CPTS 360 Final Project

Nathan Colley
Jeff Kohls

April 24, 2013

Contents

1 Organization

Our project was divided into two major parts. First was what we came to term the "kernel", which was responsible for all inode, m_inode, block, and disk I/O functions. The kernel is in the base directory and has a struct name of k_t.

The second part of our design was meant to mimic the Linux kernel's user space commands and VFS table. These functions are grouped into level zero, level one, and level two respectively. We did not attempt to complete level three but much of the infrastructure is in place to do so. With some additional time we are confident that we could complete this part of the assignment.

Level zero, while not part of the assignment specification, contains a number of additional functions we found useful while debugging our "kernel" and user space features. It is included here for completeness.

2 Core files

2.1 Makefile

Listing 1: Our makefile

```
1  CC = gcc
2  CFLAGS = -g -Wall -Wno-unused-label -Wno-parentheses -I include/
3  LIBS =
4  ONAME = sim
5
6  OBJS = shell.o \
7         proc_t.o \
8         of_t.o \
9         mino_t.o \
10        mnt_t.o \
11        k_t.o \
12        vsh_t.o \
13        string_funcs.o \
14        func_addresses.o \
15        test_func.o \
16        show_help.o \
17        print_minos.o \
18        print_vfs_mounts.o \
19        show_imap.o \
20        show_bmap.o \
21        print_ino.o \
22        print_procs.o \
23        switch.o \
24        ls.o \
25        cd.o \
26        creat.o \
27        mkdir.o \
28        pwd.o \
29        link.o \
30        unlink.o \
31        rmdir.o\
32        symblink.o\
33        stat.o \
34        touch.o \
35        chmod.o \
36        chgrp.o \
37        chown.o \
38        open.o \
39        close.o \
40        lseek.o \
41        pfd.o \
42        read.o \
43        cat.o \
44        mv.o \
45        write.o \
46        cp.o
47
48
49 shell: ${OBJS}
50     ${CC} -o ${ONAME} ${CFLAGS} ${OBJS} ${LIBS}
51
52 clean:
53     rm -f *.o
54     rm -f l0/*.o
55     rm -f l1/*.o
```

```
56      rm -f l2/*.o
57      rm -f ${ONAME}
58
59  #-------------------------------------------------
60
61  shell.o: shell.c shell.h
62      ${CC} ${CFLAGS} -c shell.c
63
64  proc_t.o: proc_t.c proc_t.h
65      ${CC} ${CFLAGS} -c proc_t.c
66
67  of_t.o: of_t.c of_t.h
68      ${CC} ${CFLAGS} -c of_t.c
69
70  mino_t.o: mino_t.c mino_t.h
71      ${CC} ${CFLAGS} -c mino_t.c
72
73  mnt_t.o: mnt_t.c mnt_t.h
74      ${CC} ${CFLAGS} -c mnt_t.c
75
76  k_t.o: k_t.c k_t.h
77      ${CC} ${CFLAGS} -c k_t.c
78
79  vsh_t.o: vsh_t.c vsh_t.h
80      ${CC} ${CFLAGS} -c vsh_t.c
81
82  string_funcs.o: string_funcs.c string_funcs.h
83      ${CC} ${CFLAGS} -c string_funcs.c
84
85  func_addresses.o: func_addresses.c func_addresses.h
86      ${CC} ${CFLAGS} -c func_addresses.c
87
88  show_help.o: l0/show_help.c l0/show_help.h
89      ${CC} ${CFLAGS} -c l0/show_help.c
90
91  print_minos.o: l0/print_minos.c l0/print_minos.h
92      ${CC} ${CFLAGS} -c l0/print_minos.c
93
94  print_vfs_mounts.o: l0/print_vfs_mounts.c l0/print_vfs_mounts.h
95      ${CC} ${CFLAGS} -c l0/print_vfs_mounts.c
96
97  test_func.o: l0/test_func.c l0/test_func.h
98      ${CC} ${CFLAGS} -c l0/test_func.c
99
100 show_imap.o: l0/show_imap.c l0/show_imap.h
101     ${CC} ${CFLAGS} -c l0/show_imap.c
102
103 show_bmap.o: l0/show_bmap.c l0/show_bmap.h
104     ${CC} ${CFLAGS} -c l0/show_bmap.c
105
106 print_ino.o: l0/print_ino.c l0/print_ino.h
107     ${CC} ${CFLAGS} -c l0/print_ino.c
108
109 print_procs.o: l0/print_procs.c l0/print_procs.h
110     ${CC} ${CFLAGS} -c l0/print_procs.c
111
112 switch.o: l0/switch.c l0/switch.h
113     ${CC} ${CFLAGS} -c l0/switch.c
114
115 ls.o: l1/ls.c l1/ls.h
116     ${CC} ${CFLAGS} -c l1/ls.c
117
118 cd.o: l1/cd.c l1/cd.h
119     ${CC} ${CFLAGS} -c l1/cd.c
120
121 mkdir.o: l1/mkdir.c l1/mkdir.h
122     ${CC} ${CFLAGS} -c l1/mkdir.c
123
124 creat.o: l1/creat.c l1/creat.h
125     ${CC} ${CFLAGS} -c l1/creat.c
126
127 pwd.o: l1/pwd.c l1/pwd.h
128     ${CC} ${CFLAGS} -c l1/pwd.c
129
130 link.o: l1/link.c l1/link.h
131     ${CC} ${CFLAGS} -c l1/link.c
132
133 unlink.o: l1/unlink.c l1/unlink.h
134     ${CC} ${CFLAGS} -c l1/unlink.c
135
136 rmdir.o: l1/rmdir.c l1/rmdir.h
```

```
137     ${CC} ${CFLAGS} -c l1/rmdir.c
138
139 symblink.o: l1/symblink.c l1/symblink.h
140     ${CC} ${CFLAGS} -c l1/symblink.c
141
142 stat.o: l1/stat.c l1/stat.h
143     ${CC} ${CFLAGS} -c l1/stat.c
144
145 touch.o: l1/touch.c l1/touch.h
146     ${CC} ${CFLAGS} -c l1/touch.c
147
148 chmod.o: l1/chmod.c l1/chmod.h
149     ${CC} ${CFLAGS} -c l1/chmod.c
150
151 chgrp.o: l1/chgrp.c l1/chgrp.h
152     ${CC} ${CFLAGS} -c l1/chgrp.c
153
154 chown.o: l1/chown.c l1/chown.h
155     ${CC} ${CFLAGS} -c l1/chown.c
156
157 open.o: l2/open.c l2/open.h
158     ${CC} ${CFLAGS} -c l2/open.c
159
160 close.o: l2/close.c l2/close.h
161     ${CC} ${CFLAGS} -c l2/close.c
162
163 lseek.o: l2/lseek.c l2/lseek.h
164     ${CC} ${CFLAGS} -c l2/lseek.c
165
166 pfd.o: l2/pfd.c l2/pfd.h
167     ${CC} ${CFLAGS} -c l2/pfd.c
168
169 read.o: l2/read.c l2/read.h
170     ${CC} ${CFLAGS} -c l2/read.c
171
172 cat.o: l2/cat.c l2/cat.h
173     ${CC} ${CFLAGS} -c l2/cat.c
174
175 mv.o: l2/mv.c l2/mv.h
176     ${CC} ${CFLAGS} -c l2/mv.c
177
178 write.o: l2/write.c l2/write.h
179     ${CC} ${CFLAGS} -c l2/write.c
180
181 cp.o: l2/cp.c l2/cp.h
182     ${CC} ${CFLAGS} -c l2/cp.c
```

2.2 Global definitions

Listing 2: Global definitions

```
1  /*****************************************************************************
2  file        : global_defs.h
3  author      : nc
4  desc        : system-wide
5  date        : 03-24-13
6   *****************************************************************************/
7
8  #pragma once
9
10 /*includes*/
11 #include <stdbool.h>
12 #include <stdint.h>
13
14 /*defines*/
15 #define false           0       //we use these a lot
16 #define true            1
17 #define NUM_FDS         128     //max number of open file descriptors
18 #define MAX_NAME        256     //max number of chars in a name
19 #define MAX_TMP         1024    //max size of tmp arrays, mostly used in string funcs
20 #define MAX_PATH        1024    //max size of the path
21 #define MAX_DEPTH       256     //how deep the file tree can be
22 #define DBLOCK          1440
23
24 #define BLK_SZ          1024    //size of a block in bytes
25 #define IE_SZ           512     //inode entry size
26 #define SB_BLK          1       //superblock offset
27 #define GD0_BLK         2       //group descriptor 0
28 #define EXT2_MAGIC      0xef53
```

```
29  #define INO_PER_BLK     8       //inodes in a block
30  #define ID_PER_BLK      256     //number of indirect inodes in a block
31  #define INO_SIZE        128     //size of inode in bytes
32  #define FILE_TYPE       1
33  #define DIR_TYPE        2
34
35  #define ROOT_INO_NUM    2
36  #define MAX_DIRECT      12      //offsets in the inode for block levels
37  #define ONE_INDIR       13
38  #define TWO_INDIR       14
39  #define THREE_INDIR     15
40  #define MAX_ENT         15
41  #define MAX_DBLOCKS     65805   //max number of double indirect
42  #define DIR_RANGE       12      //some more data block range calulators
43  #define OID_RANGE       268
44  #define TID_RANGE       65548
45
46  #define DIR_TOK         "/"
47
48  #define DEF_DMODE       0x41ed  //mode types for files and directories
49  #define DEF_FMODE       0x81a4
50  #define DEF_LMODE       0xa000
51  #define DEF_DLINKS      2
52
53  /*process status*/
54  #define S_STOPPED       0
55  #define S_RUNNING       1
56  #define S_KILLED        2
57
58  /*user ids*/
59  #define U_SUPER         0
60  #define U_ROOT          1
61  #define U_ADMIN         2
62  #define U_REG           3
63
64  /*exit codes*/
65  #define X_CRITICAL      -1      //we didn't use these as much as we intended
66  #define X_SUCCESS       0
67  #define X_NO_KERNEL     1
68  #define X_NO_SHELL      2
69  #define X_NO_MEMORY     3
70  #define X_DONE          4
71  #define X_NO_ROOT       5
72  #define X_NO_PROCS      6
73  #define X_NO_FTABLE     7
74  #define X_UNKNOWN       255
75
76  /*this is the mode that the file is opened in*/
77  typedef enum
78  {
79      M_READ,
80      M_WRITE,
81      M_READWRITE,
82      M_APPEND
83  } E_FILE_MODE;
84
85
86  /*typedefs*/
87  typedef struct ext2_group_desc      gd;
88  typedef struct ext2_super_block     sb;
89  typedef struct ext2_inode           inode;
90  typedef struct ext2_dir_entry_2     ext2_dir;
91
92  typedef int8_t      i8;
93  typedef int16_t     i16;
94  typedef int32_t     i32;
95  typedef int64_t     i64;
96  typedef uint8_t     u8;
97  typedef uint16_t    u16;
98  typedef uint32_t    u32;
99  typedef uint64_t    u64;
```

2.3 Kernel

2.3.1 Definition

Listing 3: k_t.h

```
/*****************************************************************************
file        : k_t.h
author      : nc
desc        : k_t struct
date        : 03-25-13
 *****************************************************************************/

#pragma once

#include "global_defs.h"
#include "proc_t.h"
#include "mnt_t.h"
#include "mino_t.h"
#include "of_t.h"

/*fw dec*/
typedef struct k_t k_t;


/*defines*/
#define MAX_PROCS   64      //maximum number of running processes
#define MAX_MNTS    256     //maximum number of mount table entires
#define MAX_MINOS   256     //maximum number of in-memory inodes
#define MAX_OFS     256     //maximum number of open files
#define DEF_IMG     "img/img"


/*struct*/
struct k_t
{
    proc_t*     m_proc_tb[MAX_PROCS];   //array of processes
    mnt_t*      m_mnt_tb[MAX_MNTS];     //array of mount points
    mino_t*     m_mino_tb[MAX_MINOS];   //array of memory inodes
    of_t*       m_of_tb[MAX_OFS];
    proc_t*     m_cproc;                //current process
    mnt_t*      m_cmnt;                 //current mount

};

/*prototypes*/
bool    k_t_init( k_t** );              //initialize
void    k_t_destroy( k_t** );           //shutdown
int     k_t_start( void );              //startup stuff
int     k_t_run( void );                //main loop
bool    k_t_mount_root( char*, mnt_t** );
bool    k_t_start_procs( void );
void    k_t_write_fs_to_disk( void );   //flushes everything

/*blk/mnt related*/
bool    k_t_get_blk( char* dst, int fd, unsigned off );
bool    k_t_put_blk( char* src, int fd, unsigned off );
mnt_t*  k_t_get_mnt_from_fd( int );
bool    k_t_flush_superblock( int fd );
bool    k_t_flush_gd0( int fd );

/*ino/mino related*/
bool    k_t_get_ino( int, u32, inode* );
bool    k_t_get_mino( int fd, u32 inum, mino_t** dst );
bool    k_t_put_ino( int fd, u32 num, inode* ino );
bool    k_t_put_mino( u32 );
bool    k_t_set_mino_as_mounted( int fd, u32 inum, mnt_t* mountpoint );
bool    k_t_set_mino_as_unmounted( int fd, u32 inum, mnt_t* mountpoint );
bool    k_t_flush_clean_minos( void );
bool    k_t_flush_dirty_minos( void );
bool    k_t_flush_all_not_mounted_minos( void );
bool    k_t_flush_all_minos( void );
u32     k_t_add_ino_to_fs( int fd, inode* new_ino );
bool    k_t_del_ino_from_fs( int fd, u32 inum );
bool    k_t_add_dblocks_to_mino( int fd, u32 mino_num, u32 num );
bool    k_t_del_dblocks_from_mino( int fd, u32 mino_num, u32 num );
bool    k_t_truncate_mino( mino_t* mino );

/*bitmap related*/
bool    k_t_set_imap_bit( int fd, u32 bnum, bool bval );
bool    k_t_get_imap_bit( int fd, u32 bnum, bool* retv );
bool    k_t_set_bmap_bit( int fd, u32 bnum, bool bval );
bool    k_t_get_bmap_bit( int fd, u32 bnum, bool* retv );
u32     k_t_get_next_imap_loc( int fd );
u32     k_t_get_next_bmap_loc( int fd );

/*misc*/
```

```
82  void    k_t_p_dir_items( inode*, int, unsigned );
83  int     k_t_find_dir_num( mino_t* mino, char* dir );
84  i32     k_t_find_child_by_name( mino_t* current, char* cname );
85  bool    k_t_add_child( mino_t* parent, i32 inum, char* cname );
86  bool    k_t_remove_child( mino_t* parent, char* cname );
87
88  /*of_tb related*/
89  i32     k_t_find_oft_entry( mino_t* entry );
90  bool    k_t_open_oft_entry( mino_t* entry, E_FILE_MODE mode, i32* fd_loc, char* path );
91  bool    k_t_close_oft_entry( mino_t* entry );
92  i64     k_t_lseek( i32 fd, i32 amount );
93  i32     k_t_read( i32 fd, i32 num, char* dst );
94  i32     k_t_write( i32 fd, i32 num, char* src);
```

