1;
}

// sets the nth bit of buf to 0
// returns 1
int clr_bit(char *buf, int bit) {
  int i, j;
  i = bit / 8;
  j = bit % 8;
  buf[i] &= ~(1 << j);
  return 1;
}

//// ADD REMOVE DIR

// returns the closest size as a multiple of 4 which can contain *dirp
```

```c
int ideal_len(dir_entry *dirp) {
  int ideal = 4 * ((8 + dirp->name_len + 3) / 4);
  return ideal;
}

// creates new_dirp in mip's i_block[]
// new_dirp must have name, name_len, and inode set
// return rec_len on success, 0 on failure
// increments mip link count, but does not put inode
int add_dir_entry(minode *mip, dir_entry *new_dirp) {
  char buf[BLKSIZE_1024], *bufp = buf, name[256];
  dir_entry *cur_dirp;
  int free_space;
  snprintf(name, new_dirp->name_len + 1, "%s", new_dirp->name);
  if (search_dir(mip, name)) {
    printf("dir_entry by name of %s already exists\n", name);
    return 0;
  }

  // check if dir
  if (!S_ISDIR(mip->inode.i_mode)) {
    printf("cannot mkdir in a file\n");
    return 0;
  }

  // set new_dirp rec_len to ideal
  new_dirp->rec_len = ideal_len(new_dirp);

  // iterate through direct blocks
  for (int i = 0; i < 12; i++) {
    // if allocating a new block insert record as first entry
    if (mip->inode.i_block[i] == 0) {
      mip->inode.i_block[i] = alloc_block(mip->dev);
      get_block(mip->dev, mip->inode.i_block[i], buf);
      cur_dirp = (dir_entry *)buf;
      *cur_dirp = *new_dirp;
      cur_dirp->rec_len = BLKSIZE_1024;
      put_block(mip->dev, mip->inode.i_block[i], buf);
      mip->inode.i_links_count++;
      return cur_dirp->rec_len;
    }
    // else
    get_block(mip->dev, mip->inode.i_block[i], buf);
    bufp = buf;
    // iterate through dir_entries to find space
    while (bufp < buf + BLKSIZE_1024) {
      cur_dirp = (dir_entry *)bufp;

      // check if space to insert then break
      free_space = cur_dirp->rec_len - ideal_len(cur_dirp);
      if (free_space > new_dirp->rec_len) {
        cur_dirp->rec_len = ideal_len(cur_dirp);
        bufp += cur_dirp->rec_len;
        int new_dirp_size = new_dirp->rec_len;
        new_dirp->rec_len = free_space;
        // using a memcpy here avoids over-writing the end of the buffer
        memcpy(bufp, new_dirp, new_dirp->rec_len);
        // write buffer back to block
        put_block(mip->dev, mip->inode.i_block[i], buf);
        mip->inode.i_links_count++;
        return new_dirp->rec_len;
      }
      bufp += cur_dirp->rec_len;
    }
  }
  return 0;
}

// removes dir with dir.name equal dir_name from mip
// return rec_len of last dir on success, 0 on failure
// decrements mip link_count, does not put
int rm_dir_entry(minode *mip, char *dir_name) {
  int i;
  char buf[BLKSIZE_1024], *bufp, *prev;
  char str[256];
  dir_entry *dep;
  int freed_space;
  if (!S_ISDIR(mip->inode.i_mode)) {
    DEBUG_PRINT("attempt to remove non-dir");
    return 0;
  }
  // search dir_entry direct blocks only
  for (i = 0; i < 12; i++) {
    if (mip->inode.i_block[i] == 0) {
      DEBUG_PRINT("dir_entry not found");
      return 0;
    }
    get_block(mip->dev, mip->inode.i_block[i], buf);
    dep = (dir_entry *)buf;
    bufp = buf;
    while (bufp < buf + BLKSIZE_1024) {
      snprintf(str, dep->name_len + 1, "%s", dep->name);
      if (strcmp(dir_name, str) == 0) {
        // if it's the only entry
        if (bufp == buf) {
          free_block(mip->dev, mip->inode.i_block[i]);
          mip->inode.i_block[i] = 0;
          // if last entry
        } else if (bufp + dep->rec_len >= buf + BLKSIZE_1024) {
          ((dir_entry *)prev)->rec_len += dep->rec_len;
          // if middle entry
        } else {
          dep = (dir_entry *)bufp;
          freed_space = dep->rec_len;
          // copy everything in block after current record onto current record
          memcpy(bufp, bufp + dep->rec_len,
                 (buf + BLKSIZE_1024) - (bufp + dep->rec_len));
          // find last record
          while (bufp + dep->rec_len < buf + BLKSIZE_1024 - freed_space) {
            bufp += dep->rec_len;
            dep = (dir_entry *)bufp;
          }
          // give him some extra room
          dep->rec_len += freed_space;
        }
        put_block(mip->dev, mip->inode.i_block[i], buf);
        mip->inode.i_links_count--;
        mip->inode.i_atime = mip->inode.i_ctime = mip->inode.i_mtime = time(0L);
        mip->dirty = true;
        return dep->rec_len;
      }
      prev = bufp;
      bufp += dep->rec_len;
      dep = (dir_entry *)bufp;
    }
  }
  return 0;
}