2.3.2 Implementation

Listing 4: k_t.h

```
1   /*****************************************************************************
2   file        : k_t.c
3   author      : nc
4   desc        : implementation of the k_t functions
5   date        : 03-25-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <stdio.h>
10  #include <string.h>
11  #include <unistd.h>
12  #include <fcntl.h>
13  #include <ext2fs/ext2_fs.h>
14  #include <sys/types.h>
15  #include <sys/stat.h>
16
17  #include "global_defs.h"
18  #include "k_t.h"
19  #include "mnt_t.h"
20  #include "vsh_t.h"
21  #include "mino_t.h"
22  #include "proc_t.h"
23  #include "string_funcs.h"
24
25
26  extern k_t* kr;
27
28  /*****************************************************************************
29  function    : bool k_t_init( k_t** krn )
30  author      : nc
31  desc        : create the kernel
32  date        : 03-24-13
33   *****************************************************************************/
34  bool k_t_init( k_t** krn )
35  {
36      if( kr )
37          goto fail;
38
39      *krn = malloc( sizeof(**krn) );
40
41      if( !*krn )
42          goto fail;
43
44      memset( *krn, 0, sizeof(**krn) );
45
46      return true;
47
48  fail:
49      printf( "could not create kernel\n" );
50      exit( X_NO_KERNEL );
51      return false;
52  }
53
54
55
56  /*****************************************************************************
57  function    : void k_t_destroy( k_t** krn )
58  author      : nc
59  desc        : destroy the kernel
60  date        : 03-25-13
61   *****************************************************************************/
```

```
62  void k_t_destroy( k_t** krn )
63  {
64      int i = 0;
65
66      k_t_write_fs_to_disk();
67
68      for( i = 0; i < MAX_OFS && (*krn)->m_of_tb[i]; i ++ )
69         of_t_destroy( &(*krn)->m_of_tb[i] );
70
71      for( i = 0; i < MAX_MNTS && (*krn)->m_mnt_tb[i]; i ++ )
72          mnt_t_destroy( &(*krn)->m_mnt_tb[i] );
73
74      for( i = 0; i < MAX_MINOS && (*krn)->m_mino_tb[i]; i ++ )
75          mino_t_destroy( &(*krn)->m_mino_tb[i] );
76
77      for( i = 0; i < MAX_PROCS && (*krn)->m_proc_tb[i]; i ++ )
78          proc_t_destroy( &(*krn)->m_proc_tb[i] );
79
80      if( *krn )
81          free( *krn );
82
83      *krn = NULL;
84
85
86      return;
87  }
88
89
90  /*****************************************************************************
91  function    : int k_t_start( void )
92  author      : nc
93  desc        : initialize the filesystem
94  date        : 03-30-13
95   *****************************************************************************/
96  int k_t_start( void )
97  {
98      if( !k_t_mount_root( NULL, &kr->m_mnt_tb[0] ) )
99          goto no_root;
100
101     if( !k_t_start_procs() )
102         goto no_procs;
103
104     /*add an initial entry to the file table so no one can use it*/
105     if( !of_t_init( &kr->m_of_tb[0] ) )
106         goto no_ft;
107
108     strcpy( kr->m_of_tb[0]->m_name, "root file descriptor" );
109
110     return X_SUCCESS;
111 no_ft:
112     return X_NO_FTABLE;
113 no_root:
114     return X_NO_ROOT;
115 no_procs:
116     return X_NO_PROCS;
117
118 }
119
120
121 /*****************************************************************************
122 function    : int k_t_run( void )
123 author      : nc
124 desc        : main loop for the kernel
125 date        : 03-25-13
126  *****************************************************************************/
127 int k_t_run( void )
128 {
129     vsh_t* sh = NULL;
130
131     vsh_t_init( &sh );
132
133     if( !kr || !sh )
134         goto no_krn;
135
136     printf( "type \"help\" for a list of commands\n" );
137
138     /*replace*/
139     vsh_t_run( sh, kr->m_proc_tb[1] );
140
141     if( sh )
142         vsh_t_destroy( &sh );
```

```
143
144     return X_SUCCESS;
145
146 no_krn:
147     printf( "lost kernel\n" );
148     return X_NO_KERNEL;
149
150 fail_unk:
151     return X_UNKNOWN;
152
153 }
154
155
156
157 /*****************************************************************************
158 function    : bool k_t_get_blk( char* dst, char* src, unsigned num )
159 author      : nc
160 desc        : put a block somewhere
161 date        : 03-30-13
162  *****************************************************************************/
163 bool k_t_get_blk( char* dst, int fd, unsigned num )
164 {
165     if( !dst || !fd )
166         goto fail;
167
168     /*read the block in*/
169     if( ( lseek( fd, (long)( (BLK_SZ)*num ), 0 ) ) == -1 )
170         goto fail;
171
172     read( fd, dst, BLK_SZ );
173
174     return true;
175
176 fail:
177     printf( "error reading block device: source or destination does not exist\n" );
178     return false;
179 }
180
181
182
183 /*****************************************************************************
184 function    : bool k_t_put_blk( char* src, in fd, unsigned num )
185 author      : nc
186 desc        : put a block back in the fs
187 date        : 03-30-13
188  *****************************************************************************/
189 bool k_t_put_blk( char* src, int fd, unsigned num )
190 {
191     if( !src || !fd )
192         goto fail;
193
194     /*write the block out*/
195     if( ( lseek( fd, (long)( (BLK_SZ)*num ), 0 ) ) == -1 )
196         goto fail;
197
198     write( fd, src, BLK_SZ );
199
200     return true;
201
202 fail:
203     printf( "error writing block device: source or destination does not exist\n" );
204     return false;
205 }
206
207
208
209 /*****************************************************************************
210 function    : mnt_t* k_t_get_mnt_from_fd( int fd )
211 author      : nc
212 desc        : get a mount table entry from its file descriptor
213 date        : 03-30-13
214  *****************************************************************************/
215 mnt_t* k_t_get_mnt_from_fd( int fd )
216 {
217     int i = 0;
218     for( i = 0; i < MAX_MNTS && kr->m_mnt_tb[i]; i ++ )
219     {
220         if( kr->m_mnt_tb[i]->m_fd == fd )
221             return kr->m_mnt_tb[i];
222     }
223
```

```
224     return NULL;
225 }
226
227
228
229 /*****************************************************************************
230 function    : bool k_t_mount_root( char* path )
231 author      : nc
232 desc        : mount all the root devices
233 date        : 03-30-13
234  *****************************************************************************/
235 bool k_t_mount_root( char* path, mnt_t** dst )
236 {
237     char loc[MAX_PATH]  = { '\0' };
238
239     /*use the default path for disk image if one isn't specified*/
240     if( !path )
241     {
242         strcpy( loc, DEF_IMG );
243     }
244     else
245     {
246         if( strlen( path ) - 1 > MAX_PATH )
247             goto fail;
248
249         strcpy( loc, path );
250     }
251
252     /*create the first mount device and the in-memory inode*/
253     if( !*dst )
254         mnt_t_init( dst );
255
256     if( !*dst )
257         goto fail;
258
259     mnt_t_create( *dst, loc, "/", ROOT_INO_NUM );
260
261     printf( "mounted root device successfully\n" );
262
263     return true;
264
265 fail:
266
267     if( *dst )
268         mnt_t_destroy( dst );
269
270     printf( "failed to mount root device\n" );
271     return false;
272 }
273
274
275 /*****************************************************************************
276 function    : bool k_t_start_procs( void )
277 author      : nc
278 desc        : start the initial processes
279 date        : 04-02-13
280  *****************************************************************************/
281 bool k_t_start_procs( void )
282 {
283     if( !kr )
284         goto fail;
285
286     /*make two processes*/
287     if( !proc_t_init( &kr->m_proc_tb[0] ) )
288         goto fail;
289
290     if( !proc_t_init( &kr->m_proc_tb[1] ) )
291         goto fail;
292
293     if( !proc_t_make(
294                 kr->m_proc_tb[0],
295                 U_SUPER,
296                 0,
297                 0,
298                 0,
299                 S_RUNNING,
300                 NULL,
301                 kr->m_mino_tb[0] ) )
302         goto fail;
303
304     if( !proc_t_make(
```

```
305                 kr->m_proc_tb[1],
306                 U_REG,
307                 1,
308                 0,
309                 0,
310                 S_RUNNING,
311                 kr->m_proc_tb[0],
312                 kr->m_mino_tb[0] ) )
313         goto fail;
314
315     printf( "created root processes sucessfully\n" );
316     return true;
317
318 fail:
319     printf( "unable to start processes\n" );
320     proc_t_destroy( &kr->m_proc_tb[0] );
321     proc_t_destroy( &kr->m_proc_tb[1] );
322
323     return false;
324 }
325
326
327
328 /*****************************************************************************
329 function    : bool k_t_wite_fs_to_disk( void )
330 author      : nc
331 desc        : writes everything open in memory out to disk
332 date        : 04-10-13
333  *****************************************************************************/
334 void k_t_write_fs_to_disk( void )
335 {
336     int i = 0;
337
338     /*write the basic stuff out*/
339     for( i = 0; i < MAX_MNTS; i ++ )
340     {
341         if( kr->m_mnt_tb[i] )
342         {
343             k_t_flush_superblock( kr->m_mnt_tb[i]->m_fd );
344             k_t_flush_gd0( kr->m_mnt_tb[i]->m_fd );
345         }
346     }
347
348     /*write everything else out*/
349     k_t_flush_all_minos();
350
351     return;
352 }
353
354
355
356 /*****************************************************************************
357 function    : bool k_t_flush_superblock( int fd )
358 author      : nc
359 desc        : write the superblock for a specified mount back to disk
360 date        : 04-10-13
361  *****************************************************************************/
362 bool k_t_flush_superblock( int fd )
363 {
364     mnt_t* mt       = NULL;
365     char sb[BLK_SZ] = { '\0' };
366
367     if( !fd || sizeof(sb) != BLK_SZ )
368         goto fail;
369
370     /*get the mount point*/
371     mt = k_t_get_mnt_from_fd( fd );
372
373     if( !mt )
374         goto fail;
375
376     /*write it*/
377     memcpy( sb, &mt->m_sb, BLK_SZ );
378     k_t_put_blk( (char*)&mt->m_sb, fd, SB_BLK );
379
380     return true;
381
382 fail:
383     printf( "uanble to write superblock for fd%i\n", fd );
384     return false;
385 }
```

```
386
387
388
389 /*****************************************************************************
390 function    : bool k_t_flush_gd0( int fd )
391 author      : nc
392 desc        : write the group descriptor back to disk
393 date        : 04-10-13
394  *****************************************************************************/
395 bool k_t_flush_gd0( int fd )
396 {
397     mnt_t* mt           = NULL;
398     char gd0[BLK_SZ]    = { '\0' };
399
400     if( !fd )
401         goto fail;
402
403     /*get the mount point*/
404     mt = k_t_get_mnt_from_fd( fd );
405
406     if( !mt )
407         goto fail;
408
409     memcpy( gd0, &mt->m_gd0, sizeof(gd) );
410
411     /*gd0 is a different size so we have to use a different algo here*/
412     if( ( lseek( fd, (long)( (BLK_SZ)*GD0_BLK ), 0 ) ) == -1 )
413         goto fail;
414
415     write( fd, gd0, sizeof(gd) );
416
417     return true;
418
419 fail:
420     printf( "uanble to write gd0 for fd%i\n", fd );
421     return false;
422 }
423
424
425
426
427 /*****************************************************************************
428 function    : bool k_t_get_ino( int fd, u32 num, inode* ino )
429 author      : nc
430 desc        : read an inode from a file
431 date        : 03-31-13
432  *****************************************************************************/
433 bool k_t_get_ino( int fd, u32 num, inode* ino )
434 {
435     char b[BLK_SZ]  = { '\0' };
436     mnt_t* mnt      = NULL;
437     int blk         = 0;
438     int off         = 0;
439
440     if( !num || !fd || !ino )
441         goto fail;
442
443     /*get a mnt_t* so we can find out where the inode table starts*/
444     mnt = k_t_get_mnt_from_fd( fd );
445
446     if( !mnt )
447         goto fail;
448
449     /*reset inode*/
450     memset( ino, 0, sizeof(*ino) );
451
452     num -= 1;
453
454     /*get the correct offset*/
455     blk = ( num / INO_PER_BLK );
456     off = ( num % INO_PER_BLK );
457     blk += mnt->m_ino_start;
458
459     /*copy the inode info over*/
460     k_t_get_blk( b, fd, blk );
461     memcpy( ino, b+( off*INO_SIZE ), INO_SIZE );
462
463     return true;
464
465 fail:
466     printf( "unable to find inode %i\n", num );
```

```
467      return false;
468  }
469
470
471
472
473  /*****************************************************************************
474  function    : bool k_t_get_mino( int fd, u32 num, mino_t** dst )
475  author      : nc
476  desc        : get an in-memory inode
477  date        : 03-31-13
478   *****************************************************************************/
479  bool k_t_get_mino( int fd, u32 num, mino_t** dst )
480  {
481      int i               = 0;
482      inode ino           = { 0 };
483      mino_t* new_mino    = NULL;
484
485      if( !fd )
486          goto fail;
487
488      /*see if the mino is already open*/
489      for( i = 0; i < MAX_MINOS && kr->m_mino_tb[i]; i ++ )
490      {
491          if( kr->m_mino_tb[i]->m_ino_num == num )
492          {
493              *dst = kr->m_mino_tb[i];
494              kr->m_mino_tb[i]->m_refc++;
495              goto success;
496          }
497      }
498
499      if( i >= MAX_MINOS )
500          goto fail;
501
502      /*if not, we need to open it*/
503      if( !k_t_get_ino( fd, num, &ino ) )
504          goto fail;
505
506      /*now add it to the mino table*/
507      mino_t_init( &new_mino );
508
509      /*make it*/
510      if( !mino_t_make( new_mino,
511                  &ino,
512                  fd,
513                  num,
514                  1,
515                  false,
516                  false,
517                  NULL ) )
518          goto fail;
519
520      kr->m_mino_tb[i] = new_mino;
521
522      /*point the destination to it*/
523      *dst = new_mino;
524
525  success:
526      return true;
527
528  fail:
529      mino_t_destroy( &new_mino );
530      printf( "unable to get in-memory inode %i\n", num );
531      return false;
532  }
533
534
535
536  /*****************************************************************************
537  function    : bool k_t_put_ino( int fd, u32 num, inode* ino )
538  author      : nc
539  desc        : write a memory to the file
540  date        : 03-31-13
541   *****************************************************************************/
542  bool k_t_put_ino( int fd, u32 num, inode* ino )
543  {
544      char b[BLK_SZ]  = { '\0' };
545      mnt_t* mnt      = NULL;
546      int blk         = 0;
547      int off         = 0;
```

```
548
549     if( !num || !fd || !ino )
550         goto fail;
551
552     /*get a mnt_t* so we can find out where the inode table starts*/
553     mnt = k_t_get_mnt_from_fd( fd );
554
555     if( !mnt )
556         goto fail;
557
558     num -= 1;
559
560     /*get the correct offset*/
561     blk = ( num / INO_PER_BLK );
562     off = ( num % INO_PER_BLK );
563     blk += mnt->m_ino_start;
564
565     /*copy the inode info over*/
566     k_t_get_blk( b, fd, blk );
567     memcpy( b+( off*INO_SIZE ), ino, INO_SIZE );
568     k_t_put_blk( b, fd, blk );
569
570     return true;
571
572 fail:
573     printf( "unable to find inode %i\n", num );
574     return false;
575 }
576
577
578
579 /*****************************************************************************
580 function    : bool k_t_put_mino( mino_t** src )
581 author      : nc
582 desc        : write out an in-memory inode
583 date        : 03-31-13
584  *****************************************************************************/
585 bool k_t_put_mino( u32 num )
586 {
587     int i = 0;
588
589     if( !kr )
590         goto fail;
591
592     /*see if the mino is already open*/
593     for( i = 0; i < MAX_MINOS; i ++ )
594     {
595         if( kr->m_mino_tb[i] && kr->m_mino_tb[i]->m_ino_num == num )
596         {
597             kr->m_mino_tb[i]->m_refc--;
598             break;
599         }
600     }
601
602     if( i >= MAX_MINOS )
603         goto fail;
604
605     /*check for zero ref count*/
606     if( kr->m_mino_tb[i] && kr->m_mino_tb[i]->m_refc <= 0 )
607     {
608         if( !k_t_put_ino( kr->m_mino_tb[i]->m_fd, num, &kr->m_mino_tb[i]->m_ino ) )
609             goto fail;
610
611         mino_t_destroy( &kr->m_mino_tb[i] );
612     }
613
614 success:
615     return true;
616
617 fail:
618     printf( "unable to put in-memory inode %i\n", num );
619     return false;
620 }
621
622
623
624 /*****************************************************************************
625 function    : bool k_t_set_mino_as_mounted( int fd, u32 inum, mnt_t* mountpoint )
626 author      : nc
627 desc        : flag a mino as a mount point
628 date        : 04-07-13
```

```
629  ***************************************************************************/
630 bool k_t_set_mino_as_mounted( int fd, u32 inum, mnt_t* mountpoint )
631 {
632     mino_t* tmp = NULL;
633 
634     if( !fd || !inum || !mountpoint )
635         goto fail;
636 
637     k_t_get_mino( fd, inum, &tmp );
638 
639     if( !tmp )
640         goto fail;
641 
642     tmp->m_mounted = true;
643     tmp->m_mountp = mountpoint;
644 
645     k_t_put_mino( inum );
646 
647     return true;
648 
649 fail:
650     printf( "unable to mount inode %i\n", inum );
651     return false;
652 }
653 
654 
655 
656 /***************************************************************************
657 function    : bool k_t_set_mino_as_unmounted( int fd, u32 inum, mnt_t* mountpoint )
658 author      : nc
659 desc        : flag a mino as a mount point
660 date        : 04-07-13
661  ***************************************************************************/
662 bool k_t_set_mino_as_unmounted( int fd, u32 inum, mnt_t* mountpoint )
663 {
664     mino_t* tmp = NULL;
665 
666     if( !fd || !inum || !mountpoint )
667         goto fail;
668 
669     k_t_get_mino( fd, inum, &tmp );
670 
671     if( !tmp )
672         goto fail;
673 
674     tmp->m_mounted = false;
675     tmp->m_mountp = NULL;
676 
677     k_t_put_mino( inum );
678 
679     return true;
680 
681 fail:
682     printf( "unable to unmount inode %i\n", inum );
683     return false;
684 }
685 
686 
687 
688 /***************************************************************************
689 function    : bool k_t_flush_clean_minos( void )
690 author      : nc
691 desc        : 04-10-13
692 date        : write all clean minos to disk
693  ***************************************************************************/
694 bool k_t_flush_clean_minos( void )
695 {
696     int i = 0;
697 
698     if( !kr || !kr->m_mino_tb )
699         goto fail;
700 
701     for( i = 0; i < MAX_MINOS; i++ )
702     {
703         if( kr->m_mino_tb[i] &&
704                 kr->m_mino_tb[i]->m_dirty == false &&
705                 kr->m_mino_tb[i]->m_mounted == false )
706         {
707             //printf( "found mino at location %i\n", i );
708             kr->m_mino_tb[i]->m_refc = 1;
709             k_t_put_mino( kr->m_mino_tb[i]->m_ino_num );
```

```
710         }
711     }
712
713     return true;
714
715 fail:
716     printf( "unable to flush clean in-memory inodes\n" );
717     return false;
718 }
719
720
721 /*****************************************************************************
722 function    : bool k_t_flush_dirty_minos( void )
723 author      : nc
724 desc        : 04-10-13
725 date        : write all dirty minos to disk
726  *****************************************************************************/
727 bool k_t_flush_dirty_minos( void )
728 {
729     int i = 0;
730
731     if( !kr || !kr->m_mino_tb )
732         goto fail;
733
734     for( i = 0; i < MAX_MINOS; i++ )
735     {
736         if( kr->m_mino_tb[i] &&
737                 kr->m_mino_tb[i]->m_dirty == true &&
738                 kr->m_mino_tb[i]->m_mounted == false )
739         {
740             //printf( "found mino at location %i\n", i );
741             kr->m_mino_tb[i]->m_refc = 1;
742             k_t_put_mino( kr->m_mino_tb[i]->m_ino_num );
743         }
744     }
745
746     return true;
747
748 fail:
749     printf( "unable to flush dirty in-memory inodes\n" );
750     return false;
751 }
752
753
754
755 /*****************************************************************************
756 function    : bool k_t_flush_all_not_mounted_minos( void )
757 author      : nc
758 desc        : 04-10-13
759 date        : write all non-mounted minos
760  *****************************************************************************/
761 bool k_t_flush_all_not_mounted_minos( void )
762 {
763     int i = 0;
764
765     if( !kr || !kr->m_mino_tb )
766         goto fail;
767
768     for( i = 0; i < MAX_MINOS; i++ )
769     {
770         if( kr->m_mino_tb[i] && kr->m_mino_tb[i]->m_mounted == false )
771         {
772             //printf( "found mino at location %i\n", i );
773             kr->m_mino_tb[i]->m_refc = 1;
774             k_t_put_mino( kr->m_mino_tb[i]->m_ino_num );
775         }
776     }
777
778     return true;
779
780 fail:
781     printf( "unable to flush all in-memory inodes\n" );
782     return false;
783 }
784
785
786
787 /*****************************************************************************
788 function    : bool k_t_flush_all_minos( void )
789 author      : nc
790 desc        : 04-10-13
```

```
date        : write all minos
 ******************************************************************************/
bool k_t_flush_all_minos( void )
{
    int i = 0;

    if( !kr || !kr->m_mino_tb )
        goto fail;

    for( i = 0; i < MAX_MINOS; i++ )
    {
        if( kr->m_mino_tb[i] )
        {
            //printf( "found mino at location %i\n", i );
            kr->m_mino_tb[i]->m_refc = 1;
            k_t_put_mino( kr->m_mino_tb[i]->m_ino_num );
        }
    }

    return true;

fail:
    printf( "unable to flush all in-memory inodes\n" );
    return false;
}



/*****************************************************************************
function    : u32 k_t_add_ino_to_fs( inode* new_ino )
author      : nc
desc        : 04-11-13
date        : add an inode to the filesystem. does not add to mino table
 ******************************************************************************/
u32 k_t_add_ino_to_fs( int fd, inode* new_ino )
{
    u32 loc     = 0;
    mnt_t* mt   = NULL;

    if( !fd || !new_ino )
        goto fail;

    /*get next avaliable location*/
    mt = k_t_get_mnt_from_fd( fd );
    loc =  k_t_get_next_imap_loc( fd );

    if( !loc || !mt )
        goto fail;

    if( mt->m_sb.s_free_inodes_count == 0 )
        goto fail;

    /*lock the fs*/
    mnt_t_lock( mt );

    /*add it to the map*/
    if( !k_t_put_ino( fd, loc, new_ino ) )
        goto fail;

    k_t_set_imap_bit( fd, loc, 1 );

    mt->m_sb.s_free_inodes_count--;

    mnt_t_unlock( mt );
    return loc;

fail:
    mnt_t_unlock( mt );
    printf( "unable to add inode to filesystem\n" );
    return 0;
}



/*****************************************************************************
function    : bool k_t_del_ino_from_fs( int fd, u32 inum )
author      : nc
desc        : delete an inode from the fs
date        : 04-11-13
 ******************************************************************************/
bool k_t_del_ino_from_fs( int fd, u32 inum )
```

```
872 {
873     char in[16] = { 0 };
874     mnt_t* mt   = NULL;
875     int i       = 0;
876 
877     if( !fd || !inum )
878         goto fail;
879 
880     /*check and see if they are trying to delete the root inode*/
881     if( inum == ROOT_INO_NUM )
882     {
883         printf( "warning: attempting to remove root inode.  continue (y/n)?\n" );
884         scanf( "%s", in );
885 
886         if( in[0] == 'n' )
887             goto fail;
888     }
889 
890     mt = k_t_get_mnt_from_fd( fd );
891 
892     if( !mt )
893         goto fail;
894 
895     /*check the mino table and remove it if it is open*/
896     for( i = 0; i < MAX_MINOS; i++ )
897     {
898         if( kr->m_mino_tb[i] && kr->m_mino_tb[i]->m_ino_num == inum )
899         {
900             kr->m_mino_tb[i]->m_refc = 1;
901             k_t_put_mino( kr->m_mino_tb[i]->m_ino_num );
902         }
903     }
904 
905     /*lock the fs*/
906     mnt_t_lock( mt );
907 
908     /*delete it*/
909     k_t_set_imap_bit( fd, inum, 0 );
910 
911     mt->m_sb.s_free_inodes_count++;
912 
913     /*unlock*/
914     mnt_t_unlock( mt );
915 
916     return true;
917 
918 fail:
919     printf( "unable to remove inode %i from fd%i\n", inum, fd );
920     mnt_t_unlock( mt );
921     return false;
922 }
923 
924 
925 /*****************************************************************************
926 function    : bool k_t_add_dblocks_to_mino( int fd, u32 mino_num, u16 num )
927 author      : nc
928 desc        :
929 date        :
930  *****************************************************************************/
931 bool k_t_add_dblocks_to_mino( int fd, u32 mino_num, u32 num )
932 {
933     int i           = 0;
934     int j           = 0;
935     int loc         = 0;
936     unsigned* off   = NULL;
937     unsigned* off2  = NULL;
938     int blocks_used = 0;
939     mino_t* local   = NULL;
940     char b[BLK_SZ]  = { '\0' };
941     char b2[BLK_SZ] = { '\0' };
942     mnt_t* mt       = NULL;
943 
944     if( !fd || !mino_num || !num )
945         goto fail;
946 
947     k_t_get_mino( fd, mino_num, &local );
948     mt = k_t_get_mnt_from_fd( fd );
949     j = num;
950     if( !local || !mt )
951         goto fail;
952 
```

```
953     mnt_t_lock( mt );
954
955     /*count the number of data blocks already in use*/
956     for( i = 0; i < THREE_INDIR; i++ )
957     {
958         memset(b, 0, BLK_SZ);
959         off = (unsigned*)b;
960         /*use <= because we need to count the indirect block as ours*/
961         if( local->m_ino.i_block[i] && i < ONE_INDIR - 1 )
962         {
963             blocks_used++;
964             printf( "%i ", local->m_ino.i_block[i] );
965         }
966
967         /* first set of indirect blocks*/
968         if( i == ONE_INDIR - 1 && local->m_ino.i_block[i] )
969         {
970             k_t_get_blk( b, fd, local->m_ino.i_block[i] );
971             off = (unsigned*)b;
972
973             if( !off )
974                 goto fail;
975
976             while(*off < DBLOCK && *off > 0 )
977             {
978                 //printf( "%i ", *off );
979                 off++;
980                 blocks_used++;
981             }
982         }
983
984         /*double indirect*/
985         if( i == TWO_INDIR - 1 && local->m_ino.i_block[i] )
986         {
987             /*get the block where the indirect block nums are stored*/
988             k_t_get_blk( b, fd, local->m_ino.i_block[i] );
989             off = (unsigned*)b;
990
991             /*pointer gymnastics*/
992             while( *off < DBLOCK && *off > 0 )
993             {
994                 /*read all the single indirect blocks in the double-indirect loc*/
995                 k_t_get_blk( b2, fd, *off );
996                 off2 = (unsigned*)b2;
997
998                 /*print what they point at*/
999                 while( *off2 < DBLOCK && *off2 > 0 )
1000                {
1001                    printf( "%i ", *off2 );
1002                    off2++;
1003                    blocks_used++;
1004                }
1005
1006                printf( "%i ", *off );
1007                off++;
1008                blocks_used++;
1009            }
1010        }
1011    }
1012
1013    /*make sure there is room*/
1014    if( blocks_used + num > MAX_DBLOCKS )
1015        goto fail;
1016
1017    printf( "\nadding direct blocks\n" );
1018
1019    /*start walking*/
1020    for( i = 0 ; i < THREE_INDIR && num != 0; i++ )
1021    {
1022        if( !local->m_ino.i_block[i] && i < ONE_INDIR - 1 )
1023        {
1024            /*get the next open data block*/
1025            loc = k_t_get_next_bmap_loc( fd );
1026
1027            /*tell the bmap it is ours*/
1028            if( !k_t_set_bmap_bit( fd, loc, 1 ) )
1029                goto fail;
1030
1031            /*add the new location to the inode*/
1032            local->m_ino.i_block[i] = loc;
1033
```

```
1034            printf( "added dblock at position %i to ino %i\n", loc, local->m_ino_num );
1035            num--;
1036        }
1037
1038        /* first set of indirect blocks*/
1039        if( i == ONE_INDIR - 1 )
1040        {
1041            memset( b, 0, BLK_SZ );
1042            off = (unsigned*)b;
1043
1044            if( !off )
1045                goto fail;
1046
1047            /*make an indirect data block*/
1048            if(local->m_ino.i_block[ONE_INDIR -1] == 0)
1049            {
1050                /*get the next open data block - still comes from the bitmap*/
1051                loc = k_t_get_next_bmap_loc( fd );
1052
1053                /*tell the bmap it is ours*/
1054                if( !k_t_set_bmap_bit( fd, loc, 1 ) )
1055                    goto fail;
1056
1057                local->m_ino.i_block[i] = loc;
1058
1059            }
1060            k_t_get_blk(b, fd, local->m_ino.i_block[ONE_INDIR-1]);
1061
1062            off = (unsigned*)b;
1063
1064            /*traverse used indirect*/
1065            while(*off < DBLOCK && *off > 0)
1066                off++;
1067
1068            while( num != 0 && ( off - (unsigned*)b ) < BLK_SZ/sizeof(unsigned) )
1069            {
1070
1071                /*get the next open data block - still comes from the bitmap*/
1072                loc = k_t_get_next_bmap_loc( fd );
1073
1074                /*tell the bmap it is ours*/
1075                if( !k_t_set_bmap_bit( fd, loc, 1 ) )
1076                    goto fail;
1077
1078                /*add the new location to the inode*/
1079                *off = loc;
1080                off++;
1081
1082                printf( "added id dblock at position %i to ino %i\n", loc, local->m_ino_num );
1083
1084                /*we are now adding inode positions to the data block instead of the ino*/
1085                num--;
1086            }
1087
1088            /*write it out*/
1089            if( !k_t_put_blk( b, fd, local->m_ino.i_block[ONE_INDIR - 1] ) )
1090                goto fail;
1091        }
1092
1093        /*double indirect*/
1094        if( i == TWO_INDIR - 1 )
1095        {
1096            memset( b, 0, BLK_SZ );
1097            off = (unsigned*)b;
1098
1099            if(local->m_ino.i_block[TWO_INDIR - 1] == 0)
1100            {
1101                /*get the next open data block*/
1102                loc = k_t_get_next_bmap_loc( fd );
1103
1104                /*tell the bmap it is ours*/
1105                if( !k_t_set_bmap_bit( fd, loc, 1 ) )
1106                    goto fail;
1107
1108                /*add the new location to the inode*/
1109                local->m_ino.i_block[TWO_INDIR - 1] = loc;
1110            }
1111
1112            if( !off )
1113                goto fail;
1114
```

```
1115             /*get double indirect block*/
1116             k_t_get_blk(b, fd, local->m_ino.i_block[TWO_INDIR - 1]);
1117
1118             off = (unsigned*)b;
1119
1120             while( num != 0 && ( off - (unsigned*)b ) < BLK_SZ/sizeof(unsigned) )
1121             {
1122                 if(*off > DBLOCK && *off < 0)
1123                 {
1124                     /*get the next open data block - still comes from the bitmap*/
1125                     loc = k_t_get_next_bmap_loc( fd );
1126
1127                     /*tell the bmap it is ours*/
1128                     if( !k_t_set_bmap_bit( fd, loc, 1 ) )
1129                         goto fail;
1130
1131                     /*add indirect position to data block*/
1132                     *off = loc;
1133
1134                     printf( "added 2id, id dblock at position %i to ino %i\n", loc, local->m_ino_num );
1135                 }
1136
1137                 /*read all the single indirect blocks in the double-indirect loc*/
1138                 memset( b2, 0, BLK_SZ );
1139                 off2 = (unsigned*)b2;
1140
1141                 k_t_get_blk(b2, fd, *off);
1142
1143                 /*traverse used double indirect*/
1144                 off2 = (unsigned*)b2;
1145
1146                 /*find open indirect data block*/
1147                 while(*off2 < DBLOCK && *off2 > 0)
1148                     off2++;
1149
1150                 while( num != 0 && ( off2 - (unsigned*)b2 ) < BLK_SZ/sizeof(unsigned) )
1151                 {
1152                     /*get the next open data block - still comes from the bitmap*/
1153                     loc = k_t_get_next_bmap_loc( fd );
1154
1155                     /*tell the bmap it is ours*/
1156                     if( !k_t_set_bmap_bit( fd, loc, 1 ) )
1157                         goto fail;
1158
1159                     /*add the new location to the inode*/
1160                     *off2 = loc;
1161
1162                     printf( "added 2id dblock at position %i to ino %i\n", loc, local->m_ino_num );
1163
1164                     off2++;
1165                     num--;
1166                 }
1167                 k_t_put_blk( b2, fd, *off );
1168                 off++;
1169             }
1170             /*write it out*/
1171             if( !k_t_put_blk( b, fd, local->m_ino.i_block[TWO_INDIR - 1] ) )
1172                 goto fail;
1173         }
1174     }
1175
1176     /*update i_block (indexs 512 bytes so * 2 )*/
1177     local->m_ino.i_blocks = ( blocks_used + j ) * 2;
1178
1179
1180     mnt_t_unlock( mt );
1181     return true;
1182 fail:
1183     if( mt )
1184         mnt_t_unlock( mt );
1185     printf( "unable to add data blocks to mino %i\n", num );
1186     return false;
1187 }
1188
1189
1190
1191 /*****************************************************************************
1192 function    : bool k_t_del_dblocks_from_mino( int fd, u32 mino_num, u32 num )
1193 author      : nc
1194 desc        : delete data blocks from a mino
1195 date        :
```

```
1196  ******************************************************************************/
1197 bool k_t_del_dblocks_from_mino( int fd, u32 mino_num, u32 num )
1198 {
1199     int i               = 0;
1200     int start           = 0;
1201     int cblock          = 0;
1202     unsigned* off       = NULL;
1203     unsigned* off2      = NULL;
1204     int blocks_used     = 0;
1205     mino_t* local       = NULL;
1206     char b[BLK_SZ]      = { '\0' };
1207     char b2[BLK_SZ]     = { '\0' };
1208     mnt_t* mt           = NULL;
1209
1210     if( !fd || !mino_num || !num )
1211         goto fail;
1212
1213     k_t_get_mino( fd, mino_num, &local );
1214     mt = k_t_get_mnt_from_fd( fd );
1215
1216     if( !local || !mt )
1217         goto fail;
1218
1219     /*lock the fs while we are writing to it*/
1220     mnt_t_lock( mt );
1221
1222     /*count the number of data blocks in use*/
1223     for( i = 0; i < THREE_INDIR; i++ )
1224     {
1225         memset( b, 0, BLK_SZ );
1226         off = (unsigned*)b;
1227         /*use <= because we need to count the indirect block as ours*/
1228         if( local->m_ino.i_block[i] && i < ONE_INDIR - 1 )
1229         {
1230             blocks_used++;
1231             //printf( "%i ", local->m_ino.i_block[i] );
1232         }
1233
1234         /* first set of indirect blocks*/
1235         if( i == ONE_INDIR - 1 && local->m_ino.i_block[i] != 0 )
1236         {
1237             k_t_get_blk( b, fd, local->m_ino.i_block[i] );
1238             off = (unsigned*)b;
1239
1240             if( !off )
1241                 goto fail;
1242
1243             while(*off < DBLOCK && *off > 0 )
1244             {
1245                 //printf( "%i ", *off );
1246                 off++;
1247                 blocks_used++;
1248             }
1249         }
1250
1251         /*double indirect*/
1252         if( i == TWO_INDIR - 1 && local->m_ino.i_block[i] )
1253         {
1254             /*get the block where the indirect block nums are stored*/
1255             k_t_get_blk( b, fd, local->m_ino.i_block[i] );
1256             off = (unsigned*)b;
1257
1258             /*pointer gymnastics*/
1259             while( *off < DBLOCK && *off > 0 )
1260             {
1261                 /*read all the single indirect blocks in the double-indirect loc*/
1262                 k_t_get_blk( b2, fd, *off );
1263                 off2 = (unsigned*)b2;
1264
1265                 /*print what they point at*/
1266                 while( *off2 < DBLOCK && *off2 > 0 )
1267                 {
1268                     printf( "%i ", *off2 );
1269                     off2++;
1270                     blocks_used++;
1271                 }
1272
1273                 printf( "%i ", *off );
1274                 off++;
1275                 blocks_used++;
1276             }
```

```
1277            }
1278        }
1279
1280        /*make sure it is a valid request*/
1281        if( blocks_used - num < 0 )
1282            goto fail;
1283
1284        /*we have to do this one different*/
1285        start = blocks_used - num;
1286
1287
1288        printf( "\n" );
1289
1290        /*sigh*/
1291        for( i = 0 ; i < THREE_INDIR && num != 0; i++ )
1292        {
1293            if( local->m_ino.i_block[i] && i < ONE_INDIR - 1 )
1294            {
1295                /*see if we are at the starting position*/
1296                if( cblock >= start )
1297                {
1298                    printf( "removing data block %i from inode\n", local->m_ino.i_block[i] );
1299
1300                    /*update the bitmap*/
1301                    if( !local->m_ino.i_block[i] || !k_t_set_bmap_bit( fd, local->m_ino.i_block[i], 0 ) )
1302                        goto fail;
1303
1304                    /*zero the block*/
1305                    local->m_ino.i_block[i] = 0;
1306                    num--;
1307                }
1308
1309                /*update block count*/
1310                cblock++;
1311            }
1312
1313            /* first set of indirect blocks*/
1314            if( i == ONE_INDIR - 1 && local->m_ino.i_block[i] && num != 0 )
1315            {
1316                k_t_get_blk( b, fd, local->m_ino.i_block[i] );
1317                off = (unsigned*)b;
1318                cblock++;
1319
1320                if( !off )
1321                    goto fail;
1322
1323                if( cblock >= start )
1324                {
1325                    /*free the parent*/
1326                    printf( "removing parent dblock %i\n", local->m_ino.i_block[ONE_INDIR - 1] );
1327                    k_t_set_bmap_bit( fd, local->m_ino.i_block[ONE_INDIR - 1], 0 );
1328                    local->m_ino.i_block[ONE_INDIR - 1] = 0;
1329                    num--;
1330                }
1331
1332                while( *off < DBLOCK && *off > 0)
1333                {
1334                    /*see if we are at the starting position*/
1335                    if( cblock >= start )
1336                    {
1337                        printf( "removing id data block %i from inode\n", *off );
1338
1339                        /*update the bitmap*/
1340                        if( !k_t_set_bmap_bit( fd, *off, 0 ) )
1341                            goto fail;
1342
1343                        /*zero the block*/
1344                        *off = 0;
1345                        num--;
1346                    }
1347
1348                    /*write it out*/
1349                    if( !k_t_put_blk( b, fd, local->m_ino.i_block[ONE_INDIR - 1] ) )
1350                        goto fail;
1351
1352                    off++;
1353                    cblock++;
1354                }
1355            }
1356
1357            /*double indirect*/
```

```
1358          if( i == TWO_INDIR - 1 && local->m_ino.i_block[i] && num != 0 )
1359          {
1360              k_t_get_blk( b, fd, local->m_ino.i_block[i] );
1361              off = (unsigned*)b;
1362              cblock++;
1363
1364              if( !off )
1365                  goto fail;
1366
1367              if( cblock >= start )
1368              {
1369                  /*free the parent*/
1370                  printf( "removing parent dblock %i\n", local->m_ino.i_block[TWO_INDIR - 1] );
1371                  k_t_set_bmap_bit( fd, local->m_ino.i_block[TWO_INDIR - 1], 0 );
1372                  local->m_ino.i_block[TWO_INDIR - 1] = 0;
1373                  num--;
1374              }
1375
1376              while( *off < DBLOCK && *off > 0 )
1377              {
1378                  k_t_get_blk( b2, fd, *off );
1379                  off2 = (unsigned*)b2;
1380                  cblock++;
1381
1382                  if( !off2 )
1383                      goto fail;
1384
1385                  if( cblock >= start )
1386                  {
1387                      /*free the parent*/
1388                      printf( "removing parent id dblock %i\n", *off );
1389                      k_t_set_bmap_bit( fd, *off, 0 );
1390                      *off = 0;
1391                      num--;
1392                  }
1393
1394                  while( *off2 < DBLOCK && *off2 > 0)
1395                  {
1396                      /*see if we are at the starting position*/
1397                      if( cblock >= start )
1398                      {
1399                          printf( "removing 2id data block %i from inode\n", *off2 );
1400
1401                          /*update the bitmap*/
1402                          if( !k_t_set_bmap_bit( fd, *off2, 0 ) )
1403                              goto fail;
1404
1405                          /*zero the block*/
1406                          *off2 = 0;
1407                          num--;
1408                      }
1409
1410                      /*write it out*/
1411                      if( !k_t_put_blk( b, fd, *off ) )
1412                          goto fail;
1413
1414                      off2++;
1415                      cblock++;
1416                  }
1417
1418                  /*write it out*/
1419                  if( !k_t_put_blk( b, fd, local->m_ino.i_block[TWO_INDIR - 1] ) )
1420                      goto fail;
1421
1422                  off++;
1423                  cblock++;
1424              }
1425          }
1426      }
1427
1428      /*update i_block (indexs 512 bytes so * 2 )*/
1429      local->m_ino.i_blocks = ( start ) * 2;
1430
1431      mnt_t_unlock( mt );
1432      return true;
1433  fail:
1434      if( mt )
1435          mnt_t_unlock( mt );
1436      printf( "unable to delete blocks from mino %i\n", mino_num );
1437      return false;
1438  }
```

```
1439
1440
1441 /*****************************************************************************
1442 function    : bool k_t_truncate_mino( mino_t* mino )
1443 author      : nc
1444 desc        : remove all data blocks from a mino and set size to zero
1445 date        :
1446  *****************************************************************************/
1447 bool k_t_truncate_mino( mino_t* mino )
1448 {
1449     int i               = 0;
1450     int blocks_used     = 0;
1451     char b[BLK_SZ]      = { '\0' };
1452     char b2[BLK_SZ]     = { '\0' };
1453     unsigned* off       = NULL;
1454     unsigned* off2      = NULL;
1455     mino_t* local       = NULL;
1456
1457     if( !mino )
1458     {
1459         printf( "error: no m_ino*\n" );
1460         goto fail;
1461     }
1462
1463     /*hax*/
1464     local = mino;
1465
1466     if( !local )
1467     {
1468         printf( "error: no mount point\n" );
1469         goto fail;
1470     }
1471
1472     /*find out the max block file location*/
1473     for( i = 0; i < THREE_INDIR; i++ )
1474     {
1475         memset( b, 0, BLK_SZ );
1476         off = (unsigned*)b;
1477         if( local->m_ino.i_block[i] && i < ONE_INDIR - 1 )
1478         {
1479             blocks_used++;
1480             printf( "%i ", local->m_ino.i_block[i] );
1481         }
1482
1483
1484         /* first set of indirect blocks*/
1485         if( i == ONE_INDIR - 1 && local->m_ino.i_block[i] )
1486         {
1487             k_t_get_blk( b, local->m_fd, local->m_ino.i_block[i] );
1488             off = (unsigned*)b;
1489
1490             if( !off )
1491                 goto fail;
1492
1493             while( *off < DBLOCK && *off > 0 )
1494             {
1495                 printf( "%i ", *off );
1496                 off++;
1497                 blocks_used++;
1498             }
1499         }
1500
1501         /*double indirect*/
1502         if( i == TWO_INDIR - 1 && local->m_ino.i_block[i] )
1503         {
1504             /*get the block where the indirect block nums are stored*/
1505             k_t_get_blk( b, local->m_fd, local->m_ino.i_block[i] );
1506             off = (unsigned*)b;
1507             blocks_used ++;
1508
1509             /*pointer gymnastics*/
1510             while( *off < DBLOCK && *off > 0 )
1511             {
1512                 /*read all the single indirect blocks in the double-indirect loc*/
1513                 k_t_get_blk( b2, local->m_fd, *off );
1514                 off2 = (unsigned*)b2;
1515
1516                 /*print what they point at*/
1517                 while( *off2 < DBLOCK && *off2 > 0)
1518                 {
1519                     printf( "%i ", *off2 );
```

```
                    off2++;
                    blocks_used++;
                }

                printf( "%i ", *off );
                off++;
                blocks_used++;
            }
        }
    }

    /*make sure it is a valid request*/
    if( blocks_used <= 0 )
        goto fail;

    /*now delete them*/
    if( !k_t_del_dblocks_from_mino( local->m_fd, local->m_ino_num, blocks_used - 1 ) )
    {
        printf( "error removing blocks from inode\n" );
        goto fail;
    }

    /*update the file size*/
    local->m_ino.i_size = 0;

    return true;

fail:
    return false;
}


/*****************************************************************************
function    : bool k_t_set_imap_bit( int fd, u32 bnum, bool bval )
author      : nc
desc        : set a bit in the bitmap
date        :
 *****************************************************************************/
bool k_t_set_imap_bit( int fd, u32 bnum, bool bval )
{
    mnt_t* mt               = NULL;
    int bit                 = 0;
    int byte                = 0;
    char inode_bmap[BLK_SZ] = { 0 };

    if( !fd )
        goto fail;

    mt = k_t_get_mnt_from_fd( fd );

    /*cant operate on unlocked fs*/
    if( !mt || mt->m_busy == false )
        goto fail;

    /*get the the inode block*/
    if( !k_t_get_blk( inode_bmap, fd, mt->m_gd0.bg_inode_bitmap ) )
        goto fail;

    bnum -= 1;

    /*get bit/byte loc*/
    byte = bnum / 8;
    bit = bnum % 8;

    if( byte >= BLK_SZ )
        goto fail;

    /*set it*/
    if( bval == 1 )
    {
        inode_bmap[byte] |= ( 1 << bit );
        mt->m_sb.s_free_inodes_count--;
    }
    else
    {
        inode_bmap[byte] &= ~( 1 << bit );
        mt->m_sb.s_free_inodes_count++;
    }

    if( !k_t_put_blk( inode_bmap, fd, mt->m_gd0.bg_inode_bitmap ) )
        goto fail;
```

```
1601
1602      return true;
1603
1604  fail:
1605      printf( "unable to set bit %i in inode map\n", bnum );
1606      return false;
1607  }
1608
1609
1610  /*****************************************************************************
1611  function    : bool k_t_get_imap_bit( int fd, u32 bnum, bool* retv )
1612  author      : nc
1613  desc        : get a bit from the imap
1614  date        :
1615   *****************************************************************************/
1616  bool k_t_get_imap_bit( int fd, u32 bnum, bool* retv )
1617  {
1618      mnt_t* mt               = NULL;
1619      int bit                 = 0;
1620      int byte                = 0;
1621      int tmp                 = 0;
1622      char inode_bmap[BLK_SZ] = { 0 };
1623
1624      if( !fd || !retv )
1625          goto fail;
1626
1627      mt = k_t_get_mnt_from_fd( fd );
1628
1629      /*cant operate on unlocked fs*/
1630      if( !mt || mt->m_busy == false )
1631          goto fail;
1632
1633      /*get the the inode block*/
1634      if( !k_t_get_blk( inode_bmap, fd, mt->m_gd0.bg_inode_bitmap ) )
1635          goto fail;
1636
1637      /*get bit/byte loc*/
1638      byte = bnum / 8;
1639      bit = bnum % 8;
1640
1641      if( byte >= BLK_SZ )
1642          goto fail;
1643
1644      /*get it*/
1645      tmp = inode_bmap[byte] & ( 1 << bit );
1646
1647      if( tmp == 0 )
1648          *retv = false;
1649      else
1650          *retv = true;
1651
1652      return true;
1653
1654  fail:
1655      printf( "unable to get bit %i in inode map\n", bnum );
1656      return false;
1657  }
1658
1659
1660
1661  /*****************************************************************************
1662  function    : bool k_t_set_bmap_bit( int fd, u32 bnum, bool bval )
1663  author      : nc
1664  desc        : set a bit in the data block bitmap
1665  date        :
1666   *****************************************************************************/
1667  bool k_t_set_bmap_bit( int fd, u32 bnum, bool bval )
1668  {
1669      mnt_t* mt           = NULL;
1670      int bit             = 0;
1671      int byte            = 0;
1672      char bmap[BLK_SZ]   = { 0 };
1673
1674      if( !fd )
1675          goto fail;
1676
1677      mt = k_t_get_mnt_from_fd( fd );
1678
1679      /*cant operate on unlocked fs*/
1680      if( !mt || mt->m_busy == false )
1681          goto fail;
```

```
1682
1683      /*get the the inode block*/
1684      if( !k_t_get_blk( bmap, fd, mt->m_gd0.bg_block_bitmap ) )
1685          goto fail;
1686
1687      bnum--;
1688
1689      /*get bit/byte loc*/
1690      byte = bnum / 8;
1691      bit = bnum % 8;
1692
1693      if( byte >= BLK_SZ )
1694          goto fail;
1695
1696      /*set it*/
1697      if( bval == 1 )
1698      {
1699          bmap[byte] |= ( 1 << bit );
1700          mt->m_sb.s_free_blocks_count --;
1701      }
1702      else
1703      {
1704          bmap[byte] &= ~( 1 << bit );
1705          mt->m_sb.s_free_blocks_count ++;
1706      }
1707
1708
1709      /*hax*/
1710      if( !k_t_put_blk( bmap, fd, mt->m_gd0.bg_block_bitmap ) )
1711          goto fail;
1712
1713      return true;
1714
1715  fail:
1716      printf( "unable to set bit %i in block bitmap\n", bnum );
1717      return false;
1718  }
1719
1720
1721  /*****************************************************************************
1722  function    : bool k_t_get_bmap_bit( int fd, u32 bnum, bool* retv )
1723  author      : nc
1724  desc        : get a bit from the data block bitmap
1725  date        :
1726   *****************************************************************************/
1727  bool k_t_get_bmap_bit( int fd, u32 bnum, bool* retv )
1728  {
1729      mnt_t* mt               = NULL;
1730      int bit                 = 0;
1731      int byte                = 0;
1732      int tmp                 = 0;
1733      char inode_bmap[BLK_SZ] = { 0 };
1734
1735      if( !fd || !retv )
1736          goto fail;
1737
1738      mt = k_t_get_mnt_from_fd( fd );
1739
1740      /*cant operate on unlocked fs*/
1741      if( !mt || mt->m_busy == false )
1742          goto fail;
1743
1744      /*get the the inode block*/
1745      if( !k_t_get_blk( inode_bmap, fd, mt->m_gd0.bg_block_bitmap ) )
1746          goto fail;
1747
1748      /*get bit/byte loc*/
1749      byte = bnum / 8;
1750      bit = bnum % 8;
1751
1752      if( byte >= BLK_SZ )
1753          goto fail;
1754
1755      /*get it*/
1756      tmp = inode_bmap[byte] & ( 1 << bit );
1757
1758      if( tmp == 0 )
1759          *retv = false;
1760      else
1761          *retv = true;
1762
```

```
1763      return true;
1764
1765  fail:
1766      printf( "unable to get bit %i in block bitmap\n", bnum );
1767      return false;
1768  }
1769
1770
1771  /*****************************************************************************
1772  function    : u32 k_t_get_next_imap_loc( int fd )
1773  author      : nc
1774  desc        : get the next open position in the imap
1775  date        : 04-10-13
1776   *****************************************************************************/
1777  u32 k_t_get_next_imap_loc( int fd )
1778  {
1779      mnt_t* mt               = NULL;
1780      int bit                 = 0;
1781      int byte                = 0;
1782      char inode_bmap[BLK_SZ] = { 0 };
1783
1784      if( !fd )
1785          goto fail;
1786
1787      mt = k_t_get_mnt_from_fd( fd );
1788
1789      /*get the the inode block*/
1790      if( !k_t_get_blk( inode_bmap, fd, mt->m_gd0.bg_inode_bitmap ) )
1791          goto fail;
1792
1793      /*start walking*/
1794      for( byte = 0; byte < BLK_SZ; byte++ )
1795      {
1796          for( bit = 0; bit < 8; bit++ )
1797          {
1798              /*check for end of imap*/
1799              if( ( byte * 8 ) + bit > mt->m_sb.s_inodes_count )
1800              {
1801                  printf( "exceeded max_inodes\n" );
1802                  goto fail;
1803              }
1804
1805              if( ( inode_bmap[byte] & ( 1 << bit ) ) == 0 )
1806              {
1807                  goto ok;
1808              }
1809          }
1810      }
1811
1812      /*shouldn't get here*/
1813      goto fail;
1814
1815  ok:
1816      /*have to add one*/
1817      return( ( byte * 8 ) + bit  + 1 );
1818
1819  fail:
1820      printf( "all inode locations are reported as full\n" );
1821      return 0;
1822  }
1823
1824
1825  /*****************************************************************************
1826  function    : u32 k_t_get_next_bmap_loc( int fd )
1827  author      : nc
1828  desc        : get the next open position in the bmap
1829  date        : 04-10-13
1830   *****************************************************************************/
1831  u32 k_t_get_next_bmap_loc( int fd )
1832  {
1833      mnt_t* mt               = NULL;
1834      int bit                 = 0;
1835      int byte                = 0;
1836      char dbmap[BLK_SZ]  = { 0 };
1837
1838      if( !fd )
1839          goto fail;
1840
1841      mt = k_t_get_mnt_from_fd( fd );
1842
1843      /*get the the inode block*/
```

```
1844      if( !k_t_get_blk( dbmap, fd, mt->m_gd0.bg_block_bitmap ) )
1845          goto fail;
1846
1847      /*start walking*/
1848      for( byte = 0; byte < BLK_SZ; byte++ )
1849      {
1850          for( bit = 0; bit < 8; bit++ )
1851          {
1852              /*check for end of bmap*/
1853              if( ( byte * 8 ) + bit > mt->m_sb.s_blocks_count )
1854              {
1855                  printf( "exceeded max_dblocks\n" );
1856                  goto fail;
1857              }
1858
1859              if( ( dbmap[byte] & ( 1 << bit ) ) == 0 )
1860              {
1861                  goto ok;
1862              }
1863          }
1864      }
1865
1866      /*shouldn't get here*/
1867      goto fail;
1868
1869  ok:
1870      return( ( byte * 8 ) + bit + 1);
1871
1872  fail:
1873      printf( "all bmap locations are reported as full\n" );
1874      return 0;
1875  }
1876
1877
1878
1879  /*****************************************************************************
1880  function    : void k_t_p_dir_items( inode* ino, int fd, unsigned num )
1881  author      : nc
1882  desc        : forces a listing of all dir items (old)
1883  date        :
1884   *****************************************************************************/
1885  void k_t_p_dir_items( inode* ino, int fd, unsigned num )
1886  {
1887      int i           = 0;
1888      ext2_dir* cdir  = NULL;
1889      char* off       = NULL;
1890      char name[256]  = { '\0' };
1891      char b[BLK_SZ]  = { '\0' };
1892
1893      if( !ino || !S_ISDIR( ino->i_mode ) )
1894          return;
1895
1896      printf( "\ndirectory entries for inode %i:\n", num );
1897
1898      /*print header*/
1899      printf( "num\t\trec_len\t\tname_len\tascii name\n" );
1900
1901      for( i = 0; i < MAX_DIRECT && ino->i_block[i]; i++ )
1902      {
1903          /*get the block*/
1904          k_t_get_blk( b, fd, ino->i_block[i] );
1905          cdir = (ext2_dir*)b;
1906          off = (char*)cdir;
1907
1908          while( cdir->name_len > 0 )
1909          {
1910              /*copy the name over*/
1911              memset( name, 0, 256 );
1912              memcpy( name, cdir->name, cdir->name_len );
1913
1914              /*print the name*/
1915              printf( "%i\t\t", cdir->inode );
1916              printf( "%i\t\t", cdir->rec_len );
1917              printf( "%i\t\t", cdir->name_len );
1918              //printf( ( S_ISDIR( ino->i_mode ) ) ? "d\t\t" : "-\t\t" );
1919              printf( "%s", name );
1920              printf( "\n" );
1921
1922              /*advance the pointers*/
1923              off += cdir->rec_len;
1924              cdir = (ext2_dir*)off;
```

```
1925          }
1926      }
1927
1928      return;
1929  }
1930
1931
1932
1933  /*****************************************************************************
1934  function    : void k_t_get_num_and_fd( u32* num_dst, int* fd_dst, char* path )
1935  author      : nc
1936  desc        : get an inode number and from a path
1937  date        : 04-01-13
1938   *****************************************************************************/
1939  bool k_t_get_num_and_fd( u32* num_dst, int* fd_dst, mino_t* cwd, char* path )
1940  {
1941      int i                   = 0;
1942      mino_t* mino            = { 0 };    //inode holder
1943      char p1[MAX_PATH]       = { 0 };    //absolute part of path
1944      u32 nums[MAX_MINOS]     = { 0 };    //minos opened during traversal
1945      char* pels[MAX_DEPTH]   = { 0 };    //path elements
1946      int pels_num            = 0;        //number of path elements
1947      u32 last_ino            = 0;
1948
1949      /*TODO fix path truncation*/
1950
1951      if( !num_dst || !fd_dst || !path || !kr || !kr->m_mnt_tb[0] )
1952          goto fail;
1953
1954      /*if we got a relative path with no cwd can't do anything*/
1955      if( !is_abs_path( path ) && cwd == NULL )
1956          goto fail;
1957
1958      /*combine the relative and absolute parts of the path*/
1959      if( !is_abs_path( path ) )
1960      {
1961          /*get minos path name*/
1962      }
1963
1964      /*join the paths*/
1965      strcpy( p1, path );
1966
1967      if( !strcmp( p1, "/" ) )
1968      {
1969          *num_dst = ROOT_INO_NUM;
1970          *fd_dst = kr->m_mnt_tb[0]->m_fd;
1971          goto ok;
1972      }
1973
1974      /*split up the path*/
1975      split_path( p1, MAX_DEPTH, pels, &pels_num );
1976
1977      if( pels_num == 0 )
1978          goto fail;
1979
1980      /*get the start inode*/
1981      k_t_get_mino( kr->m_mnt_tb[0]->m_fd, ROOT_INO_NUM, &mino );
1982      last_ino = ROOT_INO_NUM;
1983
1984      for( i = 0; i < MAX_DEPTH && pels[i]; i ++ )
1985      {
1986          nums[i] = last_ino;
1987
1988          /*get the inode number for the current directory*/
1989          if( ( last_ino = k_t_find_dir_num( mino, pels[i] ) ) == -1 )
1990              goto fail;
1991
1992          if( mino->m_mounted )
1993          {
1994              /*check for mount bound crossing here*/
1995          }
1996
1997          /*get the inode we just found and search it for the current pel*/
1998          if( ( k_t_get_mino( mino->m_fd, last_ino, &mino ) ) == -1 )
1999              goto fail;
2000
2001          /*is it a dir?*/
2002          if( ( S_ISDIR( mino->m_ino.i_mode ) ) == 0 )
2003              break;
2004      }
2005
```

```
2006      /*done messed up*/
2007      if( pels[i+1] )
2008          goto fail;
2009
2010  ok:
2011      /*release everything we touched*/
2012      for( i = 0; i < MAX_MINOS; i++ )
2013      {
2014          if( nums[i] )
2015              k_t_put_mino( nums[i] );
2016      }
2017
2018      return true;
2019
2020  fail:
2021      /*release everything we touched*/
2022      for( i = 0; i < MAX_MINOS; i++ )
2023      {
2024          if( nums[i] )
2025              k_t_put_mino( nums[i] );
2026      }
2027
2028      printf( "failed to resolve inode\n" );
2029      return false;
2030  }
2031
2032
2033  /*****************************************************************************
2034  function    : int k_t_find_dir_num( mino_t* mino, char* dir )
2035  author      : nc
2036  desc        : find the inode number for a directory
2037  date        : 03-12-13
2038   *****************************************************************************/
2039  int k_t_find_dir_num( mino_t* mino, char* dir )
2040  {
2041      int i           = 0;
2042      ext2_dir* cdir  = NULL;
2043      char* off       = NULL;
2044      char name[256]  = { 0 };
2045      char b[BLK_SZ]  = { 0 };
2046
2047      if( !mino || !dir )
2048          goto fail;
2049
2050
2051      for( i = 0; i < MAX_DIRECT && mino->m_ino.i_block[i]; i++ )
2052      {
2053          /*get the block*/
2054          k_t_get_blk( b, mino->m_fd, mino->m_ino.i_block[i] );
2055
2056          /*see if there is a match*/
2057          cdir = (ext2_dir*)b;
2058          off = (char*)cdir;
2059
2060          while( off < (char*)( b + BLK_SZ ) )
2061          {
2062              /*get the name*/
2063              memset( name, 0, 256 );
2064              memcpy( name, cdir->name, cdir->name_len );
2065
2066              /*found it*/
2067              if( !strcmp( name, dir ) )
2068                  goto out;
2069
2070              /*no match, so advance the pointers*/
2071              off += cdir->rec_len;
2072              cdir = (ext2_dir*)off;
2073          }
2074      }
2075
2076      /*got through the loop and couldn't find anything*/
2077      goto fail;
2078
2079  out:
2080      return cdir->inode;
2081
2082  fail:
2083      return -1;
2084  }
2085
2086
```

```
2087
2088 /*****************************************************************************
2089 function    : i32 k_t_find_child_by_name( mino_t* current, char* cname )
2090 author      : nc
2091 desc        : find a child in some dblocks
2092 date        : 04-15-13
2093  *****************************************************************************/
2094 i32 k_t_find_child_by_name( mino_t* mino, char* cname )
2095 {
2096     int i           = 0;
2097     ext2_dir* cdir  = NULL;
2098     char* off       = NULL;
2099     char name[256]  = { 0 };
2100     char b[BLK_SZ]  = { 0 };
2101
2102     if( !mino || !cname )
2103         goto fail;
2104
2105
2106     for( i = 0; i < MAX_DIRECT && mino->m_ino.i_block[i]; i++ )
2107     {
2108         /*get the block*/
2109         k_t_get_blk( b, mino->m_fd, mino->m_ino.i_block[i] );
2110
2111         /*see if there is a match*/
2112         cdir = (ext2_dir*)b;
2113         off = (char*)cdir;
2114
2115         while( off < (char*)( b + BLK_SZ ) )
2116         {
2117             /*get the name*/
2118             memset( name, 0, 256 );
2119             memcpy( name, cdir->name, cdir->name_len );
2120
2121             /*found it*/
2122             if( !strcmp( name, cname ) )
2123                 goto out;
2124
2125             /*no match, so advance the pointers*/
2126             off += cdir->rec_len;
2127             cdir = (ext2_dir*)off;
2128         }
2129     }
2130
2131     /*got through the loop and couldn't find anything*/
2132     goto fail;
2133
2134 out:
2135     return cdir->inode;
2136
2137 fail:
2138     return -1;
2139 }
2140
2141
2142
2143 /*****************************************************************************
2144 function    : i32 k_t_find_child_by_name( mino_t* current, char* cname )
2145 author      : nc
2146 desc        : find a child in some dblocks
2147 date        : 04-15-13
2148  *****************************************************************************/
2149 bool k_t_add_child( mino_t* parent, i32 inum, char* cname )
2150 {
2151     char b[BLK_SZ]          = { '\0' };
2152     char* cp                = NULL;
2153     ext2_dir* dp            = NULL;
2154     i32 tmp_size            = 0;
2155
2156
2157     /*get the data block*/
2158     if( !k_t_get_blk( b, parent->m_fd, parent->m_ino.i_block[0] ) )
2159     {
2160         printf( "error retrieving data block\n" );
2161         goto fail;
2162     }
2163
2164     /*find the last entry*/
2165     cp = b;
2166     dp = (ext2_dir*)cp;
2167
```

```
2168      while( cp + dp->rec_len < ( b + BLK_SZ ) )
2169      {
2170          cp += dp->rec_len;
2171          dp = (ext2_dir*)cp;
2172      }
2173
2174      tmp_size = dp->rec_len;
2175
2176      /*adjust the intermediate file's length*/
2177      dp->rec_len = 4 * ( ( 8 + dp->name_len + 3 ) / 4 );
2178
2179      /*subtract it from the temoprary*/
2180      tmp_size -= dp->rec_len;
2181
2182      /*now add the new entry to the directory*/
2183      cp += dp->rec_len;
2184      dp = (ext2_dir*)cp;
2185
2186      /*TODO check size*/
2187
2188      dp->inode = inum;
2189      dp->name_len = strlen( cname );
2190      dp->rec_len = tmp_size;
2191      dp->file_type = FILE_TYPE;
2192      strcpy( dp->name, cname );
2193
2194      /*write the block*/
2195      if( !k_t_put_blk( b, parent->m_fd, parent->m_ino.i_block[0] ) )
2196      {
2197          printf( "error putting block\n" );
2198          goto fail;
2199      }
2200
2201
2202      return true;
2203  fail:
2204      return false;
2205  }
2206
2207  /*****************************************************************************
2208  function    : bool k_t_remove_child( mino_t* parent, char* cname )
2209  author      : nc
2210  desc        : remove a child from the parent mino's directory
2211  date        : 04-15-13
2212   *****************************************************************************/
2213  bool k_t_remove_child( mino_t* parent, char* cname )
2214  {
2215      char b[BLK_SZ]          = { '\0' };
2216      char name[MAX_NAME]     = { '\0' };
2217      char* cp                = NULL;
2218      int tmp_size            = 0;
2219      ext2_dir* dp            = NULL;
2220
2221      if( !parent || !cname )
2222          goto fail;
2223
2224      /*get the data block*/
2225      k_t_get_blk( b, parent->m_fd, parent->m_ino.i_block[0] );
2226
2227      cp = b;
2228      dp = (ext2_dir*)cp;
2229
2230      /*try to find the entry*/
2231      while( cp < ( b + BLK_SZ ) )
2232      {
2233          memset( name, 0, MAX_NAME );
2234          strncpy( name, dp->name, dp->name_len );
2235          if( !strcmp( name, cname ) )
2236              break;
2237
2238          cp += dp->rec_len;
2239          dp = (ext2_dir*)cp;
2240      }
2241
2242      /*did we walk past?*/
2243      if( cp >= ( b + BLK_SZ ) )
2244      {
2245          if( parent->m_ino.i_block[1] == 0 )
2246              goto fail;
2247          else
2248              k_t_del_dblocks_from_mino( parent->m_fd, parent->m_ino_num, 1 );
```

```
2249      }
2250
2251      /*shift some bits*/
2252      tmp_size = dp->rec_len;
2253      memset( dp, 0, tmp_size );
2254
2255      while( ( cp + tmp_size ) < ( b + BLK_SZ ) )
2256      {
2257          *cp = *(cp + tmp_size);
2258          cp++;
2259      }
2260
2261      /*update the size of the last dir entry*/
2262      cp = b;
2263      dp = (ext2_dir*)cp;
2264
2265      while( ( cp + dp->rec_len + tmp_size ) < ( b + BLK_SZ ) )
2266      {
2267          cp += dp->rec_len;
2268          dp = (ext2_dir*)cp;
2269      }
2270
2271      dp->rec_len +=  tmp_size;
2272
2273
2274      /*put the block back*/
2275      k_t_put_blk( b, parent->m_fd, parent->m_ino.i_block[0] );
2276
2277      return true;
2278
2279  fail:
2280      printf( "error removing child %s\n", cname );
2281      return false;
2282  }
2283
2284
2285
2286  /*****************************************************************************
2287  function    : i32 k_t_find_oft_entry( mino_t* entry )
2288  author      : nc
2289  desc        : find an open entry in the open file table
2290  date        :
2291   *****************************************************************************/
2292  i32 k_t_find_oft_entry( mino_t* entry )
2293  {
2294      int i = 0;
2295
2296      for( i = 0; i < MAX_OFS; i++ )
2297      {
2298          if( kr->m_of_tb[i] && kr->m_of_tb[i]->m_minoptr == entry )
2299              break;
2300      }
2301
2302      if( i >= MAX_OFS )
2303          goto fail;
2304
2305      return i;
2306
2307  fail:
2308      return -1;
2309  }
2310
2311
2312
2313  /*****************************************************************************
2314  function    : bool k_t_open_oft_entry( ~ )
2315  author      : nc
2316  desc        : add an entry to the oft
2317  date        :
2318   *****************************************************************************/
2319  bool k_t_open_oft_entry( mino_t* entry, E_FILE_MODE mode, i32* fd_loc, char* path )
2320  {
2321      i32 ret_loc = 0;
2322
2323      if( !entry || !fd_loc || !path )
2324      {
2325          printf( "error: recieved improper arguemnts\n" );
2326          goto fail;
2327      }
2328
2329      /*see if it already exists*/
```

```
2330      ret_loc = k_t_find_oft_entry( entry );
2331
2332      /*it is already open so we have to check its mode*/
2333      if( ret_loc != -1 )
2334      {
2335          if( mode != M_READ )
2336          {
2337              printf( "error: file open in incompatible mode\n" );
2338              goto fail;
2339          }
2340      }
2341      else
2342      {
2343          /*not open so open it*/
2344          for( ret_loc = 0; ret_loc < MAX_OFS; ret_loc++ )
2345          {
2346              if( kr->m_of_tb[ret_loc] == NULL )
2347                  break;
2348          }
2349      }
2350
2351      /*make sure ret_loc is within the array*/
2352      if( ret_loc >=  MAX_OFS )
2353      {
2354          printf( "error: no open locations in kernel file table\n" );
2355          goto fail;
2356      }
2357
2358      /*alredy exists so modify it*/
2359      if( kr->m_of_tb[ret_loc] )
2360      {
2361          kr->m_of_tb[ret_loc]->m_refc++;
2362          kr->m_of_tb[ret_loc]->m_off = 0;
2363      }
2364      else
2365      {
2366          of_t_init( &kr->m_of_tb[ret_loc] );
2367          kr->m_of_tb[ret_loc]->m_mode = mode;
2368          kr->m_of_tb[ret_loc]->m_refc = 1;
2369          kr->m_of_tb[ret_loc]->m_minoptr = entry;
2370
2371          if( strlen ( path ) < MAX_NAME - 1 )
2372          {
2373              memset( kr->m_of_tb[ret_loc]->m_name, 0, MAX_NAME );
2374              strcpy( kr->m_of_tb[ret_loc]->m_name, path );
2375          }
2376
2377          /*different cases for differernt r/w modes*/
2378          switch( mode )
2379          {
2380              case M_READ:
2381                  kr->m_of_tb[ret_loc]->m_off = 0;
2382                  break;
2383              case M_WRITE:
2384                  k_t_truncate_mino( entry );
2385                  kr->m_of_tb[ret_loc]->m_off = 0;
2386                  break;
2387              case M_READWRITE:
2388                  kr->m_of_tb[ret_loc]->m_off = 0;
2389                  break;
2390              case M_APPEND:
2391                  kr->m_of_tb[ret_loc]->m_off = entry->m_ino.i_size;
2392                  break;
2393              default:
2394                  printf( "error: invalid mode for kft\n" );
2395                  goto fail;
2396                  break;
2397          }
2398      }
2399
2400      /*tell the caller what the file descriptor is*/
2401      *fd_loc = ret_loc;
2402
2403      return true;
2404
2405  fail:
2406      return false;
2407  }
2408
2409
2410
```

```
2411 /*****************************************************************************
2412 function    : bool k_t_close_oft_entry( mino_t* entry )
2413 author      : nc
2414 desc        : remove an entry from the oft
2415 date        :
2416  *****************************************************************************/
2417 bool k_t_close_oft_entry( mino_t* entry )
2418 {
2419     int loc = 0;
2420 
2421     if( !entry )
2422         goto fail;
2423 
2424     loc = k_t_find_oft_entry( entry );
2425 
2426     if( loc == -1 )
2427     {
2428         printf( "error: kft entry not open\n" );
2429         goto fail;
2430     }
2431 
2432     kr->m_of_tb[loc]->m_refc--;
2433 
2434     /*are we the last user?*/
2435     if( kr->m_of_tb[loc]->m_refc == 0 )
2436     {
2437         /*bleh*/
2438         k_t_put_mino( kr->m_of_tb[loc]->m_minoptr->m_ino_num );
2439         of_t_destroy( &kr->m_of_tb[loc] );
2440 
2441         /*learned the hard way*/
2442         kr->m_of_tb[loc] = NULL;
2443     }
2444 
2445     return true;
2446 
2447 fail:
2448     return false;
2449 }
2450 
2451 
2452 
2453 /*****************************************************************************
2454 function    : i64 k_t_lseek( i32 fd, i32 amount )
2455 author      : nc
2456 desc        : seek into a file descriptor
2457 date        :
2458  *****************************************************************************/
2459 i64 k_t_lseek( i32 fd, i32 amount )
2460 {
2461     i64 o_loc   = 0;
2462 
2463     /*get the table position and make sure it exists*/
2464     if( !kr->m_of_tb[fd] )
2465     {
2466         printf( "error: file desriptor does not exist\n" );
2467         goto fail;
2468     }
2469 
2470     /*we are supposed to return the origirnal location*/
2471     o_loc = kr->m_of_tb[fd]->m_off;
2472 
2473     /*seek the file descriptor*/
2474     if( amount < 0 )
2475     {
2476         printf( "warning: attempted to read before beginning of file\n" );
2477         kr->m_of_tb[fd]->m_off = 0;
2478     }
2479     else if( amount > kr->m_of_tb[fd]->m_minoptr->m_ino.i_size )
2480     {
2481         printf( "warning: attempted to read past end of file\n" );
2482         kr->m_of_tb[fd]->m_off = kr->m_of_tb[fd]->m_minoptr->m_ino.i_size;
2483     }
2484     else
2485     {
2486         kr->m_of_tb[fd]->m_off = amount;
2487     }
2488 
2489     return o_loc;
2490 
2491 fail:
```

```
2492      return -1;
2493
2494  }
2495
2496
2497  /*****************************************************************************
2498  function    : i32 k_t_write( i32 fd, i32 num, char* src)
2499  author      : nc
2500  desc        : write to a file descriptor
2501  date        :
2502   *****************************************************************************/
2503  i32 k_t_write( i32 fd, i32 num, char* src)
2504  {
2505      i32 bytes_writen    = 0;
2506      i32 blk         = 0;
2507      i32 byte        = 0;
2508      i32 add         = 0;
2509      i32 cblock_loc  = 0;
2510      i32 cblock_off  = 0;
2511      of_t* target    = NULL;
2512      char b[BLK_SZ]  = { '\0' };
2513      char buf[BLK_SZ] = { '\0' };
2514      char* cp        = NULL;
2515      unsigned* off   = 0;
2516
2517      if( !fd || num <= 0 || !src || !kr->m_of_tb[fd] )
2518          goto fail;
2519
2520      target = kr->m_of_tb[fd];
2521      cp = src;
2522
2523      if( target->m_mode != M_WRITE &&
2524              target->m_mode != M_READWRITE )
2525      {
2526          printf( "error: file is not open for write mode\n" );
2527          goto fail;
2528      }
2529
2530      target->m_minoptr->m_ino.i_size += num;
2531
2532      /*figure out how many blocks to add*/
2533      add += ( num / BLK_SZ );
2534
2535      if(add < 8)
2536          add += 1;
2537
2538      printf( "add %d\n", add );
2539      k_t_add_dblocks_to_mino( target->m_minoptr->m_fd, target->m_minoptr->m_ino_num, add );
2540      memset( buf, 0 , BLK_SZ );
2541      memset( b, 0, BLK_SZ );
2542
2543      while(num != 0)
2544      {
2545          /*find out our starting location*/
2546          blk = target->m_off / BLK_SZ;
2547          byte = target->m_off % BLK_SZ;
2548
2549          while( byte < BLK_SZ &&
2550                  num != 0 &&
2551                  target->m_off < target->m_minoptr->m_ino.i_size)
2552          {
2553              /*copy the data over*/
2554              strncat(b, cp, 1);
2555
2556              /*lots of counters*/
2557              cp++;
2558              target->m_off++;
2559              bytes_writen++;
2560              byte++;
2561              num--;
2562          }
2563
2564          /*find out indirection level*/
2565          if( blk <= DIR_RANGE )
2566          {
2567              k_t_put_blk(b, target->m_minoptr->m_fd, target->m_minoptr->m_ino.i_block[blk] );
2568          }
2569          if( blk > DIR_RANGE && blk <= OID_RANGE )
2570          {
2571              /*get indirect block and then get the data block*/
2572              k_t_get_blk( buf,
```

```
2573                    target->m_minoptr->m_fd,
2574                    target->m_minoptr->m_ino.i_block[ONE_INDIR - 1] );
2575
2576            /*the offset in this block is blk - 12*/
2577            off = (unsigned*)buf;
2578            off += ( blk - MAX_DIRECT );
2579
2580            if( !*off)
2581                printf( "error: no id (1id) data block\n" );
2582
2583            /*put the data block*/
2584            k_t_put_blk( b,
2585                    target->m_minoptr->m_fd,
2586                    *off );
2587
2588        }
2589        if( blk > OID_RANGE && blk <= TID_RANGE )
2590        {
2591            /*get indirect block and then get the data block*/
2592            k_t_get_blk( buf,
2593                    target->m_minoptr->m_fd,
2594                    target->m_minoptr->m_ino.i_block[TWO_INDIR - 1] );
2595
2596            /*now find out which 2id block to get - subtract one for offset*/
2597            cblock_loc = ( ( blk - MAX_DIRECT ) / ID_PER_BLK ) - 1;
2598            off = (unsigned*)buf;
2599            off += cblock_loc;
2600
2601            if( !*off )
2602            {
2603                printf( "error: no id (2id) data block\n" );
2604                goto fail;
2605            }
2606
2607            /*get the two id data block*/
2608            k_t_get_blk( buf,
2609                    target->m_minoptr->m_fd,
2610                    *off );
2611
2612            /*find out where the two id offset is*/
2613            cblock_off = ( ( blk - MAX_DIRECT ) % ID_PER_BLK );
2614            off = (unsigned*)buf;
2615            off += cblock_off;
2616
2617            if( !*off )
2618            {
2619                printf( "error: no 2id (2id) data block\n" );
2620                goto fail;
2621            }
2622
2623            /*get the two id data block*/
2624            k_t_put_blk( b,
2625                    target->m_minoptr->m_fd,
2626                    *off );
2627        }
2628        memset(b, 0, BLK_SZ);
2629
2630    }
2631
2632    return bytes_writen;
2633 fail:
2634
2635    return -1;
2636
2637
2638 }
2639
2640 /*****************************************************************************
2641 function    : i32 k_t_read( i32 fd, i32 num, char* dst )
2642 author      : nc
2643 desc        : read bytes from a file
2644 date        :
2645  *****************************************************************************/
2646 i32 k_t_read( i32 fd, i32 num, char* dst )
2647 {
2648     i32 bytes_read  = 0;
2649     i32 blk         = 0;
2650     i32 byte        = 0;
2651     i32 cblock_loc  = 0;
2652     i32 cblock_off  = 0;
2653     of_t* target    = NULL;
```

```
2654      char b[BLK_SZ]  = { '\0' };
2655      char* cp        = NULL;
2656      unsigned* off   = 0;
2657
2658      if( !fd || num <= 0 || !dst || !kr->m_of_tb[fd] )
2659          goto fail;
2660
2661      /*easier to work with*/
2662      target = kr->m_of_tb[fd];
2663      cp = dst;
2664
2665      if( target->m_mode != M_READ &&
2666              target->m_mode != M_READWRITE )
2667      {
2668          printf( "error: file is not open for read mode\n" );
2669          goto fail;
2670      }
2671
2672      while( num != 0 )
2673      {
2674          /*find out our starting location*/
2675          blk = target->m_off / BLK_SZ;
2676          byte = target->m_off % BLK_SZ;
2677
2678          /*find out indirection level*/
2679          if( blk <= DIR_RANGE )
2680          {
2681              k_t_get_blk( b, target->m_minoptr->m_fd, target->m_minoptr->m_ino.i_block[blk] );
2682          }
2683          if( blk > DIR_RANGE && blk <= OID_RANGE )
2684          {
2685              /*get indirect block and then get the data block*/
2686              k_t_get_blk( b,
2687                      target->m_minoptr->m_fd,
2688                      target->m_minoptr->m_ino.i_block[ONE_INDIR - 1] );
2689
2690              /*printf( "got id data block for mino %i at pos %i\n",
2691                      target->m_minoptr->m_ino_num,
2692                      blk );*/
2693
2694              /*the offset in this block is blk - 12*/
2695              off = (unsigned*)b;
2696              off += ( blk - MAX_DIRECT );
2697
2698              if( !*off )
2699                  printf( "error: no id (1id) data block\n" );
2700
2701              /*get the data block*/
2702              k_t_get_blk( b,
2703                      target->m_minoptr->m_fd,
2704                      *off );
2705
2706          }
2707          if( blk > OID_RANGE && blk <= TID_RANGE )
2708          {
2709              /*get indirect block and then get the data block*/
2710              k_t_get_blk( b,
2711                      target->m_minoptr->m_fd,
2712                      target->m_minoptr->m_ino.i_block[TWO_INDIR - 1] );
2713
2714              /*now find out which 2id block to get - subtract one for offset*/
2715              cblock_loc = ( ( blk - MAX_DIRECT ) / ID_PER_BLK ) - 1;
2716              off = (unsigned*)b;
2717              off += cblock_loc;
2718
2719              if( !*off )
2720              {
2721                  printf( "error: no id (2id) data block\n" );
2722                  goto fail;
2723              }
2724
2725              /*get the two id data block*/
2726              k_t_get_blk( b,
2727                      target->m_minoptr->m_fd,
2728                      *off );
2729
2730              /*find out where the two id offset is*/
2731              cblock_off = ( ( blk - MAX_DIRECT ) % ID_PER_BLK );
2732              off = (unsigned*)b;
2733              off += cblock_off;
2734
```

```
2735              if( !*off )
2736              {
2737                  printf( "error: no 2id (2id) data block\n" );
2738                  goto fail;
2739              }
2740
2741              /*get the two id data block*/
2742              k_t_get_blk( b,
2743                      target->m_minoptr->m_fd,
2744                      *off );
2745          }
2746
2747          /*loop while we are in this block*/
2748          while( byte < BLK_SZ &&
2749                  num != 0 &&
2750                  target->m_off < target->m_minoptr->m_ino.i_size )
2751          {
2752              /*copy the data over*/
2753              *cp = b[byte];
2754
2755              /*lots of counters*/
2756              cp++;
2757              target->m_off++;
2758              bytes_read++;
2759              byte++;
2760              num--;
2761          }
2762
2763          /*don't read past the end*/
2764          if( target->m_off > target->m_minoptr->m_ino.i_size )
2765              break;
2766
2767      }
2768
2769      target->m_minoptr->m_refc = 1;
2770      return bytes_read;
2771
2772  fail:
2773
2774      return -1;
2775  }
```