//// MISC

// frees all blocks (direct,indirect, etc..) from mip->inode.i_block[]
// return num blocks freed
int free_i_block(minode *mip) {
  char buf1[BLKSIZE_1024], buf2[BLKSIZE_1024], buf3[BLKSIZE_1024];
  int *fs_p1, *fs_p2, *fs_p3, freed_blocks = 0;
  path in_path;
  // direct blocks
  for (int i = 0; i < 12 && mip->inode.i_block[i]; i++)
    freed_blocks += free_block(mip->dev, mip->inode.i_block[i]);

  // indirect blocks
  if (!mip->inode.i_block[12])
    return freed_blocks;
  get_block(mip->dev, mip->inode.i_block[12], buf1);
  fs_p1 = (int *)buf1;
  while (*fs_p1 && ((char *)fs_p1 < buf1 + BLKSIZE_1024)) {
    freed_blocks += free_block(mip->dev, *fs_p1);
    fs_p1++;
  }

  // double indirect blocks
  if (!mip->inode.i_block[13])
    return freed_blocks;
  get_block(mip->dev, mip->inode.i_block[13], buf1);
  fs_p1 = (int *)buf1;
  while (*fs_p1 && ((char *)fs_p1 < buf1 + BLKSIZE_1024)) {
    get_block(mip->dev, *fs_p1, buf2);
    fs_p2 = (int *)buf2;
    while (*fs_p2 && ((char *)fs_p2 < buf2 + BLKSIZE_1024))
      freed_blocks += free_block(mip->dev, *fs_p2);
    fs_p1++;
  }

  // triple indirect blocks
  if (!mip->inode.i_block[14])
    return freed_blocks;
  get_block(mip->dev, mip->inode.i_block[14], buf1);
  fs_p1 = (int *)buf1;
  while (*fs_p1 && ((char *)fs_p1 < buf1 + BLKSIZE_1024)) {
    get_block(mip->dev, *fs_p1, buf2);
    fs_p2 = (int *)buf2;
    while (*fs_p2 && ((char *)fs_p2 < buf2 + BLKSIZE_1024)) {
      get_block(mip->dev, *fs_p2, buf3);
      fs_p3 = (int *)buf3;
      while (*fs_p3 && ((char *)fs_p3 < buf3 + BLKSIZE_1024))
        freed_blocks += free_block(mip->dev, *fs_p3);
      fs_p2++;
    }
    fs_p1++;
  }

  put_minode(mip);
  return freed_blocks;
}
```

```c
#include "cmd/cmd.h"
#include "fs/fs.h"

int main(int argc, char const *argv[]) {
  char line[128], *root_dev;
  cmd c, *user_cmd = &c;
  if (argc > 1)
    root_dev = (char *)argv[1];

  // init globals
  fs_init();
  // read device meta data
  mount_entry *me = make_me(root_dev, "/");
  // make root mnt pnt be root
  me->mnt_pnt = get_minode(me, 2);
  global_root_inode = me->mnt_pnt;

  for (int i = 0; i < NUM_PROCS; i++)
    proc_arr[i].cwd = global_root_inode;

  // progam loop
  for (;;) {
    DEBUG_PRINT("running->pid == %d\n", running->pid);

    // prompt and read
    printf("User:%d @ ", running->uid);
    _pwd(running->cwd);
    printf(": ");

    fgets(line, 128, stdin);
    line[strlen(line) - 1] = 0;
    if (!line[0])
      continue;

    // split into argv argc
    parse_cmd(line, user_cmd);

    for (int i = 0; i < user_cmd->argc; i++)
      DEBUG_PRINT("user_cmd->argv[%d] == %s\n", i, user_cmd->argv[i]);

    // execute use command
    do_cmd(user_cmd);
  }
}
```