2.4 In-memory inodes

2.4.1 Definition

Listing 5: mino_t.h

```
1  /*****************************************************************************
2  file        : mino_t.h
3  author      : nc
4  desc        : mino_t struct
5  date        : 03-25-13
6   *****************************************************************************/
7
8  #pragma once
9
10 #include <ext2fs/ext2_fs.h>
11 #include "global_defs.h"
12 #include "mnt_t.h"
13
14
15 /*fw dec*/
16 typedef struct mino_t mino_t;
17 typedef struct mnt_t mnt_t;
18
19
20 /*struct*/
21 struct mino_t
22 {
23     inode   m_ino;
24     u16     m_fd;
25     u32     m_ino_num;
26     u16     m_refc;
27     bool    m_dirty;
28     bool    m_mounted;
29     mnt_t*  m_mountp;
30 };
```

```
31
32  bool    mino_t_init( mino_t** );
33  void    mino_t_destroy( mino_t** );
34
35  bool    mino_t_make(
36          mino_t*,
37          inode*,
38          u16,
39          u32,
40          u16,
41          bool,
42          bool,
43          mnt_t* );
44
45  bool    mino_t_set_ino(
46          inode* dst,
47          u16 uid,
48          u16 gid,
49          u32 size,
50          u16 mode,
51          u16 links );
52
53  void    mino_t_show( mino_t* );
54  void    mino_t_show_ino( inode*, unsigned );
```

2.4.2 Implementation

Listing 6: mino_t.c

```
1   /*****************************************************************************
2   file        : mino_t.c
3   author      : nc
4   desc        : implementation of the mino_t functions
5   date        : 03-25-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <string.h>
10  #include <stdio.h>
11  #include <ext2fs/ext2_fs.h>
12  #include <time.h>
13
14  #include "global_defs.h"
15  #include "mino_t.h"
16  #include "mnt_t.h"
17
18
19  /*****************************************************************************
20  function    :
21  author      : nc
22  desc        :
23  date        : 03-24-13
24   *****************************************************************************/
25  bool mino_t_init( mino_t** ino )
26  {
27      *ino = malloc( sizeof(**ino) );
28
29      if( !*ino )
30          goto fail;
31
32      memset( *ino, 0, sizeof(**ino) );
33
34      return true;
35
36  fail:
37      return false;
38  }
39
40
41  /*****************************************************************************
42  function    :
43  author      : nc
44  desc        :
45  date        : 03-24-13
46   *****************************************************************************/
47  void mino_t_destroy( mino_t** ino )
48  {
49      if( *ino )
50          free( *ino );
```

```
51
52      *ino = NULL;
53
54      return;
55  }
56
57
58
59  /*****************************************************************************
60  function    : bool mino_t_create( ~ )
61  author      : nc
62  desc        : create an mino_t from args
63  date        : 03-31-13
64   *****************************************************************************/
65  bool mino_t_make( mino_t* dst,
66          inode* ino,
67          u16 fd,
68          u32 num,
69          u16 refs,
70          bool dirty,
71          bool mounted,
72          mnt_t* point )
73  {
74      if( !dst || !ino || fd <= 0 || !num || ( mounted && !point ) )
75          goto fail;
76
77      memcpy( &dst->m_ino, ino, sizeof(*ino) );
78      dst->m_fd = fd;
79      dst->m_ino_num = num;
80      dst->m_refc = refs;
81      dst->m_dirty = dirty;
82      dst->m_mounted = mounted;
83      dst->m_mountp = point;
84
85      return true;
86
87  fail:
88      return false;
89  }
90
91
92  /*****************************************************************************
93  function    :
94  author      : nc
95  desc        :
96  date        : 03-31-13
97   *****************************************************************************/
98  void mino_t_show( mino_t* mi )
99  {
100     if( !mi )
101         return;
102
103
104     printf( "mino_t %p:\n", mi );
105     mino_t_show_ino( &mi->m_ino, mi->m_ino_num );
106
107     printf( "\n" );
108     printf( "\tm_fd:\t\t%i\n", mi->m_fd );
109     printf( "\tm_ino_num:\t%i\n", mi->m_ino_num );
110     printf( "\tm_refc:\t\t%i\n", mi->m_refc );
111     printf( "\tm_dirty:\t%i\n", mi->m_dirty );
112     printf( "\tm_mounted:\t%i\n", mi->m_mounted );
113     printf( "\tm_mountp:\t%p\n", mi->m_mountp );
114
115     printf( "\n" );
116
117     return;
118 }
119
120
121 /*****************************************************************************
122 function    :
123 author      : nc
124 desc        :
125 date        : 03-12-13
126  *****************************************************************************/
127 void mino_t_show_ino( inode* ino, unsigned loc )
128 {
129     printf( "\tinode %d info:\n", loc );
130     printf( "\tuid:\t\t%d\n", ino->i_uid );
131     printf( "\tsize:\t\t%d\n", ino->i_size );
```

```
132     printf( "\tlinks:\t\t%d\n", ino->i_links_count );
133     return;
134 }
135 
136 
137 /*****************************************************************************
138 function    :
139 author      : nc
140 desc        :
141 date        : 03-12-13
142  *****************************************************************************/
143 bool mino_t_set_ino(
144         inode* dst,
145         u16 uid,
146         u16 gid,
147         u32 size,
148         u16 mode,
149         u16 links )
150 {
151     if( !dst || !mode )
152         goto fail;
153 
154     dst->i_mode         = mode;
155     dst->i_uid          = uid;
156     dst->i_gid          = gid;
157     dst->i_size         = size;
158     dst->i_links_count  = links;
159     dst->i_atime        = time(0L);
160     dst->i_ctime        = time(0L);
161     dst->i_mtime        = time(0L);
162     dst->i_blocks       = size / IE_SZ;
163 
164     return true;
165 
166 fail:
167     printf( "could not create new inode\n" );
168 
169     return false;
170 }
```

2.5 Mount points

2.5.1 Definition

Listing 7: mnt_t.h

```
1  /*****************************************************************************
2  file        : mnt_t.h
3  author      : nc
4  desc        : mnt_t struct
5  date        : 03-25-13
6   *****************************************************************************/
7  
8  #pragma once
9  
10 #include "global_defs.h"
11 #include "mino_t.h"
12 
13 
14 /*fw dec*/
15 typedef struct mnt_t mnt_t;
16 typedef struct mino_t mino_t;
17 
18 
19 /*struct*/
20 struct mnt_t
21 {
22     int         m_num_inos;
23     int         m_num_blocks;
24     int         m_first_ino;
25     int         m_ino_start;
26     int         m_data_start;
27     int         m_fd;
28     bool        m_busy;
29     mino_t*     m_root_ino;
30     char        m_dev_name[MAX_NAME];
31     char        m_mnt_name[MAX_NAME];
32     sb          m_sb;
```

```
33      gd          m_gd0;
34  };
35
36  bool    mnt_t_init( mnt_t** );
37  void    mnt_t_destroy( mnt_t** );
38  void    mnt_t_show( mnt_t* );
39  bool    mnt_t_create( mnt_t* mt, char* path_to_img, char* folder_name, u32 inum );
40  bool    mnt_t_add_root_mino( mnt_t* target, mino_t* root );
41  void    mnt_t_lock( mnt_t* args );
42  void    mnt_t_unlock( mnt_t* args );
```

2.5.2 Implementation

Listing 8: mnt_t.c

```
1   /*************************************************************************
2   file        : mnt_t.c
3   author      : nc
4   desc        : implementation of the mnt_t functions
5   date        : 03-25-13
6    *************************************************************************/
7
8   #include <stdlib.h>
9   #include <string.h>
10  #include <stdio.h>
11  #include <unistd.h>
12  #include <fcntl.h>
13
14  #include "global_defs.h"
15  #include "mnt_t.h"
16  #include "k_t.h"            //get_blk
17
18
19  /*************************************************************************
20  function    : bool mnt_t_init( mnt_t** mt )
21  author      : nc
22  desc        : create a mount point pointer
23  date        : 03-24-13
24   *************************************************************************/
25  bool mnt_t_init( mnt_t** mt )
26  {
27      *mt = malloc( sizeof(**mt) );
28
29      if( !*mt )
30          goto fail;
31
32      memset( *mt, 0, sizeof(**mt) );
33
34      return true;
35
36  fail:
37      return false;
38  }
39
40
41  /*************************************************************************
42  function    : void mnt_t_destroy( mnt_t** mt )
43  author      : nc
44  desc        : destroy a moint point
45  date        : 03-24-13
46   *************************************************************************/
47  void mnt_t_destroy( mnt_t** mt )
48  {
49      if( (*mt)->m_fd )
50          close( (*mt)->m_fd );
51
52      if( *mt )
53          free( *mt );
54
55      *mt = NULL;
56
57      return;
58  }
59
60
61  /*************************************************************************
62  function    : bool mnt_t_create( mnt_t* mt, char* path_to_img, char* folder_name )
63  author      : nc
64  desc        : create a new mount point from arguments
```

```
65  date        : 04-07-13
66   ****************************************************************************/
67  bool mnt_t_create( mnt_t* mt, char* path_to_img, char* folder_name, u32 inum )
68  {
69      int fd          = 0;
70      char b[BLK_SZ]  = { 0 };
71
72      if( !mt || !path_to_img || !folder_name )
73          goto fail;
74
75      if( inum != ROOT_INO_NUM )
76          printf( "warning: attempting to mount non-standard root inode\n" );
77
78      /*open the image to mount*/
79      fd = open( path_to_img, O_RDWR );
80
81      if( fd <= 0 )
82          goto fail;
83
84      /*assign the basic values*/
85      mt->m_fd = fd;
86      strcpy( mt->m_dev_name, path_to_img );
87      strcpy( mt->m_mnt_name, folder_name );
88
89      /*get the superblock and gd0*/
90      k_t_get_blk( (char*)&mt->m_sb, fd, SB_BLK );
91      k_t_get_blk( b, fd, GD0_BLK );
92      memcpy( &mt->m_gd0, b, sizeof(gd) );
93
94      /*check magic num*/
95      if( mt->m_sb.s_magic != EXT2_MAGIC )
96          goto fail;
97
98      /*get some basic values*/
99      mt->m_num_inos = mt->m_sb.s_inodes_count;
100     mt->m_num_blocks = mt->m_sb.s_blocks_count;
101     mt->m_first_ino = mt->m_sb.s_first_ino;
102     mt->m_data_start = mt->m_sb.s_first_data_block;
103
104     mt->m_ino_start = mt->m_gd0.bg_inode_table;
105
106     /*set the mount point name and device name*/
107     strcpy( mt->m_dev_name, path_to_img );
108     strcpy( mt->m_mnt_name, folder_name );
109
110     /*set the root mount point*/
111     if( mt->m_root_ino )
112         goto fail;
113
114     k_t_get_mino( fd, inum, &mt->m_root_ino );
115     k_t_set_mino_as_mounted( fd, inum, mt );
116
117     return true;
118
119 fail:
120     printf( "unable to create mount point %s\n", path_to_img );
121     return false;
122 }
123
124
125
126 /*****************************************************************************
127 function    : bool mnt_t_add_root_mino( mnt_t target, mino_t* root )
128 author      : nc
129 desc        : add the root inode to a mount point
130 date        : 04-07-13
131  ****************************************************************************/
132 bool mnt_t_add_root_mino( mnt_t* target, mino_t* root )
133 {
134     if( !target || !root || target->m_root_ino )
135         goto fail;
136
137     /*add the inode and set the busy status to false*/
138     target->m_root_ino = root;
139     target->m_busy = false;
140
141     return true;
142
143 fail:
144     return false;
145 }
```

```
146
147
148
149 /*****************************************************************************
150 function    :
151 author      : nc
152 desc        :
153 date        : 03-24-13
154  *****************************************************************************/
155 void mnt_t_show( mnt_t* mt )
156 {
157     if( !mt )
158         return;
159
160     printf( "mount entry information %p:\n", mt );
161     printf( "\tm_num_inos:\t%i\n", mt->m_num_inos );
162     printf( "\tm_num_blocks:\t%i\n", mt->m_num_blocks );
163     printf( "\tm_first_ino:\t%i\n", mt->m_first_ino );
164     printf( "\tm_ino_start:\t%i\n", mt->m_ino_start );
165     printf( "\tm_data_start:\t%i\n", mt->m_data_start );
166     printf( "\tm_fd:\t\t%i\n", mt->m_fd );
167     printf( "\tm_busy:\t\t%i\n", mt->m_busy );
168     printf( "\tm_root_ino:\t%p\n", mt->m_root_ino );
169     printf( "\tm_dev_name:\t%s\n", mt->m_dev_name );
170     printf( "\tm_mnt_name:\t%s\n", mt->m_mnt_name );
171     printf( "\n" );
172
173     return;
174 }
175
176
177 /*****************************************************************************
178 function    :
179 author      : nc
180 desc        :
181 date        : 03-24-13
182  *****************************************************************************/
183 void mnt_t_lock( mnt_t* args )
184 {
185     if( args )
186         args->m_busy = true;
187 }
188
189
190 /*****************************************************************************
191 function    :
192 author      : nc
193 desc        :
194 date        : 03-24-13
195  *****************************************************************************/
196 void mnt_t_unlock( mnt_t* args )
197 {
198     if( args )
199         args->m_busy = false;
200 }
```

2.6 File table entries

2.6.1 Definition

Listing 9: of_t.h

```
1  /*****************************************************************************
2  file        : of_t.h
3  author      : nc
4  desc        : of_t struct
5  date        : 03-24-13
6   *****************************************************************************/
7
8  #pragma once
9
10 #include "global_defs.h"
11 #include "mino_t.h"
12
13
14 /*fw dec*/
15 typedef struct of_t of_t;
16
```

```
17
18  /*struct*/
19  struct of_t
20  {
21      E_FILE_MODE     m_mode;
22      i32             m_refc;
23      mino_t*         m_minoptr;
24      i64             m_off;
25      char            m_name[MAX_NAME];
26  };
27
28  bool    of_t_init( of_t** );
29  void    of_t_destroy( of_t** );
```

2.6.2 Implementation

Listing 10: of_t.c

```
1   /*****************************************************************************
2   file        : of_t.c
3   author      : nc
4   desc        : implementation of the of_t functions
5   date        : 03-24-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <string.h>
10  #include "global_defs.h"
11  #include "of_t.h"
12
13
14  /*****************************************************************************
15  function    :
16  author      : nc
17  desc        :
18  date        : 03-24-13
19   *****************************************************************************/
20  bool of_t_init( of_t** ft )
21  {
22      *ft = malloc( sizeof(**ft) );
23
24      if( !*ft )
25          goto fail;
26
27      memset( *ft, 0, sizeof(**ft) );
28
29      return true;
30
31  fail:
32      return false;
33  }
34
35
36  /*****************************************************************************
37  function    :
38  author      : nc
39  desc        :
40  date        : 03-24-13
41   *****************************************************************************/
42  void of_t_destroy( of_t** ft )
43  {
44      if( *ft )
45          free( *ft );
46
47      *ft = NULL;
48
49      return;
50  }
```

2.7 Process type

2.7.1 Definition

Listing 11: proc_t.h

```
1   /*****************************************************************************
```

```
2  file        : proc_t.h
3  author      : nc
4  desc        : proc_t struct
5  date        : 03-24-13
6   ********************************************************************************/
7
8  #pragma once
9
10 #include "global_defs.h"
11 #include "of_t.h"
12 #include "mino_t.h"
13
14 /*fw dec*/
15 typedef struct proc_t proc_t;
16
17 /*struct*/
18 struct proc_t
19 {
20     i32         m_uid;
21     i32         m_pid;
22     i32         m_gid;
23     i32         m_parid;
24     i32         m_status;
25     proc_t*     m_parent;
26     mino_t*     m_cwd;
27     i32         m_fds[NUM_FDS];
28 };
29
30 bool    proc_t_init( proc_t** );
31 void    proc_t_destroy( proc_t** );
32 bool    proc_t_make(
33         proc_t*,
34         i16,
35         i16,
36         i16,
37         i16,
38         i16,
39         proc_t*,
40         mino_t* );
41
42 i32     proc_t_get_next_fd_loc( proc_t* );
43 bool    proc_t_add_fd_to_tb( proc_t* pr, i32 fd );
44 bool    proc_t_del_fd_from_tb( proc_t* pr, i32 fd );
```

2.7.2 Implementation

Listing 12: proc_t.c

```
1  /********************************************************************************
2  file        : proc_t.c
3  author      : nc
4  desc        : implementation of the proc_t functions
5  date        : 03-24-13
6   ********************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include "global_defs.h"
12 #include "proc_t.h"
13
14
15 /********************************************************************************
16 function    :
17 author      : nc
18 desc        :
19 date        : 03-24-13
20  ********************************************************************************/
21 bool proc_t_init( proc_t** pr )
22 {
23     *pr = malloc( sizeof(**pr) );
24
25     if( !*pr )
26         goto fail;
27
28     memset( *pr, 0, sizeof(**pr) );
29
30     return true;
31
```

```
32  fail:
33      return false;
34  }
35
36
37  /*****************************************************************************
38  function    :
39  author      : nc
40  desc        :
41  date        : 03-24-13
42   *****************************************************************************/
43  void proc_t_destroy( proc_t** pr )
44  {
45      if( *pr )
46          free( *pr );
47
48      *pr = NULL;
49
50      return;
51  }
52
53
54  /*****************************************************************************
55  function    :
56  author      : nc
57  desc        : make a new process
58  date        : 03-24-13
59   *****************************************************************************/
60  bool proc_t_make(
61          proc_t* p,
62          i16 uid,
63          i16 pid,
64          i16 gid,
65          i16 parid,
66          i16 status,
67          proc_t* parent,
68          mino_t* cwd )
69  {
70
71      if( !p || !cwd )
72          goto fail;
73
74      p->m_uid = uid;
75      p->m_pid = pid;
76      p->m_gid = gid;
77      p->m_parid = parid;
78      p->m_status = status;
79
80      if( parent == NULL )
81          p->m_parent = p;
82      else
83          p->m_parent = parent;
84
85      p->m_cwd = cwd;
86
87      return true;
88  fail:
89      printf( "unable to create process %i\n", pid );
90      return false;
91  }
92
93
94
95  /*****************************************************************************
96  function    :
97  author      : nc
98  desc        :
99  date        : 03-24-13
100  *****************************************************************************/
101 i32 proc_t_get_next_fd_loc( proc_t* pr )
102 {
103     int i = 0;
104     for( i = 0; i < NUM_FDS; i++ )
105     {
106         if( pr->m_fds[i] == 0 )
107             break;
108     }
109
110     if( i >= NUM_FDS )
111     {
112         printf( "error: no open file descriptors avaliable\n" );
```

```
113         goto fail;
114     }
115
116     return i;
117
118 fail:
119     return -1;
120 }
121
122
123 /*****************************************************************************
124 function    : bool proc_t_add_fd_to_tb( proc_t* pr, i32 fd )
125 author      : nc
126 desc        :
127 date        : 03-24-13
128  *****************************************************************************/
129 bool proc_t_add_fd_to_tb( proc_t* pr, i32 fd )
130 {
131     int i = 0;
132
133     i = proc_t_get_next_fd_loc( pr );
134
135     if( i == -1 )
136     {
137         goto fail;
138     }
139
140     pr->m_fds[i] = fd;
141
142     return true;
143
144 fail:
145     return false;
146 }
147
148
149
150 /*****************************************************************************
151 function    : bool proc_t_del_fd_from_tb( proc_t* pr, i32 fd )
152 author      : nc
153 desc        :
154 date        : 03-24-13
155  *****************************************************************************/
156 bool proc_t_del_fd_from_tb( proc_t* pr, i32 fd )
157 {
158     if( !pr )
159         goto fail;
160
161     int i = 0;
162
163     for( i = 0; i < NUM_FDS; i++ )
164     {
165         if( pr->m_fds[i] == fd )
166             break;
167     }
168
169     if( i >= NUM_FDS )
170     {
171         printf( "error: can't remove fd, doesn't exist in proct oft\n" );
172         goto fail;
173     }
174
175
176     return true;
177
178 fail:
179     return false;
180
181 }
```

2.8 String functions

2.8.1 Definition

Listing 13: string_funcs.h

```
1 /*****************************************************************************
2 file        : string_funcs.h
```

```
3  author      : nc
4  desc        : misc string functions
5  date        : 03-28-13
6   ********************************************************************************/
7
8  void    strip_nr( char*, unsigned );
9  bool    split_path( char* path, unsigned ar_size, char** dest, int* num );
10 char*   loc_dirname( char* );
11 char*   loc_basename( char* );
12 bool    is_abs_path( char* );
```

2.8.2 Implementation

Listing 14: string_funcs.c

```
1  /********************************************************************************
2  file        : string_funcs.h
3  author      : nc
4  desc        : k_t struct
5  date        : 03-28-13
6   ********************************************************************************/
7
8  #include <stdlib.h>
9  #include <string.h>
10 #include "global_defs.h"
11 #include "string_funcs.h"
12
13
14 /********************************************************************************
15 function    : void strip_nr( char* string, unsigned len )
16 author      : nc
17 desc        : strip \n\r chars from a line
18 date        : 03-28-13
19  ********************************************************************************/
20 void strip_nr( char* string, unsigned len )
21 {
22     unsigned i          = 0;
23     unsigned j          = 0;
24     char local[MAX_TMP] = { 0 };
25
26     /*not gonna have n/r anyway*/
27     if( !string || len > MAX_TMP || len == 0 )
28         return;
29
30     for( i = 0; i < len; i ++ )
31     {
32         while( string[i] == '\n' || string[i] == '\r' )
33             i++;
34
35         if( i > len || !string[i] )
36             break;
37
38         local[i] = string[j];
39
40         j++;
41     }
42
43     /*copy it back over*/
44     memset( string, 0, len );
45     strcpy( string, local );
46     string[strlen(string)] = '\0';
47
48     return;
49
50 }
51
52
53 /********************************************************************************
54 function    : void split_path( char* path, char** ar, int* num )
55 author      : nc
56 desc        : if you call this you need to free the destination array
57 date        :
58  ********************************************************************************/
59 bool split_path( char* path, unsigned ar_size, char** dst, int* num )
60 {
61     int i               = 0;
62     char tmp[MAX_PATH]  = { '\0' };
63     char* tok           = NULL;
64
```

```
65      if( !path || !dst || !num || strlen( path ) > MAX_PATH - 1 || ar_size > MAX_DEPTH )
66          goto fail;
67
68      strcpy( tmp, path );
69
70      /*clear everything in the destination array*/
71      for( i = 0; i < ar_size; i ++ )
72      {
73          if( dst[i] )
74          {
75              free( dst[i] );
76              dst[i] = NULL;
77          }
78      }
79
80      i = 0;
81
82      /*tokenize the path*/
83      tok = strtok( tmp, DIR_TOK );
84
85      while( tok )
86      {
87          dst[i] = malloc( strlen( tok ) + 1 );
88
89          if( !dst[i] )
90              goto fail;
91
92          memset( dst[i], 0, strlen( tok ) + 1 );
93          strcpy( dst[i], tok );
94
95          i++;
96
97          tok = strtok( NULL, DIR_TOK );
98      }
99
100     /*set the number of path elements*/
101     *num = i;
102
103     return true;;
104
105 fail:
106
107     /*try to clean up*/
108     for( i = 0; i < ar_size; i ++ )
109     {
110         if( dst[i] )
111         {
112             free( dst[i] );
113             dst[i] = NULL;
114         }
115     }
116
117     return false;
118 }
119
120
121 /*****************************************************************************
122 function    : char* loc_dirname( char* path )
123 author      : nc
124 desc        :
125 date        : modified version of bsd dirname()
126  *****************************************************************************/
127 char* loc_dirname( char* path_in )
128 {
129     static char result[MAX_PATH] = { '\0' };
130     char path[MAX_PATH] = { '\0' };
131     char* end       = NULL;
132     unsigned len    = 0;
133
134     /*don't want to modify input*/
135     if( strlen( path_in ) + 1 > MAX_PATH )
136         return NULL;
137
138     memcpy( path, path_in, strlen( path_in ) );
139
140     /*check for cwd*/
141     if( ( path == NULL ) || ( *path == '\0' ) )
142         goto cwd;
143
144     /*remove trailing slashes*/
145     end = path + strlen( path ) - 1;
```

```
146     while( end != path && *end == '/' )
147         end--;
148 
149     while( --end >= path )
150     {
151         if( *end == '/' )
152         {
153             /*remove trailing slashes again*/
154             while( end != path && *end == '/' )
155                 end--;
156 
157             /*copy result*/
158             len = (end - path) + 1;
159             if( len > ( MAX_PATH - 1 ) )
160                 len = MAX_PATH - 1;
161 
162             memcpy( result, path, len );
163             result[len] = '\0';
164 
165             return result;
166         }
167     }
168 
169 cwd:
170     result[0] = '.';
171     result[1] = '\0';
172 
173     return result;
174 }
175 
176 
177 /*****************************************************************************
178 function    : char* loc_dirname( char* path )
179 author      : nc
180 desc        :
181 date        : 03-30-13
182  *****************************************************************************/
183 char* loc_basename( char* path_in )
184 {
185     static char bname[MAX_PATH] = { '\0' };
186     char path[MAX_PATH]         = { '\0' };
187     char* end       = NULL;
188     char* start     = NULL;
189 
190     /*don't want to modify input*/
191     if( strlen( path_in ) + 1 > MAX_PATH )
192         return NULL;
193 
194     memcpy( path, path_in, strlen( path_in ) );
195 
196     /*check for cwd*/
197     if( !path_in || *path == '\0')
198     {
199         strcpy( bname, "." );
200         return bname;
201     }
202 
203     /*remove trailing slashes*/
204     end = path + strlen( path ) - 1;
205 
206     while( end > path && *end == '/' )
207         end--;
208 
209     /*root dir*/
210     if( end == path && *end == '/' )
211     {
212         strcpy( bname, "/" );
213         return bname;
214     }
215 
216     /*find the start of the base*/
217     start = end;
218     while( start > path && *( start - 1 ) != '/')
219         start--;
220 
221     if( end - start + 2 > MAX_PATH )
222         return NULL;
223 
224     strncpy( bname, start, end - start + 1 );
225     bname[end - start + 1] = '\0';
226     return bname;
```

```
227 }
228
229
230
231 /*****************************************************************************
232 function    : bool is_abs_path( char* path )
233 author      : nc
234 desc        : check and see if a path is absolute or not
235 date        : 03-30-13
236  *****************************************************************************/
237 bool is_abs_path( char* path )
238 {
239     if( !path || path[0] != '/' )
240         return false;
241
242     if( path[0] == '/' )
243         return true;
244
245     /*???*/
246     return false;
247 }
```

2.9 Virtual shell

2.9.1 Definition

Listing 15: vsh_t.h

```
1  /*****************************************************************************
2  file        : vsh_t.h
3  author      : nc
4  desc        : vsh_t struct
5  date        : 03-25-13
6   *****************************************************************************/
7
8  #pragma once
9
10 #include "global_defs.h"
11 #include "proc_t.h"
12
13
14 /*defines*/
15 #define MAX_LINE    1024    //maximum number of input chars
16 #define MAX_ARGS    16      //maximum number of different args
17 #define TOK_STR     " "     //token for strtok
18
19 #define QUIT_STR    "q"     //exit command
20
21 /*fw dec*/
22 typedef struct vsh_t vsh_t;
23
24
25 /*struct*/
26 struct vsh_t
27 {
28     char    m_line[MAX_LINE];           //input line
29     char*   m_argv[MAX_ARGS];           //arguments passed in
30     int     m_argc;                     //number of args in m_argv
31     proc_t* m_proc;
32     int     (*m_ftx)( proc_t*, int, char** );
33 };
34
35 bool    vsh_t_init( vsh_t** );          //initialize
36 void    vsh_t_destroy( vsh_t** );       //shutdown
37 int     vsh_t_run( vsh_t*, proc_t* );   //main loop
38 int     vsh_t_get_cmd( vsh_t* );        //get a command from the user
39 int     vsh_t_run_cmd( vsh_t* );        //run the command
```

2.9.2 Implementation

Listing 16: vsh_t.c

```
1  /*****************************************************************************
2  file        : vsh_t.c
3  author      : nc
4  desc        : implementation of the vsh_t functions
```

```
5  date        : 03-30-13
6   ********************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include "global_defs.h"
12 #include "vsh_t.h"
13 #include "string_funcs.h"
14 #include "func_addresses.h"
15 #include "k_t.h"
16
17 /*globals (remove*/
18 extern k_t* kr;
19
20
21
22 /*********************************************************************************
23 function    : bool vsh_t_init( vsh_t** sh )
24 author      : nc
25 desc        : create the shell
26 date        : 03-24-13
27  ********************************************************************************/
28 bool vsh_t_init( vsh_t** sh )
29 {
30     *sh = malloc( sizeof(**sh) );
31
32     if( !*sh )
33         goto fail;
34
35     memset( *sh, 0, sizeof(**sh) );
36
37     return true;
38
39 fail:
40     exit( X_NO_SHELL );
41     return false;
42 }
43
44
45
46 /*********************************************************************************
47 function    : void vsh_t_destroy( vsh_t** sh )
48 author      : nc
49 desc        : destroy the shell
50 date        : 03-30-13
51  ********************************************************************************/
52 void vsh_t_destroy( vsh_t** sh )
53 {
54     int i = 0;
55     if( !*sh )
56         return;
57
58     /*free eveything in the arg array*/
59     for( i = 0; i < MAX_ARGS; i++ )
60     {
61         if( (*sh)->m_argv[i] )
62         {
63             free( (*sh)->m_argv[i] );
64             (*sh)->m_argv[i] = NULL;
65         }
66     }
67
68     if( *sh )
69         free( *sh );
70
71     *sh = NULL;
72
73     return;
74 }
75
76
77
78 /*********************************************************************************
79 function    : int vsh_t_run( void )
80 author      : nc
81 desc        : main loop for the shell
82 date        : 03-30-13
83  ********************************************************************************/
84 int vsh_t_run( vsh_t* sh, proc_t* pr )
85 {
```

```
86      if( !sh || !pr )
87          goto no_sh;
88
89      sh->m_proc = pr;
90
91      while( true )
92      {
93          vsh_t_get_cmd( sh );
94
95          if( vsh_t_run_cmd( sh ) == X_DONE )
96              break;
97
98      }
99
100     return X_SUCCESS;
101
102 no_sh:
103     printf( "lost shell\n" );
104     return X_NO_SHELL;
105
106 fail_unk:
107     return X_UNKNOWN;
108
109 }
110
111
112
113 /*****************************************************************************
114 function    : int vsh_t_get_cmd( void )
115 author      : nc
116 desc        : split a user command up
117 date        : 03-30-13
118  *****************************************************************************/
119 int vsh_t_get_cmd( vsh_t* sh )
120 {
121     int i               = 0;
122     char tmp[MAX_LINE]  = { '\0' };
123     char* tok           = NULL;
124
125     if( !sh )
126         goto no_sh;
127
128     /*delete some stuff*/
129     memset( sh->m_line, 0, MAX_LINE );
130
131     for( i = 0; i < MAX_ARGS; i++ )
132     {
133         if( sh->m_argv[i] )
134         {
135             free( sh->m_argv[i] );
136             sh->m_argv[i] = NULL;
137         }
138     }
139
140     sh->m_ftx = NULL;
141
142     sh->m_argc = 0;
143
144     /*get some stuff*/
145     if( kr->m_cproc )
146     {
147        printf( "proc %i$ ", kr->m_cproc->m_pid );
148     }
149     else
150     {
151         printf( "np$ " );
152     }
153     fgets( sh->m_line, MAX_LINE, stdin );
154
155     strip_nr( sh->m_line, strlen( sh->m_line ) );
156     memcpy( tmp, sh->m_line, MAX_LINE );
157
158     /*find out the arg count*/
159     i = 0;
160     tok = strtok( tmp, TOK_STR );
161
162     /*split up the line*/
163     while( tok )
164     {
165         /*set up argv*/
166         sh->m_argv[i] = malloc( strlen( tok ) + 1 );
```

```
167
168         if( !sh->m_argv[i] )
169             goto no_mem;
170
171         memset( sh->m_argv[i], 0, strlen( tok ) + 1 );
172         strcpy( sh->m_argv[i], tok );
173
174         tok = strtok( NULL, TOK_STR );
175         i++;
176     }
177
178     sh->m_argc = i;
179
180     return X_SUCCESS;
181 no_sh:
182     return X_NO_SHELL;
183 no_mem:
184     return X_NO_MEMORY;
185 fail:
186     return X_CRITICAL;
187
188 };
189
190
191
192 /*****************************************************************************
193 function    : int vsh_t_run( vsh_t* sh )
194 author      : nc
195 desc        : run the command we got
196 date        : 03-30-13
197  *****************************************************************************/
198 int vsh_t_run_cmd( vsh_t* sh )
199 {
200     if( !sh )
201         goto fail;
202
203     /*first check for no command*/
204     if( !sh->m_argc || !sh->m_argv[0] )
205         goto succeed;
206
207     /*check for exit*/
208     if( !strcmp( sh->m_argv[0], QUIT_STR ) )
209         goto done;
210
211     /*otherwise try to execute a command*/
212     sh->m_ftx = get_fp( sh->m_argv[0] );
213
214     if( sh->m_ftx == NULL )
215         goto not_found;
216
217     sh->m_ftx( sh->m_proc, sh->m_argc, sh->m_argv );
218     goto succeed;
219
220 not_found:
221     printf( "command not found\n" );
222     return X_SUCCESS;
223 succeed:
224     return X_SUCCESS;
225 done:
226     return X_DONE;
227 fail:
228     return X_CRITICAL;
229 }
```

2.10 Function addresses

2.10.1 Definition

Listing 17: func_addresses.h

```
1  /*****************************************************************************
2  file        : func_addresses.h
3  author      : nc
4  desc        : adrresses for function pointers
5  date        : 04-07-13
6   *****************************************************************************/
7
8  #pragma once
```

```
9
10 #include "l0/show_help.h"
11 #include "l0/test_func.h"
12 #include "l0/print_minos.h"
13 #include "l0/print_vfs_mounts.h"
14 #include "l0/show_imap.h"
15 #include "l0/show_bmap.h"
16 #include "l0/print_ino.h"
17 #include "l0/print_procs.h"
18 #include "l0/switch.h"
19
20 #include "l1/ls.h"
21 #include "l1/cd.h"
22 #include "l1/creat.h"
23 #include "l1/mkdir.h"
24 #include "l1/pwd.h"
25 #include "l1/link.h"
26 #include "l1/unlink.h"
27 #include "l1/rmdir.h"
28 #include "l1/symblink.h"
29 #include "l1/stat.h"
30 #include "l1/touch.h"
31 #include "l1/chown.h"
32 #include "l1/chmod.h"
33 #include "l1/chgrp.h"
34
35 #include "l2/open.h"
36 #include "l2/close.h"
37 #include "l2/lseek.h"
38 #include "l2/pfd.h"
39 #include "l2/read.h"
40 #include "l2/cat.h"
41 #include "l2/write.h"
42 #include "l2/cp.h"
43 #include "l2/mv.h"
44
45 #include "global_defs.h"
46
47
48 typedef int (*fp_tb)( proc_t*, int, char** );
49
50 fp_tb get_fp( char* );
```

2.10.2 Implementation

Listing 18: func_addresses.c

```
1  /*****************************************************************************
2  file        : func_addresses.c
3  author      : nc
4  desc        : adrresses for function pointers
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <string.h>
10 #include "global_defs.h"
11 #include "func_addresses.h"
12
13
14 fp_tb get_fp( char* string )
15 {
16     /*level 0*/
17     if( !strcmp( string, "help" ) )
18         return show_help;
19     if( !strcmp( string, "test" ) )
20         return test_cmd;
21     if( !strcmp( string, "pminos" ) )
22         return print_minos;
23     if( !strcmp( string, "pvfs" ) )
24         return print_vfs_mounts;
25     if( !strcmp( string, "show_imap" ) )
26         return show_imap;
27     if( !strcmp( string, "show_bmap" ) )
28         return show_bmap;
29     if( !strcmp( string, "pino" ) )
30         return print_ino;
31     if( !strcmp( string, "pprocs" ) )
32         return print_procs;
```

```
33      if( !strcmp( string, "switch" ) )
34          return switch_cmd;
35
36      /*level 1*/
37      if( !strcmp( string, "ls" ) )
38          return ls_cmd;
39      if( !strcmp( string, "cd" ) )
40          return cd_cmd;
41      if( !strcmp( string, "creat" ) )
42          return creat_cmd;
43      if( !strcmp( string, "mkdir" ) )
44          return mkdir_cmd;
45      if( !strcmp( string, "pwd" ) )
46          return pwd_cmd;
47      if( !strcmp( string, "link" ) )
48          return link_cmd;
49      if( !strcmp( string, "unlink" ) )
50          return unlink_cmd;
51      if( !strcmp( string, "rmdir" ) )
52          return rmdir_cmd;
53      if( !strcmp( string, "symlink") )
54          return symblink_cmd;
55      if( !strcmp( string, "stat") )
56          return stat_cmd;
57      if( !strcmp( string, "touch") )
58          return touch_cmd;
59      if( !strcmp( string, "chown") )
60          return chown_cmd;
61      if( !strcmp( string, "chmod") )
62          return chmod_cmd;
63      if( !strcmp( string, "chgrp") )
64          return chgrp_cmd;
65
66      /*level 2*/
67      if( !strcmp( string, "open" ) )
68          return open_cmd;
69      if( !strcmp( string, "close" ) )
70          return close_cmd;
71      if( !strcmp( string, "lseek" ) )
72          return lseek_cmd;
73      if( !strcmp( string, "pfd" ) )
74          return pfd_cmd;
75      if( !strcmp( string, "read" ) )
76          return read_cmd;
77      if( !strcmp( string, "cat" ) )
78          return cat_cmd;
79      if( !strcmp( string, "write") )
80          return write_cmd;
81      if( !strcmp( string, "cp") )
82          return cp_cmd;
83
84      return NULL;
85  }
```

3 Level zero

3.1 Print inodes

3.1.1 Definition

Listing 19: print_ino.h

```
1  /*************************************************************************
2  file        : print_ino.h
3  author      : nc
4  desc        : prints out a specific in-memory inode
5  date        : 04-01-13
6   *************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int print_ino( proc_t*, int argc, char** argv );
```

3.1.2 Implementation

Listing 20: print_ino.c

```
1  /*****************************************************************************
2  file        : print_ino.c
3  author      : nc
4  desc        : prints out the in-memory inodes
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10
11 #include "../global_defs.h"
12 #include "../k_t.h"
13 #include "../mino_t.h"
14 #include "../proc_t.h"
15 #include "print_ino.h"
16
17 extern k_t* kr;
18
19 int print_ino( proc_t* pr, int argc, char** argv )
20 {
21     int i       = 0;
22     int j       = 1;
23     mino_t* tmp = NULL;
24
25     if( !kr )
26         goto fail;
27
28     if( argc < 2 )
29     {
30         printf( "usage: pino [list of inodes]\n" );
31         goto fail;
32     }
33
34     printf( "inode info:\n" );
35     printf( "\tmode\tuid\tsize\tgid\tlinks\tblocks\tblock (direct)\n" );
36
37     for( i = 1; i < argc; i ++ )
38     {
39         k_t_get_mino( pr->m_cwd->m_fd, atoi( argv[i] ), &tmp );
40
41         if( !tmp )
42         {
43             printf( "\terror retrieveing inode\n" );
44             goto fail;
45         }
46
47         printf( "\t%x\t%i\t%i\t%i\t%i\t%i\t",
48                 tmp->m_ino.i_mode,
49                 tmp->m_ino.i_uid,
50                 tmp->m_ino.i_size,
51                 tmp->m_ino.i_gid,
52                 tmp->m_ino.i_links_count,
53                 tmp->m_ino.i_blocks );
54
55         for( j = 0; j < MAX_ENT; j ++ )
56         {
57             printf( "%i ", tmp->m_ino.i_block[j] );
58         }
59
60         printf( "\n" );
61
62         k_t_put_mino( atoi( argv[i] ) );
63
64     }
65
66
67 ok:
68     return true;
69
70 fail:
71     return false;
72 }
```

3.2 Print in-memory inodes

3.2.1 Definition

Listing 21: print_minos.h

```
1  /*****************************************************************************
2  file        : print_minos.h
3  author      : nc
4  desc        : prints out a specific in-memory inode
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int print_minos( proc_t*, int argc, char** argv );
```

3.2.2 Implementation

Listing 22: print_minos.c

```
1  /*****************************************************************************
2  file        : print_minos.c
3  author      : nc
4  desc        : prints out the in-memory inodes
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10
11 #include "../global_defs.h"
12 #include "../k_t.h"
13 #include "../mino_t.h"
14 #include "../proc_t.h"
15 #include "print_minos.h"
16
17 extern k_t* kr;
18
19 int print_minos( proc_t* pr, int argc, char** argv )
20 {
21     int i = 0;
22     int j = 1;
23
24     if( !kr )
25         goto fail;
26
27     /*print all the loaded in-memory inodes*/
28     printf( "memory inode information:\n" );
29     printf( "\toffset\t\tinode num\tfd\t\trefs\t\tdirty\t\tmounted\t\tmount point\n" );
30
31     /*if we were given a list*/
32     if( argc > 1 )
33     {
34         for( j = 1; j < argc; j++ )
35         {
36             i = atoi( argv[j] );
37             if( kr->m_mino_tb[i] )
38             {
39                 printf(
40                         "\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%p\n",
41                         i,
42                         kr->m_mino_tb[i]->m_ino_num,
43                         kr->m_mino_tb[i]->m_fd,
44                         kr->m_mino_tb[i]->m_refc,
45                         kr->m_mino_tb[i]->m_dirty,
46                         kr->m_mino_tb[i]->m_mounted,
47                         kr->m_mino_tb[i]->m_mountp
48                       );
49             }
50             else
51             {
52                 printf( "\tinvalid entry %i\n", i );
53             }
54         }
55         goto ok;
```

```
56      }
57
58      /*otherwise print them all*/
59      for( i = 0; i < MAX_MINOS && kr->m_mino_tb[i]; i++ )
60      {
61          printf(
62                  "\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%p\n",
63                  i,
64                  kr->m_mino_tb[i]->m_ino_num,
65                  kr->m_mino_tb[i]->m_fd,
66                  kr->m_mino_tb[i]->m_refc,
67                  kr->m_mino_tb[i]->m_dirty,
68                  kr->m_mino_tb[i]->m_mounted,
69                  kr->m_mino_tb[i]->m_mountp
70                );
71      }
72
73      printf( "\n" );
74
75  ok:
76      return true;
77
78  fail:
79      return false;
80  }
```

3.3 Print processes

3.3.1 Definition

Listing 23: print_procs.h

```
1   /*****************************************************************************
2   file        : print_proc.h
3   author      : nc
4   desc        : prints out a specific in-memory inode
5   date        : 04-01-13
6    *****************************************************************************/
7   #pragma once
8
9   #include "../global_defs.h"
10  #include "../proc_t.h"
11
12  int print_procs( proc_t*, int argc, char** argv );
```

3.3.2 Implementation

Listing 24: print_procs.c

```
1   /*****************************************************************************
2   file        : print_proc.c
3   author      : nc
4   desc        : prints out the in-memory inodes
5   date        : 04-01-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <stdio.h>
10
11  #include "../global_defs.h"
12  #include "../k_t.h"
13  #include "../mino_t.h"
14  #include "../proc_t.h"
15  #include "print_procs.h"
16
17  extern k_t* kr;
18
19  int print_procs( proc_t* pr, int argc, char** argv )
20  {
21      int i       = 0;
22
23      if( !kr || !argc )
24          goto fail;
25
26      printf( "process information\n" );
27      printf( "\tuid\tpid\tgid\tparid\tstatus\n" );
```

```
28
29      for( i = 0; i < MAX_PROCS; i++ )
30      {
31          if( kr->m_proc_tb[i] )
32          {
33              printf( "\t%i\t%i\t%i\t%i\t%i\n",
34                      kr->m_proc_tb[i]->m_uid,
35                      kr->m_proc_tb[i]->m_pid,
36                      kr->m_proc_tb[i]->m_gid,
37                      kr->m_proc_tb[i]->m_parid,
38                      kr->m_proc_tb[i]->m_status );
39          }
40      }
41
42      printf( "\n" );
43
44
45  ok:
46      return true;
47
48  fail:
49      return false;
50  }
```

3.4 Print VFS mounts

3.4.1 Definition

Listing 25: print_vfs_mounts.h

```
1  /*****************************************************************************
2  file        : print_vfs_mounts.c
3  author      : nc
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #pragma once
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int print_vfs_mounts( proc_t*, int argc, char** argv );
```

3.4.2 Implementation

Listing 26: print_vsf_mounts.c

```
1  /*****************************************************************************
2  file        : print_vfs_mounts.c
3  author      : nc
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10
11 #include "print_vfs_mounts.h"
12
13 #include "../global_defs.h"
14 #include "../k_t.h"
15 #include "../mnt_t.h"
16 #include "../proc_t.h"
17
18 extern k_t* kr;
19
20 int print_vfs_mounts( proc_t* pr, int argc, char** argv )
21 {
22     int i = 0;
23
24     if( !kr )
25         goto fail;
26
27     printf( "vfs mount information:\n" );
28     printf( "\tinodes\t\tblocks\t\tbusy\t\troot inode\tdev name\tmount name\n" );
29
```

```
30      for( i = 0; i < MAX_MNTS && kr->m_mnt_tb[i]; i++ )
31      {
32          printf(
33                  "\t%i\t\t%i\t\t%i\t\t%p\t%s\t\t%s\n",
34                  kr->m_mnt_tb[i]->m_num_inos,
35                  kr->m_mnt_tb[i]->m_num_blocks,
36                  kr->m_mnt_tb[i]->m_busy,
37                  kr->m_mnt_tb[i]->m_root_ino,
38                  kr->m_mnt_tb[i]->m_dev_name,
39                  kr->m_mnt_tb[i]->m_mnt_name
40                );
41
42          printf( "\n" );
43      }
44
45  ok:
46      return true;
47
48  fail:
49      return false;
50  }
```

3.5 Show inode bitmap

3.5.1 Definition

Listing 27: show_imap.h

```
1   /*****************************************************************************
2   file        : show_imap.h
3   author      : nc
4   desc        : print out the inode bitmap
5   date        : 04-07-13
6    *****************************************************************************/
7   #pragma once
8
9   #include "../global_defs.h"
10  #include "../proc_t.h"
11
12  int show_imap( proc_t*, int argc, char** argv );
```

3.5.2 Implementation

Listing 28: show_imap.c

```
1   /*****************************************************************************
2   file        : show_imap.c
3   author      : nc
4   desc        : print out the inode bitmap
5   date        : 04-07-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <stdio.h>
10
11  #include "../global_defs.h"
12  #include "../k_t.h"
13  #include "../mino_t.h"
14  #include "../proc_t.h"
15  #include "../mnt_t.h"
16  #include "show_imap.h"
17
18  extern k_t* kr;
19
20  int show_imap( proc_t* pr, int argc, char** argv )
21  {
22      u32 cnt_inodes          = 0;
23      int i                   = 0;
24      int j                   = 0;
25      bool ret                = 0;
26      mnt_t* current_mount    = NULL;
27
28      /*get the current mount point*/
29      current_mount = k_t_get_mnt_from_fd( pr->m_cwd->m_fd );
30
31      if( !current_mount )
```

```
32          goto fail;
33
34      mnt_t_lock( current_mount );
35
36      if( argc < 2 )
37      {
38
39          printf( "inode bitmap for mount point %s (%s):\n\n",
40                  current_mount->m_mnt_name,
41                  current_mount->m_dev_name );
42
43          printf( "\t[0-7]\t\t\t  [8-15]" );
44          for( i = 0; i < current_mount->m_sb.s_inodes_count; i ++ )
45          {
46              if( i % 16 == 0 )
47              {
48                  j = 0;
49                  printf( "\n%i:\t", i );
50              }
51
52              if( j > 1 && j % 8 == 0 )
53              {
54                  printf( "  " );
55              }
56
57              k_t_get_imap_bit( pr->m_cwd->m_fd, i, &ret );
58              printf( "%d  ", ret );
59
60              j++;
61
62              /*increment our count*/
63              if( ret )
64                  cnt_inodes++;
65          }
66
67          printf( "\n\n" );
68
69          printf( "superblock reports:\t%i inodes, %i free and %i used\n",
70                  current_mount->m_sb.s_inodes_count,
71                  current_mount->m_sb.s_free_inodes_count,
72                  current_mount->m_sb.s_inodes_count - current_mount->m_sb.s_free_inodes_count );
73
74          printf( "counted:\t\t%i inodes, %i free and %i used\n",
75                  current_mount->m_sb.s_inodes_count,
76                  current_mount->m_sb.s_inodes_count - cnt_inodes,
77                  cnt_inodes );
78      }
79      else
80      {
81          if( atoi( argv[1] ) > current_mount->m_sb.s_inodes_count )
82          {
83              printf( "error, value too large - maximum inode value is %i\n",
84                      current_mount->m_sb.s_inodes_count );
85              goto ok;
86          }
87
88          k_t_get_imap_bit( pr->m_cwd->m_fd, atoi( argv[1] ), &ret );
89
90          printf( "inode %i is set to %i\n",
91                  atoi( argv[1] ),
92                  ret );
93      }
94
95
96      mnt_t_unlock( current_mount );
97  ok:
98      return true;
99
100 fail:
101     printf( "error retrieving imap\n") ;
102     mnt_t_unlock( current_mount );
103     return false;
104 }
```

3.6 Show data block bitmap

3.6.1 Definition

Listing 29: show_bmap.h

```
/*********************************************************************************
file        : show_bmap.h
author      : nc
desc        : print out the data block bitmap
date        : 04-07-13
 *********************************************************************************/
#pragma once

#include "../global_defs.h"
#include "../proc_t.h"

int show_bmap( proc_t*, int argc, char** argv );
```

3.6.2 Implementation

Listing 30: show_bmap.c

```
/*********************************************************************************
file        : how_bmap.c
author      : nc
desc        : print out the data block bitmap
date        : 04-07-13
 *********************************************************************************/

#include <stdlib.h>
#include <stdio.h>

#include "../global_defs.h"
#include "../k_t.h"
#include "../mino_t.h"
#include "../proc_t.h"
#include "../mnt_t.h"
#include "show_imap.h"

extern k_t* kr;

int show_bmap( proc_t* pr, int argc, char** argv )
{
    u32 cnt_dblocks         = 0;
    int i                   = 0;
    int j                   = 0;
    bool ret                = 0;
    mnt_t* current_mount    = NULL;

    /*get the current mount point*/
    current_mount = k_t_get_mnt_from_fd( pr->m_cwd->m_fd );

    if( !current_mount )
        goto fail;

    mnt_t_lock( current_mount );

    if( argc < 2 )
    {

        printf( "data block bitmap for mount point %s (%s):\n\n",
                current_mount->m_mnt_name,
                current_mount->m_dev_name );

        printf( "\t[0-7]\t\t\t  [8-15]\t\t    [16-23]\t\t      [24-31]" );

        for( i = 0; i < current_mount->m_sb.s_blocks_count; i ++ )
        {
            if( i % 32 == 0 )
            {
                j = 0;
                printf( "\n%i:\t", i );
            }

            if( j > 1 && j % 8 == 0 )
            {
                printf( "  " );
            }

            k_t_get_bmap_bit( pr->m_cwd->m_fd, i, &ret );
            printf( "%d  ", ret );

```

```
61              j++;
62
63              /*increment our count*/
64              if( ret )
65                  cnt_dblocks++;
66          }
67
68          printf( "\n\n" );
69
70          printf( "superblock reports:\t%i dblocks, %i free and %i used\n",
71                  current_mount->m_sb.s_blocks_count,
72                  current_mount->m_sb.s_free_blocks_count,
73                  current_mount->m_sb.s_blocks_count - current_mount->m_sb.s_free_blocks_count );
74
75          printf( "counted:\t\t%i dblocks, %i free and %i used\n",
76                  current_mount->m_sb.s_blocks_count,
77                  current_mount->m_sb.s_blocks_count - cnt_dblocks,
78                  cnt_dblocks );
79      }
80      else
81      {
82          if( atoi( argv[1] ) > current_mount->m_sb.s_blocks_count )
83          {
84              printf( "error, value too large - maximum dblock value is %i\n",
85                      current_mount->m_sb.s_blocks_count );
86              goto ok;
87          }
88
89          k_t_get_bmap_bit( pr->m_cwd->m_fd, atoi( argv[1] ), &ret );
90
91          printf( "data block %i is set to %i\n",
92                  atoi( argv[1] ),
93                  ret );
94      }
95
96  ok:
97      mnt_t_unlock( current_mount );
98      return true;
99
100 fail:
101     printf( "error retrieving dmap\n") ;
102     mnt_t_unlock( current_mount );
103     return false;
104 }
```

3.7 Switch processes

3.7.1 Definition

Listing 31: switch.h

```
1  /*****************************************************************************
2  file        : print_proc.h
3  author      : nc
4  desc        : prints out a specific in-memory inode
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int switch_cmd( proc_t*, int argc, char** argv );
```

3.7.2 Implementation

Listing 32: switch.c

```
1  /*****************************************************************************
2  file        : print_proc.c
3  author      : nc
4  desc        : prints out the in-memory inodes
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
```

```
9  #include <stdio.h>
10
11 #include "../global_defs.h"
12 #include "../k_t.h"
13 #include "../mino_t.h"
14 #include "../proc_t.h"
15 #include "switch.h"
16
17 extern k_t* kr;
18
19 int switch_cmd( proc_t* pr, int argc, char** argv )
20 {
21     int i = 0;
22
23     if( argc != 2 )
24     {
25         printf( "usage: switch [proc number]\n" );
26         goto fail;
27     }
28
29     for( i = 0; i < MAX_PROCS; i++ )
30     {
31         if( kr->m_proc_tb[i] && kr->m_proc_tb[i]->m_pid == atoi(argv[1]) )
32         {
33             kr->m_cproc = kr->m_proc_tb[i];
34             break;
35         }
36     }
37
38     if( i >= MAX_PROCS )
39     {
40         printf( "error: process does not exist\n" );
41         goto fail;
42     }
43
44     return 0;
45
46 fail:
47     return -1;
48 }
```

3.8 Testing function

3.8.1 Definition

Listing 33: test_func.h

```
1  /*********************************************************************************
2  file        : test_func.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *********************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int test_cmd( proc_t*, int argc, char** argv );
```

3.8.2 Implementation

Listing 34: test_func.c

```
1  /*********************************************************************************
2  file        : test_func.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *********************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "test_func.h"
```

```
13  #include "../global_defs.h"
14  #include "../k_t.h"
15  #include "../mino_t.h"
16  #include "../proc_t.h"
17
18
19  extern k_t* kr;
20
21  int test_cmd( proc_t* pr, int argc, char** argv )
22  {
23      //bool i            = 0;
24      //mnt_t* mt     = NULL;
25      //inode new_ino = { 0 };
26      //u32 loc           = 0;
27
28      if( !kr )
29          goto fail;
30
31      printf( "returned: %i\n", k_t_get_next_imap_loc( pr->m_cwd->m_fd ) );
32      printf( "returned: %i\n", k_t_get_next_bmap_loc( pr->m_cwd->m_fd ) );
33      k_t_flush_clean_minos();
34
35      if( argc > 3 && !strcmp( argv[1], "add" ) )
36      {
37          k_t_add_dblocks_to_mino( pr->m_cwd->m_fd, atoi( argv[2] ), atoi( argv[3] ) );
38          printf( "ok\n" );
39      }
40
41      if( argc > 3 && !strcmp( argv[1], "del" ) )
42      {
43          k_t_del_dblocks_from_mino( pr->m_cwd->m_fd, atoi( argv[2] ), atoi( argv[3] ) );
44          printf( "ok\n" );
45      }
46
47      return true;
48
49  fail:
50      return false;
51  }
```

4 Level one

4.1 Change directory

4.1.1 Definition

Listing 35: cd.h

```
1  /*****************************************************************************
2  file        : cd.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int cd_cmd( proc_t*, int argc, char** argv );
```

4.1.2 Implementation

Listing 36: cd.c

```
1  /*****************************************************************************
2  file        : cd.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
```

```
10  #include <string.h>
11
12  #include "cd.h"
13  #include "../string_funcs.h"
14  #include "../global_defs.h"
15  #include "../k_t.h"
16  #include "../mino_t.h"
17  #include "../proc_t.h"
18
19
20  extern k_t* kr;
21
22  int cd_cmd( proc_t* pr, int argc, char** argv )
23  {
24      char* split[MAX_DEPTH]  = { '\0' }; //split path elements
25      int to_free[MAX_DEPTH]  = { 0 };    //array of minos to put back
26      int num_splits          = 0;
27      int i                   = 0;
28      int inum                = 0;
29      mino_t* current         = NULL;     //current directory
30
31      if( !kr )
32          goto fail;
33
34      if( argc > 1 )
35      {
36
37          if( is_abs_path( argv[1] ) )
38          {
39              current = kr->m_mino_tb[0];
40          }
41          else
42          {
43              current = pr->m_cwd;
44          }
45
46          if( !current )
47          {
48              printf( "error: couldn't find starting directory\n" );
49              goto fail;
50          }
51
52
53          if(argc == 2 )
54          {
55              if( !split_path( argv[1], MAX_DEPTH, split, &num_splits ) )
56              {
57                  printf( "error splitting path\n" );
58                  goto fail;
59              }
60
61          }
62
63          /*walk through the directories*/
64          for( i = 0; i < num_splits; i++ )
65          {
66              /*find the inode number for the current directory*/
67              inum = k_t_find_dir_num( current, split[i] );
68              if( inum != -1 )
69              {
70                  /*get the next inode*/
71                  to_free[i] = inum;
72                  k_t_get_mino( current->m_fd, inum, &current );
73              }
74              else
75              {
76                  printf( "could not find directory %s\n", split[i] );
77                  goto fail;
78              }
79          }
80      }
81      else
82      {
83          current = kr->m_mino_tb[0];
84      }
85
86      pr->m_cwd = current;
87
88      /*put everything back*/
89      for( i = 0; i < num_splits; i++ )
90      {
```

```
91          if( to_free[i] )
92              k_t_put_mino( to_free[i] );
93      }
94
95      /*free the split*/
96      for( i = 0; i < num_splits; i++ )
97      {
98          if( split[i] )
99              free( split[i] );
100     }
101
102     return 0;
103
104 fail:
105
106     /*free the split*/
107     for( i = 0; i < num_splits; i++ )
108     {
109         if( split[i] )
110             free( split[i] );
111     }
112
113
114     return -1;
115 }
```

4.2 Change group

4.2.1 Definition

Listing 37: chgrp.h

```
1  /*****************************************************************************
2  file        : chgrp.h
3  author      : jk
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int chgrp_cmd( proc_t*, int argc, char** argv );
```

4.2.2 Implementation

Listing 38: chgrp.c

```
1  /*****************************************************************************
2  file        : chgrp.c
3  author      :
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <sys/stat.h>
13 #include <sys/types.h>
14
15 #include "chgrp.h"
16 #include "../string_funcs.h"
17 #include "../global_defs.h"
18 #include "../k_t.h"
19 #include "../mino_t.h"
20 #include "../proc_t.h"
21
22
23 extern k_t* kr;
24
25
26 int chgrp_cmd( proc_t* pr, int argc, char** argv )
27 {
```

```
28      int i                   = 0;
29      int inum                = 0;
30      int group               = 0;
31      int num_splits          = 0;
32      int to_free[MAX_DEPTH]  = { 0 };
33      char* split[MAX_DEPTH]  = { 0 };
34      mino_t* current         = NULL;
35
36      if(argc < 3)
37      {
38          printf("Usage: touch [pathname] [Group ID]\n");
39          goto fail;
40      }
41
42      if( is_abs_path( argv[1] ) )
43      {
44          current = kr->m_mino_tb[0];
45      }
46      else
47      {
48          current = pr->m_cwd;
49      }
50
51      /*current and dst now point at the starting directories*/
52      if( !current)
53      {
54          printf( "error: couldn't find starting directory\n" );
55          goto fail;
56      }
57
58      /*split the first path*/
59      if( !split_path( argv[1] , MAX_DEPTH, split, &num_splits ) )
60      {
61          printf( "error splitting path\n" );
62          goto fail;
63      }
64
65      /*walk through the directories*/
66      for( i = 0; i < num_splits; i++ )
67      {
68          /*find the inode number for the current directory*/
69          inum = k_t_find_dir_num( current, split[i] );
70          if( inum != -1 )
71          {
72              /*get the next inode*/
73              to_free[i] = inum;
74              k_t_get_mino( current->m_fd, inum, &current );
75          }
76          else
77          {
78              printf( "could not find directory %s\n", split[i] );
79              goto fail;
80          }
81      }
82
83      /*if the file/dir can't be found*/
84      if(!inum)
85          goto fail;
86
87      group = atoi(argv[2]);
88
89      current->m_ino.i_gid = group;
90
91      /*put back all the minos not in use*/
92      for( i = 0; i < num_splits; i ++ )
93      {
94          if( to_free[i] )
95              k_t_put_mino( to_free[i] );
96      }
97
98      /*free what is left*/
99      for( i = 0; i < num_splits; i++ )
100     {
101         if( split[i] )
102         {
103             free( split[i] );
104             split[i] = NULL;
105         }
106     }
107
108     return 0;
```

```
109
110
111 fail:
112     /*free what is left*/
113     for( i = 0; i < num_splits; i++ )
114     {
115         if( split[i] )
116         {
117             free( split[i] );
118             split[i] = NULL;
119         }
120     }
121
122     return -1;
123
124 }
```

4.3 Change mode

4.3.1 Definition

Listing 39: chmod.h

```
1  /*****************************************************************************
2  file        : chmod.h
3  author      : jk
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int chmod_cmd( proc_t*, int argc, char** argv );
```

4.3.2 Implementation

Listing 40: chmod.c

```
1  /*****************************************************************************
2  file        : chmod.c
3  author      :
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <sys/stat.h>
13 #include <sys/types.h>
14
15 #include "chmod.h"
16 #include "../string_funcs.h"
17 #include "../global_defs.h"
18 #include "../k_t.h"
19 #include "../mino_t.h"
20 #include "../proc_t.h"
21
22
23 extern k_t* kr;
24
25
26 int chmod_cmd( proc_t* pr, int argc, char** argv )
27 {
28     int i                   = 0;
29     int inum                = 0;
30     int num_splits          = 0;
31     int to_free[MAX_DEPTH]  = { 0 };
32     char* split[MAX_DEPTH]  = { 0 };
33     mino_t* current         = NULL;
34     unsigned long mode          = 0;
35
36     if(argc < 3)
```

```
37      {
38          printf("Usage: touch [pathname] [Octal Permissions]\n");
39          goto fail;
40      }
41
42      if( is_abs_path( argv[1] ) )
43      {
44          current = kr->m_mino_tb[0];
45      }
46      else
47      {
48          current = pr->m_cwd;
49      }
50
51      /*current and dst now point at the starting directories*/
52      if( !current)
53      {
54          printf( "error: couldn't find starting directory\n" );
55          goto fail;
56      }
57
58      /*split the first path*/
59      if( !split_path( argv[1] , MAX_DEPTH, split, &num_splits ) )
60      {
61          printf( "error splitting path\n" );
62          goto fail;
63      }
64
65      /*walk through the directories*/
66      for( i = 0; i < num_splits; i++ )
67      {
68          /*find the inode number for the current directory*/
69          inum = k_t_find_dir_num( current, split[i] );
70          if( inum != -1 )
71          {
72              /*get the next inode*/
73              to_free[i] = inum;
74              k_t_get_mino( current->m_fd, inum, &current );
75          }
76          else
77          {
78              printf( "could not find directory %s\n", split[i] );
79              goto fail;
80          }
81      }
82
83      /*if the file/dir can't be found*/
84      if(!inum)
85          goto fail;
86
87      mode = strtoul(argv[2], NULL, 8);
88      current->m_ino.i_mode &= mode;
89      current->m_ino.i_mode |= mode;
90
91      /*put back all the minos not in use*/
92      for( i = 0; i < num_splits; i ++ )
93      {
94          if( to_free[i] )
95              k_t_put_mino( to_free[i] );
96      }
97
98      /*free what is left*/
99      for( i = 0; i < num_splits; i++ )
100     {
101         if( split[i] )
102         {
103             free( split[i] );
104             split[i] = NULL;
105         }
106     }
107
108     return 0;
109
110
111 fail:
112     /*free what is left*/
113     for( i = 0; i < num_splits; i++ )
114     {
115         if( split[i] )
116         {
117             free( split[i] );
```

```
118             split[i] = NULL;
119         }
120     }
121
122     return -1;
123
124 }
```

4.4 Change owner

4.4.1 Definition

Listing 41: chown.h

```
1  /*****************************************************************************
2  file        : chown.h
3  author      : jk
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int chown_cmd( proc_t*, int argc, char** argv );
```

4.4.2 Implementation

Listing 42: chown.c

```
1  /*****************************************************************************
2  file        : chown.c
3  author      :
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <sys/stat.h>
13 #include <sys/types.h>
14
15 #include "chown.h"
16 #include "../string_funcs.h"
17 #include "../global_defs.h"
18 #include "../k_t.h"
19 #include "../mino_t.h"
20 #include "../proc_t.h"
21
22
23 extern k_t* kr;
24
25
26 int chown_cmd( proc_t* pr, int argc, char** argv )
27 {
28     int i                   = 0;
29     int inum                = 0;
30     int user                = 0;
31     int num_splits          = 0;
32     int to_free[MAX_DEPTH]  = { 0 };
33     char* split[MAX_DEPTH]  = { 0 };
34     mino_t* current         = NULL;
35
36     if(argc < 3)
37     {
38         printf("Usage: touch [pathname] [User ID]\n");
39         goto fail;
40     }
41
42     if( is_abs_path( argv[1] ) )
43     {
44         current = kr->m_mino_tb[0];
45     }
```

```
46      else
47      {
48          current = pr->m_cwd;
49      }
50
51      /*current and dst now point at the starting directories*/
52      if( !current)
53      {
54          printf( "error: couldn't find starting directory\n" );
55          goto fail;
56      }
57
58      /*split the first path*/
59      if( !split_path( argv[1] , MAX_DEPTH, split, &num_splits ) )
60      {
61          printf( "error splitting path\n" );
62          goto fail;
63      }
64
65      /*walk through the directories*/
66      for( i = 0; i < num_splits; i++ )
67      {
68          /*find the inode number for the current directory*/
69          inum = k_t_find_dir_num( current, split[i] );
70          if( inum != -1 )
71          {
72              /*get the next inode*/
73              to_free[i] = inum;
74              k_t_get_mino( current->m_fd, inum, &current );
75          }
76          else
77          {
78              printf( "could not find directory %s\n", split[i] );
79              goto fail;
80          }
81      }
82
83      /*if the file/dir can't be found*/
84      if(!inum)
85          goto fail;
86
87      user = atoi(argv[2]);
88
89      current->m_ino.i_uid = user;
90
91      /*put back all the minos not in use*/
92      for( i = 0; i < num_splits; i ++ )
93      {
94          if( to_free[i] )
95              k_t_put_mino( to_free[i] );
96      }
97
98      /*free what is left*/
99      for( i = 0; i < num_splits; i++ )
100     {
101         if( split[i] )
102         {
103             free( split[i] );
104             split[i] = NULL;
105         }
106     }
107
108     return 0;
109
110
111 fail:
112     /*free what is left*/
113     for( i = 0; i < num_splits; i++ )
114     {
115         if( split[i] )
116         {
117             free( split[i] );
118             split[i] = NULL;
119         }
120     }
121
122     return -1;
123
124 }
```

4.5 Create file

4.5.1 Definition

Listing 43: creat.h

```
1  /*****************************************************************************
2  file        : creat.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int creat_cmd( proc_t*, int argc, char** argv );
```

4.5.2 Implementation

Listing 44: creat.c

```
1  /*****************************************************************************
2  file        : creat.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include <time.h>
12
13 #include "creat.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18 #include "../string_funcs.h"
19
20
21 extern k_t* kr;
22
23 int creat_cmd( proc_t* pr, int argc, char** argv )
24 {
25     if( !kr )
26         goto fail;
27
28     char dn[MAX_NAME]       = { '\0' }; //path to folder to create
29     char bn[MAX_NAME]       = { '\0' }; //name of folder to create
30     char b[BLK_SZ]          = { '\0' };
31     char* split[MAX_DEPTH]  = { '\0' }; //split path elements
32     char* cp                = NULL;
33     int to_free[MAX_DEPTH]  = { 0 };    //array of minos to put back
34     int num_splits          = 0;
35     int i                   = 0;
36     int inum                = 0;
37     int newdir_size         = 0;
38     int int_size            = 0;
39     int sub_val             = 0;
40     mino_t* current         = NULL;     //current directory
41     mino_t* new_mino        = NULL;
42     inode new_ino           = { 0 };    //inode we are adding
43     mnt_t* mt               = NULL;     //mount
44     ext2_dir* dp            = NULL;
45
46     if( !kr )
47         goto fail;
48
49     if( argc != 2 )
50     {
51         printf( "usage: mkdir directory\n" );
52         goto fail;
53     }
54
55     strcpy( bn, loc_basename( argv[1] ) );
```

```
56      strcpy( dn, loc_dirname( argv[1] ) );
57
58      if( is_abs_path( bn ) )
59      {
60          current = kr->m_mino_tb[0];
61      }
62      else
63      {
64          current = pr->m_cwd;
65      }
66
67      if( !current )
68      {
69          printf( "error: couldn't find starting directory\n" );
70          goto fail;
71      }
72
73      if( !split_path( dn, MAX_DEPTH, split, &num_splits ) )
74      {
75          printf( "error splitting path\n" );
76          goto fail;
77      }
78
79      /*walk through the directories*/
80      for( i = 0; i < num_splits; i++ )
81      {
82          /*find the inode number for the current directory*/
83          inum = k_t_find_dir_num( current, split[i] );
84          if( inum != -1 )
85          {
86              /*get the next inode*/
87              to_free[i] = inum;
88              k_t_get_mino( current->m_fd, inum, &current );
89          }
90          else
91          {
92              printf( "could not find directory %s\n", split[i] );
93              goto fail;
94          }
95      }
96
97      /*start of mkdir code*/
98      if( k_t_find_dir_num( current, bn ) != -1 )
99      {
100         printf( "error, file already exists\n" );
101         goto fail;
102     }
103
104     /*make the inode*/
105     mino_t_set_ino( &new_ino,
106             pr->m_uid,
107             pr->m_gid,
108             0,
109             DEF_FMODE,
110             0 );
111
112     /*write it to disk*/
113     inum = k_t_add_ino_to_fs( current->m_fd, &new_ino );
114
115     if( inum == 0 )
116     {
117         printf( "error adding new inode to the filesystem\n" );
118         goto fail;
119     }
120
121     /*re-open it as a mino to do work*/
122     k_t_get_mino( current->m_fd, inum, &new_mino );
123
124     if( !new_mino )
125     {
126         printf( "error retrieveing mino\n" );
127         goto fail;
128     }
129
130     /*add a data block*/
131     k_t_add_dblocks_to_mino( new_mino->m_fd, inum, 1 );
132
133     /*add ourselves to the parent*/
134     mt = k_t_get_mnt_from_fd( current->m_fd );
135     while( mt->m_busy ) {};
136     mnt_t_lock( mt );
```

```
137
138     k_t_get_blk( b, current->m_fd, current->m_ino.i_block[0] );
139
140     dp = (ext2_dir*)b;
141     cp = b;
142
143     /*make it a multiple of four*/
144     newdir_size = 4 * ( ( 8 + strlen( bn ) + 3 ) / 4 );
145
146     /*find the last directory entry*/
147     while( cp < ( b + 1024 ) )
148     {
149         if( cp + dp->rec_len >= b + 1024 )
150             break;
151
152         cp += dp->rec_len;
153         dp = (ext2_dir*) cp;
154     }
155
156     int_size = 4 * ( ( 8 + dp->name_len + 3 ) / 4 );
157
158     /*add it*/
159     if( dp->rec_len - int_size >= newdir_size )
160     {
161         sub_val = dp->rec_len - int_size;
162         dp->rec_len = int_size;
163
164         cp -= sub_val;
165         dp = (ext2_dir*)cp;
166         dp->inode = inum;
167
168         dp->name_len = strlen( bn );
169
170         strncpy( dp->name, bn, strlen( bn ) );
171         dp->rec_len = sub_val;
172
173         current->m_ino.i_links_count++;
174         current->m_dirty = true;
175         current->m_ino.i_atime = time(0L);
176     }
177
178     memcpy( b + ( 1024 - dp->rec_len ), cp, dp->rec_len );
179     dp = (ext2_dir*)b;
180     cp = b;
181
182     k_t_put_blk( b, current->m_fd, current->m_ino.i_block[0] );
183
184
185     /*put everything back*/
186     for( i = 0; i < num_splits; i++ )
187     {
188         if( to_free[i] )
189             k_t_put_mino( to_free[i] );
190     }
191
192     k_t_put_mino( new_mino->m_ino_num );
193
194
195     /*unlock the fs*/
196     mnt_t_unlock( mt );
197
198     /*free the split*/
199     for( i = 0; i < num_splits; i++ )
200     {
201         if( split[i] )
202             free( split[i] );
203     }
204     return 0;
205
206 fail:
207     if( mt )
208         mnt_t_unlock( mt );
209
210     /*free the split*/
211     for( i = 0; i < num_splits; i++ )
212     {
213         if( split[i] )
214             free( split[i] );
215     }
216
217     return -1;
```

```
218 }
```

4.6 Link

4.6.1 Definition

Listing 45: link.h

```
1  /*****************************************************************************
2  file        : link.h
3  author      : nc
4  desc        :
5  date        : 04-06-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int link_cmd( proc_t*, int argc, char** argv );
```

4.6.2 Implementation

Listing 46: link.c

```
1  /*****************************************************************************
2  file        : link.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include <sys/stat.h>
12
13 #include "link.h"
14 #include "../string_funcs.h"
15 #include "../global_defs.h"
16 #include "../k_t.h"
17 #include "../mino_t.h"
18 #include "../proc_t.h"
19
20
21 extern k_t* kr;
22
23 int link_cmd( proc_t* pr, int argc, char** argv )
24 {
25     char b[BLK_SZ]          = { '\0' };     //get higher
26     char* split[MAX_DEPTH]  = { '\0' };     //split path elements
27     char bn_1[MAX_PATH]     = { '\0' };     //fist basename
28     char dn_1[MAX_PATH]     = { '\0' };     //first dirname
29     char bn_2[MAX_PATH]     = { '\0' };     //second basename
30     char dn_2[MAX_PATH]     = { '\0' };     //second dirname
31     char name[MAX_NAME]     = { '\0' };
32     int to_free[MAX_DEPTH]  = { 0 };        //array of minos to put back
33     int num_splits          = 0;
34     int i                   = 0;
35     int inum                = 0;
36     u32 target_inum         = 0;
37     int tmp_size            = 0;
38     mino_t* current         = NULL;         //current directory
39     mino_t* target          = NULL;
40     mino_t* dst             = NULL;         //destination dir
41     char* cp                = NULL;         //  "
42     ext2_dir* dp            = NULL;         //  "
43
44     if( !kr )
45         goto fail;
46
47     if( argc != 3 )
48     {
49         printf( "usage: link [file 1] [file 2]\n" );
50         goto fail;
51     }
```

```
52
53      /*get first and second basenames*/
54      strcpy( bn_1, loc_basename( argv[1] ) );
55      strcpy( dn_1, loc_dirname( argv[1] ) );
56      strcpy( bn_2, loc_basename( argv[2] ) );
57      strcpy( dn_2, loc_dirname( argv[2] ) );
58
59      /*watch me as i navigate ha ha ha ha haaaaaaaaa >.<*/
60      if( is_abs_path( bn_1 ) )
61      {
62          current = kr->m_mino_tb[0];
63      }
64      else
65      {
66          current = pr->m_cwd;
67      }
68
69      if( is_abs_path( bn_2 ) )
70      {
71          dst = kr->m_mino_tb[0];
72      }
73      else
74      {
75          dst = pr->m_cwd;
76      }
77
78      /*current and dst now point at the starting directories*/
79      if( !current || !dst )
80      {
81          printf( "error: couldn't find starting directory\n" );
82          goto fail;
83      }
84
85      /*split the first path*/
86      if( !split_path( dn_1, MAX_DEPTH, split, &num_splits ) )
87      {
88          printf( "error splitting path\n" );
89          goto fail;
90      }
91
92      /*walk through the directories*/
93      for( i = 0; i < num_splits; i++ )
94      {
95          /*find the inode number for the current directory*/
96          inum = k_t_find_dir_num( current, split[i] );
97          if( inum != -1 )
98          {
99              /*get the next inode*/
100             to_free[i] = inum;
101             k_t_get_mino( current->m_fd, inum, &current );
102         }
103         else
104         {
105             printf( "could not find directory %s\n", split[i] );
106             goto fail;
107         }
108     }
109
110     /*put back all the minos not in use*/
111     for( i = 0; i < num_splits; i ++ )
112     {
113         if( to_free[i] )
114             k_t_put_mino( to_free[i] );
115     }
116
117     /*current is now our relative dir so find the inode we want*/
118     k_t_get_blk( b, pr->m_cwd->m_fd, current->m_ino.i_block[0] );
119
120     dp = (ext2_dir*)b;
121     cp = b;
122
123     /*walk*/
124     while( true )
125     {
126         /*have to do this in case of names that contain substrings*/
127         memset( name, 0, MAX_NAME );
128         memcpy( name, dp->name, dp->name_len );
129
130         /*found it*/
131         if( !strcmp( name, bn_1 ) )
132             break;
```

```
133
134         if( cp >= ( b + 1024 ) )
135         {
136             printf( "could not find file\n" );
137             goto fail;
138         }
139
140         cp += dp->rec_len;
141         dp = (ext2_dir*)cp;
142     }
143
144     /*get dis inode*/
145     target_inum = dp->inode;
146
147     if( !inum )
148         goto fail;
149
150     /*don't get the node yet - find where we are putting it*/
151     memset( b, 0, BLK_SZ );
152
153     /*free the first split*/
154     for( i = 0; i < num_splits; i++ )
155     {
156         if( split[i] )
157         {
158             free( split[i] );
159             split[i] = NULL;
160         }
161     }
162
163     /*split path two*/
164     if( !split_path( dn_2, MAX_DEPTH, split, &num_splits ) )
165     {
166         printf( "error splitting path\n" );
167         goto fail;
168     }
169
170     /*walk through the directories*/
171     memset( to_free, 0, MAX_PATH );
172
173     for( i = 0; i < num_splits; i++ )
174     {
175         /*find the inode number for the current directory*/
176         inum = k_t_find_dir_num( current, split[i] );
177         if( inum != -1 )
178         {
179             /*get the next inode*/
180             to_free[i] = inum;
181             k_t_get_mino( dst->m_fd, inum, &dst );
182         }
183         else
184         {
185             printf( "could not find directory %s\n", split[i] );
186             goto fail;
187         }
188     }
189
190     /*put back all the minos not in use*/
191     for( i = 0; i < num_splits; i ++ )
192     {
193         if( to_free[i] )
194             k_t_put_mino( to_free[i] );
195     }
196
197     /*dst now points at the target directory*/
198     memset( b, 0, BLK_SZ );
199     k_t_get_blk( b, pr->m_cwd->m_fd, dst->m_ino.i_block[0] );
200
201     /*show some stuff*/
202     printf( "current inode is %i\n", current->m_ino_num );
203     printf( "dst inode is %i\n", dst->m_ino_num );
204
205     dp = (ext2_dir*)b;
206     cp = b;
207
208     /*walk*/
209     while( cp + dp->rec_len < ( b + 1024 ) )
210     {
211         /*have to do this in case of names that contain substrings*/
212         memset( name, 0, MAX_NAME );
213         memcpy( name, dp->name, dp->name_len );
```

```
214
215         /*found it*/
216         if( !strcmp( name, bn_2 ) )
217         {
218             printf( "file already exists\n" );
219             goto fail;
220         }
221
222         cp += dp->rec_len;
223         dp = (ext2_dir*)cp;
224     }
225
226     /*now do work - dp points at the last entry so re-adjust its size*/
227     tmp_size = dp->rec_len;
228     dp->rec_len = 4 * ( ( 8 + dp->name_len + 3 ) / 4 );
229     tmp_size -= dp->rec_len;
230
231     cp += dp->rec_len;
232     dp = (ext2_dir*)cp;
233
234     /*zero out the rest of the block*/
235     memset( cp, 0, tmp_size );
236
237     /*add an entry - target contains our inode*/
238     dp->inode = target_inum;
239     memcpy( dp->name, bn_2, strlen( bn_2 ) );
240     dp->file_type = FILE_TYPE;
241     dp->name_len = strlen( bn_2 );
242
243     dp->rec_len = tmp_size;
244
245     /*put the block back*/
246     k_t_put_blk( b, pr->m_cwd->m_fd, dst->m_ino.i_block[0] );
247
248     /*get the target inode*/
249     if( !k_t_get_mino( current->m_fd, target_inum, &target ) ||
250             S_ISDIR( target->m_ino.i_mode ) )
251     {
252         printf( "error retrieving target inode\n" );
253         goto fail;
254     }
255
256     printf( "got inode %i to link\n", target_inum );
257
258     /*update the link count*/
259     target->m_ino.i_links_count++;
260     target->m_dirty = true;
261
262     /*put the mino back*/
263     k_t_put_mino( target_inum );
264
265     /*free what is left*/
266     for( i = 0; i < num_splits; i++ )
267     {
268         if( split[i] )
269         {
270             free( split[i] );
271             split[i] = NULL;
272         }
273     }
274
275     return 0;
276
277 fail:
278
279     /*free the split*/
280     for( i = 0; i < num_splits; i++ )
281     {
282         if( split[i] )
283         {
284             free( split[i] );
285             split[i] = NULL;
286         }
287     }
288
289
290     return -1;
291 }
```

4.7 List

4.7.1 Definition

Listing 47: ls.h

```
1  /*****************************************************************************
2  file        : ls.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int ls_cmd( proc_t*, int argc, char** argv );
```

4.7.2 Implementation

Listing 48: ls.c

```
1  /*****************************************************************************
2  file        : ls.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <sys/stat.h>
13 #include <sys/types.h>
14
15 #include "ls.h"
16 #include "../string_funcs.h"
17 #include "../global_defs.h"
18 #include "../k_t.h"
19 #include "../mino_t.h"
20 #include "../proc_t.h"
21
22
23 extern k_t* kr;
24
25
26 /*****************************************************************************
27 function    : int ls_cmd( proc_t* pr, int argc, char** argv )
28 author      : jk
29 desc        : ls command
30 date        :
31  *****************************************************************************/
32 int ls_cmd( proc_t* pr, int argc, char** argv )
33 {
34     mino_t* tmp             = NULL;
35     int i                   = 0;
36     int inum                = 0;
37     int num_splits          = 0;
38     int to_free[MAX_DEPTH]  = { 0 };
39     char buf[BLK_SZ]        = { 0 };
40     char lnk[BLK_SZ]        = { 0 };
41     char name[256]          = { 0 };
42     char* split[MAX_DEPTH]  = { 0 };
43     char* cp                = NULL;
44     ext2_dir* dp            = NULL;
45     mino_t* current         = NULL;
46     time_t temp         = 0;
47
48     if( !kr || !pr || !argc )
49         goto fail;
50
51     /*path was provided*/
52     if( argc > 1 )
53     {
54         if( is_abs_path( argv[1] ) )
55         {
```

```
56              current = kr->m_mino_tb[0];
57          }
58          else
59          {
60              current = pr->m_cwd;
61          }
62          if( !current )
63          {
64              printf( "error: couldn't find starting directory\n" );
65              goto fail;
66          }
67
68          if( !split_path( argv[1], MAX_DEPTH, split, &num_splits ) )
69          {
70              printf( "error splitting path\n" );
71              goto fail;
72          }
73
74          /*walk through the directories*/
75          for( i = 0; i < num_splits; i++ )
76          {
77              /*find the inode number for the current directory*/
78              inum = k_t_find_dir_num( current, split[i] );
79              if( inum != -1 )
80              {
81                  /*get the next inode*/
82                  to_free[i] = inum;
83                  k_t_get_mino( current->m_fd, inum, &current );
84              }
85              else
86              {
87                  printf( "could not find directory %s\n", split[i] );
88                  goto fail;
89              }
90          }
91
92          /*free what we don't need*/
93          for( i = 0; i < num_splits; i ++ )
94          {
95              if( to_free[i] )
96                  k_t_put_mino( to_free[i] );
97          }
98
99      }
100     else
101     {
102         current = pr->m_cwd;
103     }
104
105
106     /*don't need to lock since we aren't writing to the fs*/
107     k_t_get_blk( buf, current->m_fd, current->m_ino.i_block[0] );
108     cp = buf;
109     dp = (ext2_dir*) buf;
110
111     i = 0;
112     /*read the contents*/
113     while( cp && cp < ( buf + BLK_SZ ) )
114     {
115         k_t_get_mino( current->m_fd, dp->inode, &tmp );//make new m inode
116
117         /*print ino number*/
118         printf( "%i\t", dp->inode );
119
120         /*print mode and other information of file/directory*/
121         printf( (S_ISDIR(tmp->m_ino.i_mode)) ? "d" :
122                 (S_ISLNK(tmp->m_ino.i_mode) ? "l" : "-"));
123
124         /*check r/w/x bits*/
125         for( i = 0; i < 7; i+= 3 )
126         {
127             printf( ( ( tmp->m_ino.i_mode & 0x0100 >> i ) == 0 ) ?
128                     "-" : "r");
129             printf( ( ( tmp->m_ino.i_mode & 0x0100 >> ( i + 1 ) ) == 0 ) ?
130                     "-" : "w");
131             printf( ( ( tmp->m_ino.i_mode & 0x0100 >> ( i + 2 ) ) == 0 ) ?
132                     "-" : "x");
133         }
134
135         printf( "   " );
136
```

```
137         /*show the number of links*/
138         printf( "%i  ", tmp->m_ino.i_links_count );
139
140         /*show uid*/
141         printf( "%i  ", tmp->m_ino.i_uid );
142
143         /*show gid*/
144         printf( "%i  ", tmp->m_ino.i_gid );
145
146         /*show the ctime*/
147         temp = tmp->m_ino.i_ctime;
148         printf( "%.24s\t", ctime( &(temp) ) );
149
150         printf( "%d %d \t", tmp->m_ino.i_uid,
151                 tmp->m_ino.i_size);
152
153
154
155         /*print name of file/directory*/
156         memset( name, 0, 256);
157         memcpy( name, dp->name, dp->name_len);
158         printf( "%s", name);
159
160         /*print link if found*/
161         if(S_ISLNK(tmp->m_ino.i_mode))
162         {
163             k_t_get_blk(lnk, tmp->m_fd, tmp->m_ino.i_block[0]);
164             printf(" -> %s", lnk);
165         }
166         printf("\n");
167         cp += dp->rec_len;
168         dp = (ext2_dir*)cp;
169
170     }
171
172
173 fail:
174
175     /*free the split*/
176     for( i = 0; i < num_splits; i++ )
177     {
178         if( split[i] )
179         {
180             free( split[i] );
181             split[i] = NULL;
182         }
183     }
184
185
186     return -1;
187 }
```

4.8 Make directory

4.8.1 Definition

Listing 49: mkdir.h

```
1  /*****************************************************************************
2  file        : mkdir.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int mkdir_cmd( proc_t*, int argc, char** argv );
```

4.8.2 Implementation

Listing 50: mkdir.c

```
1  /*****************************************************************************
```

```
2  file        : mkdir.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   ********************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include <time.h>
12
13 #include "mkdir.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18 #include "../string_funcs.h"
19
20
21 extern k_t* kr;
22
23 int mkdir_cmd( proc_t* pr, int argc, char** argv )
24 {
25     char dn[MAX_NAME]       = { '\0' }; //path to folder to create
26     char bn[MAX_NAME]       = { '\0' }; //name of folder to create
27     char b[BLK_SZ]          = { '\0' };
28     char* split[MAX_DEPTH]  = { '\0' }; //split path elements
29     char* cp                = NULL;
30     int to_free[MAX_DEPTH]  = { 0 };    //array of minos to put back
31     int num_splits          = 0;
32     int i                   = 0;
33     int inum                = 0;
34     int newdir_size         = 0;
35     int int_size            = 0;
36     int sub_val             = 0;
37     mino_t* current         = NULL;     //current directory
38     mino_t* new_mino        = NULL;
39     inode new_ino           = { 0 };    //inode we are adding
40     mnt_t* mt               = NULL;     //mount
41     ext2_dir* dp            = NULL;
42
43     if( !kr )
44         goto fail;
45
46     if( argc != 2 )
47     {
48         printf( "usage: mkdir directory\n" );
49         goto fail;
50     }
51
52     strcpy( bn, loc_basename( argv[1] ) );
53     strcpy( dn, loc_dirname( argv[1] ) );
54
55     if( is_abs_path( argv[1] ) )
56     {
57         current = kr->m_mino_tb[0];
58     }
59     else
60     {
61         current = pr->m_cwd;
62     }
63
64     if( !current )
65     {
66         printf( "error: couldn't find starting directory\n" );
67         goto fail;
68     }
69
70     if( !split_path( dn, MAX_DEPTH, split, &num_splits ) )
71     {
72         printf( "error splitting path\n" );
73         goto fail;
74     }
75
76     /*walk through the directories*/
77     for( i = 0; i < num_splits; i++ )
78     {
79         /*find the inode number for the current directory*/
80         inum = k_t_find_dir_num( current, split[i] );
81         if( inum != -1 )
82         {
```

```
83              /*get the next inode*/
84              to_free[i] = inum;
85              k_t_get_mino( current->m_fd, inum, &current );
86          }
87          else
88          {
89              printf( "could not find directory %s\n", split[i] );
90              goto fail;
91          }
92      }
93
94      /*start of mkdir code*/
95      if( k_t_find_dir_num( current, bn ) != -1 )
96      {
97          printf( "error, directory already exists\n" );
98          goto fail;
99      }
100
101     /*make the inode*/
102     mino_t_set_ino( &new_ino,
103             pr->m_uid,
104             pr->m_gid,
105             BLK_SZ,
106             DEF_DMODE,
107             DEF_DLINKS );
108
109     /*write it to disk*/
110     inum = k_t_add_ino_to_fs( current->m_fd, &new_ino );
111
112     if( inum == 0 )
113     {
114         printf( "error adding new inode to the filesystem\n" );
115         goto fail;
116     }
117
118     /*re-open it as a mino to do work*/
119     k_t_get_mino( current->m_fd, inum, &new_mino );
120
121     if( !new_mino )
122     {
123         printf( "error retrieveing mino\n" );
124         goto fail;
125     }
126
127     /*add a data block*/
128     k_t_add_dblocks_to_mino( new_mino->m_fd, inum, 1 );
129
130     /*lock the fs*/
131     mt = k_t_get_mnt_from_fd( new_mino->m_fd );
132     while( mt->m_busy ) {};
133     mnt_t_lock( mt );
134
135     k_t_get_blk( b, new_mino->m_fd, new_mino->m_ino.i_block[0] );
136
137     /*add parents*/
138     dp = (ext2_dir*)b;
139     dp->inode = inum;
140     strncpy( dp->name, ".", 1 );
141     dp->name_len = 1;
142     dp->rec_len = 12;
143
144     cp = b;
145     cp += dp->rec_len;
146     dp = (ext2_dir*)cp;
147
148     dp->inode = current->m_ino_num;
149     dp->name_len = 2;
150     strncpy(dp->name, "..",2);
151     dp->rec_len = BLK_SZ - 12;
152
153     k_t_put_blk( b, new_mino->m_fd, new_mino->m_ino.i_block[0] );
154     mnt_t_unlock( mt );
155
156     /*add ourselves to the parent*/
157     mt = k_t_get_mnt_from_fd( current->m_fd );
158     while( mt->m_busy ) {};
159     mnt_t_lock( mt );
160
161     k_t_get_blk( b, current->m_fd, current->m_ino.i_block[0] );
162
163     dp = (ext2_dir*)b;
```

```
164     cp = b;
165
166     /*make it a multiple of four*/
167     newdir_size = 4 * ( ( 8 + strlen( bn ) + 3 ) / 4 );
168
169     /*find the last directory entry*/
170     while( cp < ( b + 1024 ) )
171     {
172         if( cp + dp->rec_len >= b + 1024 )
173             break;
174
175         cp += dp->rec_len;
176         dp = (ext2_dir*) cp;
177     }
178
179     int_size = 4 * ( ( 8 + dp->name_len + 3 ) / 4 );
180
181     /*add it*/
182     if( dp->rec_len - int_size >= newdir_size ) //if there is enough room for the new directory
183     {
184         sub_val = dp->rec_len - int_size;
185         dp->rec_len = int_size;
186
187         cp -= sub_val;
188         dp = (ext2_dir*)cp;
189         dp->inode = inum;
190
191         dp->name_len = strlen( bn );
192
193         strncpy( dp->name, bn, strlen( bn ) );
194         dp->rec_len = sub_val;
195
196         current->m_ino.i_links_count++;
197         current->m_dirty = true;
198         current->m_ino.i_atime = time(0L);
199     }
200
201     memcpy( b + ( 1024 - dp->rec_len ), cp, dp->rec_len );
202     dp = (ext2_dir*)b;
203     cp = b;
204
205     k_t_put_blk( b, current->m_fd, current->m_ino.i_block[0]);//write new data back to system
206
207
208     /*put everything back*/
209     for( i = 0; i < num_splits; i++ )
210     {
211         if( to_free[i] )
212             k_t_put_mino( to_free[i] );
213     }
214
215     /*de-reference new inode*/
216     new_mino->m_refc--;
217     k_t_put_mino( new_mino->m_ino_num );
218
219
220     /*unlock the fs*/
221     mnt_t_unlock( mt );
222
223     /*free the split*/
224     for( i = 0; i < num_splits; i++ )
225     {
226         if( split[i] )
227             free( split[i] );
228     }
229     return 0;
230
231 fail:
232     if( mt )
233         mnt_t_unlock( mt );
234
235     /*free the split*/
236     for( i = 0; i < num_splits; i++ )
237     {
238         if( split[i] )
239             free( split[i] );
240     }
241
242     return -1;
243 }
```

4.9 Print working directory

4.9.1 Definition

Listing 51: pwd.h

```
1  /*****************************************************************************
2  file        : pwd.h
3  author      : nc
4  desc        :
5  date        : 04-06-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int pwd_cmd( proc_t*, int argc, char** argv );
```

4.9.2 Implementation

Listing 52: pwd.c

```
1  /*****************************************************************************
2  file        : pwd.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "pwd.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int pwd_cmd( proc_t* pr, int argc, char** argv )
23 {
24     char b[BLK_SZ]          = { '\0' };
25     char* split[MAX_DEPTH]  = { '\0' }; //split path elements
26     int to_free[MAX_DEPTH]  = { 0 };    //array of minos to put back
27     int num_splits          = 0;
28     int i                   = 0;
29     int inum                = 0;
30     mino_t* current         = NULL;     //current directory
31     mino_t* parent          = NULL;
32     char* cp                = NULL;
33     ext2_dir* dp            = NULL;
34
35     if( !kr )
36         goto fail;
37
38     /*start at cwd*/
39     current = pr->m_cwd;
40     parent = kr->m_mino_tb[0];
41
42     while( current &&
43             ( inum = k_t_find_dir_num( current, ".." ) ) )
44     {
45         /*find the parent*/
46         k_t_get_mino( current->m_fd, inum, &parent );
47
48         if( current->m_ino_num == parent->m_ino_num )
49             break;
50
51         if( !parent )
52             goto fail;
53
54         to_free[i] = inum;
55         i++;
```

```
56
57          /*find our name*/
58          k_t_get_blk( b, parent->m_fd, parent->m_ino.i_block[0] );
59          cp = b;
60          dp = (ext2_dir*)b;
61
62          /*walk through the entries*/
63          while( cp < ( b + 1024 ) )
64          {
65              /*found a match*/
66              if( dp->inode == current->m_ino_num )
67              {
68                  /*get memory and copy the name over*/
69                  split[num_splits] = malloc( dp->name_len + 1 );
70
71                  if( !split[num_splits] )
72                  {
73                      printf( "error: no memory\n" );
74                      goto fail;
75                  }
76
77                  memset( split[num_splits], 0, dp->name_len + 1 );
78                  strncpy( split[num_splits], dp->name, dp->name_len );
79                  num_splits++;
80                  break;
81              }
82
83              cp += dp->rec_len;
84              dp = (ext2_dir*) cp;
85          }
86
87          current = parent;
88          parent = NULL;
89      }
90
91      /*print the dir*/
92      printf( "/" );
93
94      for( i = 0; i < num_splits; i++ )
95      {
96          printf( "%s/", split[num_splits - i - 1] );
97      }
98
99      printf( "\n" );
100
101
102     /*put everything back*/
103     for( i = 0; i < num_splits; i++ )
104     {
105         if( to_free[i] )
106             k_t_put_mino( to_free[i] );
107     }
108
109     /*free the split*/
110     for( i = 0; i < num_splits; i++ )
111     {
112         if( split[i] )
113             free( split[i] );
114     }
115
116     return 0;
117
118 fail:
119
120     /*free the split*/
121     for( i = 0; i < num_splits; i++ )
122     {
123         if( split[i] )
124             free( split[i] );
125     }
126
127
128     return -1;
129 }
```

4.10 Remove directory

4.10.1 Definition

Listing 53: rmdir.h

```
/*****************************************************************************
file        : mkdir.h
author      :
desc        :
date        : 04-01-13
 *****************************************************************************/
#pragma once

#include "../global_defs.h"
#include "../proc_t.h"

int rmdir_cmd( proc_t*, int argc, char** argv );
```

4.10.2 Implementation

Listing 54: rmdir.c

```
/*****************************************************************************
file        : mkdir.c
author      :
desc        :
date        : 04-17-13
 *****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

#include "mkdir.h"
#include "../global_defs.h"
#include "../k_t.h"
#include "../mino_t.h"
#include "../proc_t.h"
#include "../string_funcs.h"

extern k_t* kr;

int rmdir_cmd( proc_t* pr, int argc, char** argv )
{
    char dn[MAX_NAME]       = { '\0' }; //path to folder to remove
    char bn[MAX_NAME]       = { '\0' }; //name of folder to remove
    char b[BLK_SZ]          = { '\0' };
    char* split[MAX_DEPTH]  = { '\0' }; //split path elements
    char* cp                = NULL;
    int to_free[MAX_DEPTH]  = { 0 };    //array of minos to put back
    int num_splits          = 0;
    int i                   = 0;
    int inum                = 0;
    int par_ino             = 0;
    mino_t* current         = NULL;     //current directory
    mnt_t* mt               = NULL;     //mount
    ext2_dir* dp            = NULL;

    if( !kr )
        goto fail;

    if( argc != 2 )
    {
        printf( "usage: rmdir directory\n" );
        goto fail;
    }

    strcpy( bn, loc_basename( argv[1] ) );
    strcpy( dn, loc_dirname( argv[1] ) );

    if( is_abs_path( argv[1] ) )
    {
        current = kr->m_mino_tb[0];
    }
    else
    {
        current = pr->m_cwd;
    }

    if( !current )
    {
```

```
61          printf( "error: couldn't find starting directory\n" );
62          goto fail;
63      }
64
65      if( !split_path( dn, MAX_DEPTH, split, &num_splits ) )
66      {
67          printf( "error splitting path\n" );
68          goto fail;
69      }
70
71      /*walk through the directories*/
72      for( i = 0; i < num_splits; i++ )
73      {
74          /*find the inode number for the current directory*/
75          inum = k_t_find_dir_num( current, split[i] );
76          if( inum != -1 )
77          {
78              /*get the next inode*/
79              to_free[i] = inum;
80              k_t_get_mino( current->m_fd, inum, &current );
81          }
82          else
83          {
84              printf( "could not find directory %s\n", split[i] );
85              goto fail;
86          }
87      }
88
89      /*save parent location and enter directory*/
90      par_ino = current->m_ino_num;
91      inum = k_t_find_dir_num(current, bn);
92      i++;
93      to_free[i] = inum;
94
95      if(inum == -1) //if not directory
96          goto fail;
97
98      k_t_get_mino( current->m_fd, inum, &current);
99      k_t_get_blk( b, current->m_fd, current->m_ino.i_block[0] );
100
101     if(current->m_ino.i_links_count > 2)
102         goto fail;
103
104     dp = (ext2_dir*)b;
105     cp = b;
106
107     /*find the last directory entry*/
108     while( cp < ( b + 1024 ) )
109     {
110         if( cp + dp->rec_len >= b + 1024 )
111             break;
112         cp += dp->rec_len;
113         dp = (ext2_dir*) cp;
114     }
115
116     /*if their aren't any file names besides . and .. then deletion is easy*/
117     if(strncmp(dp->name, ".", 1) != 0)
118         goto fail;
119
120     /*delete blocks held by directory*/
121     k_t_del_dblocks_from_mino( current->m_fd,
122         current->m_ino_num, 1 );
123
124     /*delete inode*/
125     k_t_del_ino_from_fs( current->m_fd, current->m_ino_num);
126
127     /*get parent inode*/
128     k_t_get_mino( current->m_fd, par_ino, &current );
129
130     if( !k_t_remove_child( current, bn ) )
131     {
132         printf( "error: could not remove child %s from inode\n", bn );
133         goto fail;
134     }
135
136     current->m_ino.i_links_count--;
137
138     /*this is on purpose*/
139     if( current->m_ino.i_links_count == 0 )
140     {
141         k_t_put_mino( par_ino );
```

```
142             k_t_del_ino_from_fs( current->m_fd, par_ino );
143         }
144         else
145         {
146             k_t_put_mino( par_ino );
147         }
148 
149         /*put everything back*/
150         for( i = 0; i < num_splits; i++ )
151         {
152             if( to_free[i] )
153                 k_t_put_mino( to_free[i] );
154         }
155 
156         /*unlock the fs*/
157         mnt_t_unlock( mt );
158 
159         /*free the split*/
160         for( i = 0; i < num_splits; i++ )
161         {
162             if( split[i] )
163                 free( split[i] );
164         }
165         return 0;
166 
167     fail:
168         if( mt )
169             mnt_t_unlock( mt );
170 
171         /*free the split*/
172         for( i = 0; i < num_splits; i++ )
173         {
174             if( split[i] )
175                 free( split[i] );
176         }
177 
178         return -1;
179     }
```

4.11 Status

4.11.1 Definition

Listing 55: stat.h

```
1  /*****************************************************************************
2  file        : stat.h
3  author      : jk
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7  #pragma once
8 
9  #include <sys/stat.h>
10 #include "../global_defs.h"
11 #include "../proc_t.h"
12 
13 int do_stat( char* pathname, struct stat* stPtr, proc_t* pr );
14 int stat_cmd( proc_t*, int argc, char** argv );
```

4.11.2 Implementation

Listing 56: stat.c

```
1  /*****************************************************************************
2  file        : stat.c
3  author      :
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7 
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <sys/stat.h>
```

```
13  #include <sys/types.h>
14
15  #include "stat.h"
16  #include "../string_funcs.h"
17  #include "../global_defs.h"
18  #include "../k_t.h"
19  #include "../mino_t.h"
20  #include "../proc_t.h"
21
22
23  extern k_t* kr;
24
25
26  int do_stat(char *pathname, struct stat *stPtr, proc_t* pr)
27  {
28      int i                   = 0;
29      int inum                = 0;
30      int num_splits          = 0;
31      int to_free[MAX_DEPTH]  = { 0 };
32      char* split[MAX_DEPTH]  = { 0 };
33      mino_t* current         = NULL;
34
35      if( is_abs_path( pathname ) )
36      {
37          current = kr->m_mino_tb[0];
38      }
39      else
40      {
41          current = pr->m_cwd;
42      }
43
44      /*current and dst now point at the starting directories*/
45      if( !current)
46      {
47          printf( "error: couldn't find starting directory\n" );
48          goto fail;
49      }
50
51      /*split the first path*/
52      if( !split_path( pathname , MAX_DEPTH, split, &num_splits ) )
53      {
54          printf( "error splitting path\n" );
55          goto fail;
56      }
57
58      /*walk through the directories*/
59      for( i = 0; i < num_splits; i++ )
60      {
61          /*find the inode number for the current directory*/
62          inum = k_t_find_dir_num( current, split[i] );
63          if( inum != -1 )
64          {
65              /*get the next inode*/
66              to_free[i] = inum;
67              k_t_get_mino( current->m_fd, inum, &current );
68          }
69          else
70          {
71              printf( "could not find directory %s\n", split[i] );
72              goto fail;
73          }
74      }
75
76      /*if the file/dir can't be found*/
77      if( !inum )
78          goto fail;
79
80      //apparently the device name isn't put into the kernel :(
81      stPtr->st_dev =         current->m_fd;
82      stPtr->st_ino =         current->m_ino_num;
83      stPtr->st_mode =        current->m_ino.i_mode;
84      stPtr->st_nlink =       current->m_ino.i_links_count;
85      stPtr->st_uid =         current->m_ino.i_uid;
86      stPtr->st_gid =         current->m_ino.i_gid;
87      stPtr->st_size =        current->m_ino.i_size;
88      stPtr->st_blksize =     1024;
89      stPtr->st_blocks =      current->m_ino.i_blocks;
90      stPtr->st_atime =       current->m_ino.i_atime;
91      stPtr->st_ctime =       current->m_ino.i_ctime;
92      stPtr->st_mtime =       current->m_ino.i_mtime;
93
```

```
94
95      printf( "filename:\t%s\t", split[num_splits-1] );
96      printf( "device:\t%d\n", (i32)stPtr->st_dev );
97      printf( "inode:\t%d\t", (i32)stPtr->st_ino );
98      printf( "mode:\t%x\t", stPtr->st_mode );
99      printf( "link count:\t%u\t", (u32)stPtr->st_nlink );
100     printf( "Uid:\t%u\t", stPtr->st_uid );
101     printf( "\ngid:\t%u\t", stPtr->st_gid );
102     printf( "size:\t%d\t", (i32)stPtr->st_size );
103     printf( "blk size:\t%d\t", (i32)stPtr->st_blksize );
104     printf( "blocks:\t%d\t\n", (i32)stPtr->st_blocks );
105     printf( "creation time:\t%.24s\t\n", ctime( &(stPtr->st_ctime) ) );
106     printf( "accessed time:\t%.24s\t\n", ctime( &(stPtr->st_atime) ) );
107     printf( "m time:\t\t%.24s\t\n", ctime( &(stPtr->st_mtime) ) );
108
109     /*put back all the minos not in use*/
110     for( i = 0; i < num_splits; i ++ )
111     {
112         if( to_free[i] )
113             k_t_put_mino( to_free[i] );
114     }
115
116     /*free the split*/
117     for( i = 0; i < num_splits; i++ )
118     {
119         if( split[i] )
120         {
121             free( split[i] );
122             split[i] = NULL;
123         }
124     }
125     return 0;
126
127 fail:
128     /*free the split*/
129     for( i = 0; i < num_splits; i++ )
130     {
131         if( split[i] )
132         {
133             free( split[i] );
134             split[i] = NULL;
135         }
136     }
137
138
139     return -1;
140
141 }
142
143 int stat_cmd( proc_t* pr, int argc, char** argv )
144 {
145     int r               = 0;
146     struct stat mystat;
147
148     if( !kr || !pr || !argc )
149         goto fail;
150
151     if(argc < 2)
152     {
153         printf("Usage is: stat [pathname]");
154         goto fail;
155     }
156
157     r = do_stat(argv[1], &mystat, pr);
158
159     if(r == -1)
160         goto fail;
161
162     return 0;
163
164 fail:
165     return -1;
166
167 }
```

4.12 Symbolic link

4.12.1 Definition

Listing 57: symblink.h

```
/*****************************************************************************
file        : symblink.h
author      : jk
desc        :
date        : 04-20-13
 *****************************************************************************/
#pragma once

#include "../global_defs.h"
#include "../proc_t.h"

int symblink_cmd( proc_t*, int argc, char** argv );
```

4.12.2 Implementation

Listing 58: symblink.c

```
/*****************************************************************************
file        : symblink.c
author      :
desc        :
date        : 04-20-13
 *****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>

#include "symblink.h"
#include "../string_funcs.h"
#include "../global_defs.h"
#include "../k_t.h"
#include "../mino_t.h"
#include "../proc_t.h"


extern k_t* kr;

int symblink_cmd( proc_t* pr, int argc, char** argv )
{
    char b[BLK_SZ]          = { '\0' };     //get higher
    char* split[MAX_DEPTH]  = { '\0' };     //split path elements
    char bn_2[MAX_PATH]     = { '\0' };     //second basename
    char dn_2[MAX_PATH]     = { '\0' };     //second dirname
    int to_free[MAX_DEPTH]  = { 0 };        //array of minos to put back
    int num_splits          = 0;
    int i                   = 0;
    int inum                = 0;
    int newdir_size         = 0;
    int int_size            = 0;
    int sub_val         = 0;

    mino_t* current         = NULL;         //current directory
    mino_t* new_mino        = NULL;
    mino_t* dst             = NULL;         //destination dir
    char* cp                = NULL;         //  "
    ext2_dir* dp            = NULL;         //  "
    mnt_t* mt               = NULL;     //mount
    inode new_ino           = { 0 };    //inode we are adding

    if( !kr )
        goto fail;

    if( argc != 3 )
    {
        printf( "usage: symlink [file 1] [file 2]\n" );
        goto fail;
    }

    /*get first and second basenames*/
    strcpy( bn_2, loc_basename( argv[2] ) );
    strcpy( dn_2, loc_dirname( argv[2] ) );

    if( is_abs_path( argv[1]) )
    {
```

```
61          current = kr->m_mino_tb[0];
62      }
63      else
64      {
65          current = pr->m_cwd;
66      }
67
68      if( is_abs_path( argv[2] ) )
69      {
70          dst = kr->m_mino_tb[0];
71      }
72      else
73      {
74          dst = pr->m_cwd;
75      }
76
77      /*current and dst now point at the starting directories*/
78      if( !current || !dst )
79      {
80          printf( "error: couldn't find starting directory\n" );
81          goto fail;
82      }
83
84      /*split the first path*/
85      if( !split_path( argv[1], MAX_DEPTH, split, &num_splits ) )
86      {
87          printf( "error splitting path\n" );
88          goto fail;
89      }
90
91      /*walk through the directories*/
92      for( i = 0; i < num_splits; i++ )
93      {
94          /*find the inode number for the current directory*/
95          inum = k_t_find_dir_num( current, split[i] );
96          if( inum != -1 )
97          {
98              /*get the next inode*/
99              to_free[i] = inum;
100             k_t_get_mino( current->m_fd, inum, &current );
101         }
102         else
103         {
104             printf( "could not find directory %s\n", split[i] );
105             goto fail;
106         }
107     }
108
109     /*put back all the minos not in use*/
110     for( i = 0; i < num_splits; i ++ )
111     {
112         if( to_free[i] )
113             k_t_put_mino( to_free[i] );
114     }
115
116     /*if the file/dir can't be found*/
117     if(!inum)
118         goto fail;
119
120     /*don't get the node yet - find where we are putting it*/
121     memset( b, 0, BLK_SZ );
122
123     /*free the first split*/
124     for( i = 0; i < num_splits; i++ )
125     {
126         if( split[i] )
127         {
128             free( split[i] );
129             split[i] = NULL;
130         }
131     }
132
133     /*split path two*/
134     if( !split_path( dn_2, MAX_DEPTH, split, &num_splits ) )
135     {
136         printf( "error splitting path\n" );
137         goto fail;
138     }
139
140     /*walk through the directories*/
141     memset( to_free, 0, MAX_PATH );
```

```
142
143     for( i = 0; i < num_splits; i++ )
144     {
145         /*find the inode number for the current directory*/
146         inum = k_t_find_dir_num( dst, split[i] );
147         if( inum != -1 )
148         {
149             /*get the next inode*/
150             to_free[i] = inum;
151             k_t_get_mino( dst->m_fd, inum, &dst );
152         }
153         else
154         {
155             printf( "could not find directory %s\n", split[i] );
156             goto fail;
157         }
158     }
159
160     /*start of symbolic link code*/
161     if( k_t_find_dir_num( dst , bn_2 ) != -1 )
162     {
163         printf( "error, directory already exists\n" );
164         goto fail;
165     }
166
167     /*make the inode*/
168     mino_t_set_ino( &new_ino,
169             pr->m_uid,
170             pr->m_gid,
171             BLK_SZ,
172             DEF_LMODE,
173             1 );
174
175     /*write it to disk*/
176     inum = k_t_add_ino_to_fs( current->m_fd, &new_ino );
177
178     if( inum == 0 )
179     {
180         printf( "error adding new inode to the filesystem\n" );
181         goto fail;
182     }
183
184     /*re-open it as a mino to do work*/
185     k_t_get_mino( current->m_fd, inum, &new_mino );
186     to_free[i+1] = inum;
187
188     if( !new_mino )
189     {
190         printf( "error retrieving mino\n" );
191         goto fail;
192     }
193
194     /*add a data block*/
195     k_t_add_dblocks_to_mino( new_mino->m_fd, inum, 1 );
196
197     /*lock the fs*/
198     mt = k_t_get_mnt_from_fd( new_mino->m_fd );
199     while( mt->m_busy ) {};
200     mnt_t_lock( mt );
201
202     /*write pathname to data block*/
203     k_t_get_blk( b, new_mino->m_fd, new_mino->m_ino.i_block[0] );
204
205     strcpy(b,argv[1]);
206
207     k_t_put_blk( b, new_mino->m_fd, new_mino->m_ino.i_block[0] );
208
209     mnt_t_unlock( mt );
210
211     /*add ourselves to the parent*/
212     mt = k_t_get_mnt_from_fd( dst->m_fd );
213     while( mt->m_busy ) {};
214     mnt_t_lock( mt );
215
216     k_t_get_blk( b, dst->m_fd, dst->m_ino.i_block[0] );
217
218     dp = (ext2_dir*)b;
219     cp = b;
220
221     /*make it a multiple of four*/
222     newdir_size = 4 * ( ( 8 + strlen( bn_2 ) + 3 ) / 4 );
```

```
223
224     /*find the last directory entry*/
225     while( cp < ( b + 1024 ) )
226     {
227         if( cp + dp->rec_len >= b + 1024 )
228             break;
229
230         cp += dp->rec_len;
231         dp = (ext2_dir*) cp;
232     }
233
234     int_size = 4 * ( ( 8 + dp->name_len + 3 ) / 4 );
235
236     /*add it*/
237     if( dp->rec_len - int_size >= newdir_size ) //if there is enough room for the new directory
238     {
239         sub_val = dp->rec_len - int_size;
240         dp->rec_len = int_size;
241
242         cp -= sub_val;
243         dp = (ext2_dir*)cp;
244         dp->inode = inum;
245
246         dp->name_len = strlen( bn_2 );
247
248         strncpy( dp->name, bn_2, strlen( bn_2 ) );
249         dp->rec_len = sub_val;
250
251         dst->m_ino.i_links_count++;
252         dst->m_dirty = true;
253         dst->m_ino.i_atime = time(0L);
254     }
255
256     memcpy( b + ( 1024 - dp->rec_len ), cp, dp->rec_len );
257     dp = (ext2_dir*)b;
258     cp = b;
259
260     k_t_put_blk( b, dst->m_fd, dst->m_ino.i_block[0]);//write new data back to system
261
262     /*put everything back*/
263     for( i = 0; i < num_splits; i++ )
264     {
265         if( to_free[i] )
266             k_t_put_mino( to_free[i] );
267     }
268
269     new_mino->m_refc--;
270     k_t_put_mino( new_mino->m_ino_num );
271
272     /*unlock the fs*/
273     mnt_t_unlock( mt );
274
275     /*free what is left*/
276     for( i = 0; i < num_splits; i++ )
277     {
278         if( split[i] )
279         {
280             free( split[i] );
281             split[i] = NULL;
282         }
283     }
284
285     return 0;
286
287 fail:
288
289     /*free the split*/
290     for( i = 0; i < num_splits; i++ )
291     {
292         if( split[i] )
293         {
294             free( split[i] );
295             split[i] = NULL;
296         }
297     }
298
299
300     return -1;
301 }
```

4.13 Touch

4.13.1 Definition

Listing 59: touch.h

```
1  /*****************************************************************************
2  file        : touch.h
3  author      : jk
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int touch_cmd( proc_t*, int argc, char** argv );
```

4.13.2 Implementation

Listing 60: touch.c

```
1  /*****************************************************************************
2  file        : touch.c
3  author      :
4  desc        :
5  date        : 04-20-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <sys/stat.h>
13 #include <sys/types.h>
14
15 #include "touch.h"
16 #include "../string_funcs.h"
17 #include "../global_defs.h"
18 #include "../k_t.h"
19 #include "../mino_t.h"
20 #include "../proc_t.h"
21
22
23 extern k_t* kr;
24
25
26 int touch_cmd( proc_t* pr, int argc, char** argv )
27 {
28     int i                   = 0;
29     int inum                = 0;
30     int num_splits          = 0;
31     int to_free[MAX_DEPTH]  = { 0 };
32     char* split[MAX_DEPTH]  = { 0 };
33     mino_t* current         = NULL;
34
35     if(argc < 2)
36     {
37         printf("Usage: touch [pathname]\n");
38         goto fail;
39     }
40
41     if( is_abs_path( argv[1] ) )
42     {
43         current = kr->m_mino_tb[0];
44     }
45     else
46     {
47         current = pr->m_cwd;
48     }
49
50     /*current and dst now point at the starting directories*/
51     if( !current)
52     {
53         printf( "error: couldn't find starting directory\n" );
54         goto fail;
55     }
```

```
56
57      /*split the first path*/
58      if( !split_path( argv[1] , MAX_DEPTH, split, &num_splits ) )
59      {
60          printf( "error splitting path\n" );
61          goto fail;
62      }
63
64      /*walk through the directories*/
65      for( i = 0; i < num_splits; i++ )
66      {
67          /*find the inode number for the current directory*/
68          inum = k_t_find_dir_num( current, split[i] );
69          if( inum != -1 )
70          {
71              /*get the next inode*/
72              to_free[i] = inum;
73              k_t_get_mino( current->m_fd, inum, &current );
74          }
75          else
76          {
77              printf( "could not find directory %s\n", split[i] );
78              goto fail;
79          }
80      }
81
82      /*if the file/dir can't be found*/
83      if(!inum)
84          goto fail;
85
86      current->m_ino.i_mtime = time(0L);
87      current->m_ino.i_atime = time(0L);
88
89      /*put back all the minos not in use*/
90      for( i = 0; i < num_splits; i ++ )
91      {
92          if( to_free[i] )
93              k_t_put_mino( to_free[i] );
94      }
95
96      /*free what is left*/
97      for( i = 0; i < num_splits; i++ )
98      {
99          if( split[i] )
100         {
101             free( split[i] );
102             split[i] = NULL;
103         }
104     }
105
106     return 0;
107
108
109 fail:
110     /*free what is left*/
111     for( i = 0; i < num_splits; i++ )
112     {
113         if( split[i] )
114         {
115             free( split[i] );
116             split[i] = NULL;
117         }
118     }
119
120     return -1;
121
122 }
```

4.14 Unlink

4.14.1 Definition

Listing 61: unlink.h

```
1  /*****************************************************************************
2  file        : link.h
3  author      : nc
4  desc        :
```

```
5  date        : 04-06-13
6   ****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int unlink_cmd( proc_t*, int argc, char** argv );
```

4.14.2 Implementation

Listing 62: unlink.c

```
1  /****************************************************************************
2  file        : link.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   ****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include <sys/stat.h>
12
13 #include "unlink.h"
14 #include "../string_funcs.h"
15 #include "../global_defs.h"
16 #include "../k_t.h"
17 #include "../mino_t.h"
18 #include "../proc_t.h"
19
20
21 extern k_t* kr;
22
23 int unlink_cmd( proc_t* pr, int argc, char** argv )
24 {
25     char* split[MAX_DEPTH]  = { '\0' };     //split path elements
26     char bn[MAX_PATH]       = { '\0' };     //fist basename
27     char dn[MAX_PATH]       = { '\0' };     //first dirname
28     int to_free[MAX_DEPTH]  = { 0 };        //array of minos to put back
29     int num_splits          = 0;
30     int i                   = 0;
31     u32 inum                = 0;
32     mino_t* current         = NULL;         //current directory
33     mino_t* target          = NULL;
34
35     if( !kr )
36         goto fail;
37
38     if( argc != 2 )
39     {
40         printf( "usage: unlink [file]\n" );
41         goto fail;
42     }
43
44     /*get first and second basenames*/
45     strcpy( bn, loc_basename( argv[1] ) );
46     strcpy( dn, loc_dirname( argv[1] ) );
47
48     if( is_abs_path( bn ) )
49     {
50         current = kr->m_mino_tb[0];
51     }
52     else
53     {
54         current = pr->m_cwd;
55     }
56
57
58     /*current and dst now point at the starting directories*/
59     if( !current )
60     {
61         printf( "error: couldn't find starting directory\n" );
62         goto fail;
63     }
64
65     /*split the first path*/
66     if( !split_path( dn, MAX_DEPTH, split, &num_splits ) )
```

```
67      {
68          printf( "error splitting path\n" );
69          goto fail;
70      }
71
72      /*walk through the directories*/
73      for( i = 0; i < num_splits; i++ )
74      {
75          /*find the inode number for the current directory*/
76          inum = k_t_find_dir_num( current, split[i] );
77          if( inum != -1 )
78          {
79              /*get the next inode*/
80              to_free[i] = inum;
81              k_t_get_mino( current->m_fd, inum, &current );
82          }
83          else
84          {
85              printf( "could not find directory %s\n", split[i] );
86              goto fail;
87          }
88      }
89
90      /*put back all the minos not in use*/
91      for( i = 0; i < num_splits; i ++ )
92      {
93          if( to_free[i] )
94              k_t_put_mino( to_free[i] );
95      }
96
97      /*get the inum of the child before it is deleted*/
98      inum = k_t_find_dir_num( current, bn );
99
100     /*remove the child from the data blocks*/
101     if( !k_t_remove_child( current, bn ) )
102     {
103         printf( "error: could not remove child %s from inode\n", bn );
104         goto fail;
105     }
106
107     /*decrease the link count of the inode*/
108     if( !k_t_get_mino( current->m_fd, inum, &target ) )
109     {
110         printf( "error retrieveing target inode\n" );
111         goto fail;
112     }
113
114     target->m_ino.i_links_count--;
115
116     /*this is on purpose*/
117     if( target->m_ino.i_links_count == 0 )
118     {
119         k_t_put_mino( inum );
120         k_t_del_ino_from_fs( current->m_fd, inum );
121     }
122     else
123     {
124         k_t_put_mino( inum );
125     }
126
127     /*free what is left*/
128     for( i = 0; i < num_splits; i++ )
129     {
130         if( split[i] )
131         {
132             free( split[i] );
133             split[i] = NULL;
134         }
135     }
136
137     return 0;
138
139 fail:
140
141     /*free the split*/
142     for( i = 0; i < num_splits; i++ )
143     {
144         if( split[i] )
145         {
146             free( split[i] );
147             split[i] = NULL;
```

```
148         }
149     }
150
151
152     return -1;
153 }
```

5 Level two

5.1 Cat

5.1.1 Definition

Listing 63: cat.h

```
1  /*****************************************************************************
2  file        : cd.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int cat_cmd( proc_t*, int argc, char** argv );
```

5.1.2 Implementation

Listing 64: cat.c

```
1  /*****************************************************************************
2  file        : cd.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "cat.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int cat_cmd( proc_t* pr, int argc, char** argv )
23 {
24     char dn[MAX_NAME]       = { '\0' };
25     char bn[MAX_NAME]       = { '\0' };
26     char* split[MAX_DEPTH]  = { '\0' };
27     int to_free[MAX_DEPTH]  = { 0 };
28     int i                   = 0;
29     long iterations         = 0;
30     long remainder          = 0;
31     i32 ret                 = 0;
32     int inum                = 0;
33     int num_splits          = 0;
34     i32 fd_to_cat           = 0;
35     char b[8*BLK_SZ]        = { '\0' };
36     mino_t* current         = NULL;
37     mino_t* target          = NULL;
38
39     if( argc != 2 )
40     {
41         printf( "usage: cat [file]\n" );
42         goto fail;
```

```
43      }
44
45      /*split into dirname and basename*/
46      strcpy( bn, loc_basename( argv[1] ) );
47      strcpy( dn, loc_dirname( argv[1] ) );
48
49      if( is_abs_path( bn ) )
50      {
51          current = kr->m_mino_tb[0];
52      }
53      else
54      {
55          current = pr->m_cwd;
56      }
57
58      if( !current )
59      {
60          printf( "error: couldn't find starting directory\n" );
61          goto fail;
62      }
63
64      if( !split_path( dn, MAX_DEPTH, split, &num_splits ) )
65      {
66          printf( "error splitting path\n" );
67          goto fail;
68      }
69
70      /*walk through the directories*/
71      for( i = 0; i < num_splits; i++ )
72      {
73          /*find the inode number for the current directory*/
74          inum = k_t_find_dir_num( current, split[i] );
75          if( inum != -1 )
76          {
77              /*get the next inode*/
78              to_free[i] = inum;
79              k_t_get_mino( current->m_fd, inum, &current );
80          }
81          else
82          {
83              printf( "could not find directory %s\n", split[i] );
84              goto fail;
85          }
86      }
87
88      /*free the split*/
89      for( i = 0; i < num_splits; i++ )
90      {
91          if( split[i] )
92          {
93              k_t_put_mino( to_free[i] );
94              free( split[i] );
95              split[i] = NULL;
96          }
97      }
98
99      /*current now points at the target directory*/
100     inum = k_t_find_child_by_name( current, bn );
101
102     if( inum == -1 )
103     {
104         printf( "error: file to cat does not exist\n" );
105         goto fail;
106     }
107
108     /*now open the inode num*/
109     k_t_get_mino( current->m_fd, inum, &target );
110
111     if( !target )
112     {
113         printf( "error retrieveing m_inode %i\n", inum );
114         goto fail;
115     }
116
117     /*get the fd*/
118     fd_to_cat = k_t_find_oft_entry( target );
119
120     if( fd_to_cat == -1 )
121     {
122         printf( "error: file does not exist in kft\n" );
123         goto fail;
```

```
124     }
125
126     /*seek to the beginning*/
127     k_t_lseek( fd_to_cat, 0 );
128
129     /*calculate number of bytes to read*/
130     iterations = ( target->m_ino.i_size / ( 8 * BLK_SZ ) ) + 1;
131     remainder =  target->m_ino.i_size % ( 8 * BLK_SZ );
132
133     /*loop*/
134     while( iterations != 0 )
135     {
136         /*reset the block*/
137         memset( b, 0, 8 * BLK_SZ );
138
139         /*if the size is less than the total buffer size*/
140         if( iterations == 1 )
141         {
142             if( ( ret = k_t_read( fd_to_cat, remainder, b ) ) == -1 )
143             {
144                 printf( "error reading file\n" );
145                 goto fail;
146             }
147         }
148         else
149         {
150             if( ( ret = k_t_read( fd_to_cat, 8 * BLK_SZ, b ) ) == -1 )
151             {
152                 printf( "error reading file\n" );
153                 goto fail;
154             }
155         }
156
157         for( i = 0; i < ret; i ++ )
158         {
159             printf( "%c", b[i] );
160         }
161
162         iterations --;
163
164     }
165
166     /*put it back*/
167     k_t_put_mino( inum );
168
169
170     printf( "\n\n" );
171
172     return 0;
173
174 fail:
175
176     return -1;
177 }
```

5.2 Close

5.2.1 Definition

Listing 65: close.h

```
1  /*****************************************************************************
2  file        : cd.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int close_cmd( proc_t*, int argc, char** argv );
```

5.2.2 Implementation

Listing 66: close.c

```
/*****************************************************************************
file        : cd.c
author      :
desc        :
date        : 04-01-13
 *****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "close.h"
#include "../string_funcs.h"
#include "../global_defs.h"
#include "../k_t.h"
#include "../mino_t.h"
#include "../proc_t.h"


extern k_t* kr;

int close_cmd( proc_t* pr, int argc, char** argv )
{
    int i = 0;

    if( argc != 2 )
    {
        printf( "usage: close [fd number]\n" );
        goto fail;
    }

    while( pr->m_fds[i++] != atoi( argv[1] ) && i < NUM_FDS ) {};

    if( i >= NUM_FDS )
    {
        printf( "error: file descriptor is not open by this process\n" );
        goto fail;
    }

    /*tell the kernel to close it*/
    if( !k_t_close_oft_entry( kr->m_of_tb[ atoi( argv[1] ) ]->m_minoptr ) )
    {
        printf( "error closing kernel kft entry\n" );
        goto fail;
    }

    /*put nothing there*/
    pr->m_fds[ atoi( argv[1] ) ] = 0;

    return 0;

fail:

    return -1;
}
```

5.3 Copy

5.3.1 Definition

Listing 67: write.h

```
/*****************************************************************************
file        : cd.h
author      :
desc        :
date        : 04-01-13
 *****************************************************************************/
#pragma once

#include "../global_defs.h"
#include "../proc_t.h"

int write_cmd( proc_t*, int argc, char** argv );
```

5.3.2 Implementation

Listing 68: write.c

```
1  /*****************************************************************************
2  file        : cd.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "write.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int write_cmd( proc_t* pr, int argc, char** argv )
23 {
24     int i               = 0;
25     int j               = 0;
26     i32 ret             = 0;
27     i32 remainder           = 0;
28     i32 iteration           = 0;
29     char b[8*BLK_SZ]    = { '\0' };
30
31     if( argc < 3 )
32     {
33         printf( "usage: [fd number] [string to write]\n" );
34         goto fail;
35     }
36
37     if( atoi( argv[1] ) > 8 * BLK_SZ )
38     {
39         printf( "error: size too large, max is %i\n", 8 * BLK_SZ );
40         goto fail;
41     }
42
43     iteration = (strlen(argv[2]) / (8 * BLK_SZ) ) + 1;
44     remainder = strlen(argv[2]) % (8 * BLK_SZ);
45
46     printf("%d rem %d it\n", remainder, iteration);
47     while(iteration != 0)
48     {
49         memset(b, 0 , 8*BLK_SZ);
50         if( iteration == 1)
51         {
52             for(i = 0; i < remainder; i++)
53                 b[i] = argv[2][(j * 8 * BLK_SZ) + i];
54             if( ( ret += k_t_write( atoi( argv[1] ), remainder , b ) ) == -1 )
55             {
56                 printf( "error writing file\n" );
57                 goto fail;
58             }
59             iteration--;
60         }
61         else
62         {
63             for(i = 0; i < 8 * BLK_SZ; i++)
64                 b[i] = argv[2][(j * 8 * BLK_SZ)+i];
65             if( ( ret += k_t_write( atoi( argv[1] ), 8*BLK_SZ , b ) ) == -1 )
66             {
67                 printf( "error writing file\n" );
68                 goto fail;
69             }
70             j++;
71             iteration--;
72         }
73     }
74
75     printf( "bytes written: %i from fd %i\n", ret, atoi( argv[1] ) );
76
```

```
77
78      return 0;
79
80  fail:
81
82      return -1;
83  }
```

5.4 Seek

5.4.1 Definition

Listing 69: lseek.h

```
1   /*****************************************************************************
2   file        : cd.h
3   author      :
4   desc        :
5   date        : 04-01-13
6    *****************************************************************************/
7   #pragma once
8
9   #include "../global_defs.h"
10  #include "../proc_t.h"
11
12  int lseek_cmd( proc_t*, int argc, char** argv );
```

5.4.2 Implementation

Listing 70: lseek.c

```
1   /*****************************************************************************
2   file        : cd.c
3   author      :
4   desc        :
5   date        : 04-01-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <stdio.h>
10  #include <string.h>
11
12  #include "lseek.h"
13  #include "../string_funcs.h"
14  #include "../global_defs.h"
15  #include "../k_t.h"
16  #include "../mino_t.h"
17  #include "../proc_t.h"
18
19
20  extern k_t* kr;
21
22  int lseek_cmd( proc_t* pr, int argc, char** argv )
23  {
24      long o_loc      = 0;
25
26      if( argc != 3 )
27      {
28          printf( "usage: lseek [fd number] [position]\n" );
29          goto fail;
30      }
31
32      /*alread wrote this command in the kernel*/
33      if( ( o_loc = k_t_lseek( atoi( argv[1] ), atoi( argv[2] ) ) == -1 ) )
34          goto fail;
35
36      return o_loc;
37
38  fail:
39      return -1;
40  }
```

5.5 Move

5.5.1 Definition

Listing 71: mv.h

```
1  /*****************************************************************************
2  file        : mv.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int mv_cmd( proc_t*, int argc, char** argv );
```

5.5.2 Implementation

Listing 72: mv.c

```
1  /*****************************************************************************
2  file        : mv.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "mv.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int mv_cmd( proc_t* pr, int argc, char** argv )
23 {
24     char bn_1[MAX_PATH]     = { '\0' };     //fist basename
25     char dn_1[MAX_PATH]     = { '\0' };     //first dirname
26     char bn_2[MAX_PATH]     = { '\0' };     //second basename
27     char dn_2[MAX_PATH]     = { '\0' };     //second dirname
28     char name[MAX_NAME]     = { '\0' };
29     char* split[MAX_DEPTH]  = { '\0' };
30     int to_free[MAX_DEPTH]  = { 0 };
31     int i                   = 0;
32     i32 inum                = 0;
33     i32 target_inum         = 0;
34     int num_splits          = 0;
35     char b[BLK_SZ]          = { '\0' };
36     char* cp                = NULL;
37     ext2_dir* dp            = NULL;
38     mino_t* current         = NULL;
39     mino_t* dst             = NULL;
40
41     if( argc != 3 )
42     {
43         printf( "usage: mv [src] [dst]\n" );
44         goto fail;
45     }
46
47     /*get first and second basenames*/
48     strcpy( bn_1, loc_basename( argv[1] ) );
49     strcpy( dn_1, loc_dirname( argv[1] ) );
50     strcpy( bn_2, loc_basename( argv[2] ) );
51     strcpy( dn_2, loc_dirname( argv[2] ) );
52
53     /*navigate*/
54     if( is_abs_path( dn_1 ) )
55     {
```

```
56          current = kr->m_mino_tb[0];
57      }
58      else
59      {
60          current = pr->m_cwd;
61      }
62
63      if( is_abs_path( dn_2 ) )
64      {
65          dst = kr->m_mino_tb[0];
66      }
67      else
68      {
69          dst = pr->m_cwd;
70      }
71
72      /*current and dst now point at the starting directories*/
73      if( !current || !dst )
74      {
75          printf( "error: couldn't find starting directory\n" );
76          goto fail;
77      }
78
79      /*split the first path*/
80      if( !split_path( dn_1, MAX_DEPTH, split, &num_splits ) )
81      {
82          printf( "error splitting path one\n" );
83          goto fail;
84      }
85
86      /*walk through the directories*/
87      for( i = 0; i < num_splits; i++ )
88      {
89          /*find the inode number for the current directory*/
90          inum = k_t_find_dir_num( current, split[i] );
91          if( inum != -1 )
92          {
93              /*get the next inode*/
94              to_free[i] = inum;
95              k_t_get_mino( current->m_fd, inum, &current );
96          }
97          else
98          {
99              printf( "could not find directory %s\n", split[i] );
100             goto fail;
101         }
102     }
103
104     /*put back all the minos not in use*/
105     for( i = 0; i < num_splits; i ++ )
106     {
107         if( to_free[i] )
108             k_t_put_mino( to_free[i] );
109     }
110
111     /*free the first split*/
112     for( i = 0; i < num_splits; i++ )
113     {
114         if( split[i] )
115         {
116             free( split[i] );
117             split[i] = NULL;
118         }
119     }
120
121     /*put back all the minos not in use*/
122     for( i = 0; i < num_splits; i ++ )
123     {
124         if( to_free[i] )
125             k_t_put_mino( to_free[i] );
126     }
127
128     /*current is now our relative dir so find the inode we want*/
129     k_t_get_blk( b, pr->m_cwd->m_fd, current->m_ino.i_block[0] );
130
131     dp = (ext2_dir*)b;
132     cp = b;
133
134     /*walk*/
135     while( true )
136     {
```

```
137         /*have to do this in case of names that contain substrings*/
138         memset( name, 0, MAX_NAME );
139         memcpy( name, dp->name, dp->name_len );
140 
141         /*found it*/
142         if( !strcmp( name, bn_1 ) )
143             break;
144 
145         if( cp >= ( b + 1024 ) )
146         {
147             printf( "could not find file\n" );
148             goto fail;
149         }
150 
151         cp += dp->rec_len;
152         dp = (ext2_dir*)cp;
153     }
154 
155     /*get dis inode*/
156     target_inum = dp->inode;
157 
158     if( !target_inum )
159         goto fail;
160 
161 
162     /*split path two*/
163     if( !split_path( dn_2, MAX_DEPTH, split, &num_splits ) )
164     {
165         printf( "error splitting path two\n" );
166         goto fail;
167     }
168 
169     /*walk through the directories*/
170     memset( to_free, 0, MAX_PATH );
171 
172     for( i = 0; i < num_splits; i++ )
173     {
174         /*find the inode number for the current directory*/
175         inum = k_t_find_dir_num( dst, split[i] );
176         if( inum != -1 )
177         {
178             /*get the next inode*/
179             to_free[i] = inum;
180             k_t_get_mino( dst->m_fd, inum, &dst );
181         }
182         else
183         {
184             printf( "could not find target %s\n", split[i] );
185             goto fail;
186         }
187     }
188 
189     /*put back all the minos not in use*/
190     for( i = 0; i < num_splits; i ++ )
191     {
192         if( to_free[i] )
193             k_t_put_mino( to_free[i] );
194     }
195 
196     /*dst now points at the target directory*/
197     memset( b, 0, BLK_SZ );
198     k_t_get_blk( b, pr->m_cwd->m_fd, dst->m_ino.i_block[0] );
199 
200     /*show some stuff*/
201     printf( "current inode is %i\n", current->m_ino_num );
202     printf( "dst inode is %i\n", dst->m_ino_num );
203 
204     dp = (ext2_dir*)b;
205     cp = b;
206 
207     /*walk*/
208     while( cp + dp->rec_len < ( b + 1024 ) )
209     {
210         /*have to do this in case of names that contain substrings*/
211         memset( name, 0, MAX_NAME );
212         memcpy( name, dp->name, dp->name_len );
213 
214         /*found it*/
215         if( !strcmp( name, bn_2 ) )
216         {
217             printf( "error: target already exists\n" );
```

```
218             goto fail;
219         }
220
221         cp += dp->rec_len;
222         dp = (ext2_dir*)cp;
223     }
224
225     /*dst now points at the target directory*/
226     if( !k_t_add_child( dst, target_inum, bn_2 ) )
227     {
228         printf( "error adding child\n" );
229         goto fail;
230     }
231
232     /*remove the child from the parent*/
233     if( !k_t_remove_child( current, bn_1 ) )
234     {
235         printf( "error removing child from parent\n" );
236         goto fail;
237     }
238
239     /*free the second split*/
240     for( i = 0; i < num_splits; i++ )
241     {
242         if( split[i] )
243         {
244             free( split[i] );
245             split[i] = NULL;
246         }
247     }
248
249     return 0;
250
251 fail:
252
253     /*free the first split*/
254     for( i = 0; i < num_splits; i++ )
255     {
256         if( split[i] )
257         {
258             free( split[i] );
259             split[i] = NULL;
260         }
261     }
262
263
264     return -1;
265 }
```

5.6 Open

5.6.1 Definition

Listing 73: open.h

```
1  /*****************************************************************************
2  file        : cd.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int open_cmd( proc_t*, int argc, char** argv );
```

5.6.2 Implementation

Listing 74: open.c

```
1  /*****************************************************************************
2  file        : cd.c
3  author      :
4  desc        :
```

```
5  date        : 04-01-13
6   ******************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "open.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int open_cmd( proc_t* pr, int argc, char** argv )
23 {
24     char dn[MAX_NAME]       = { '\0' };
25     char bn[MAX_NAME]       = { '\0' };
26     char* split[MAX_DEPTH]  = { '\0' };
27     int to_free[MAX_DEPTH]  = { 0 };
28     int num_splits          = 0;
29     int i                   = 0;
30     int inum                = 0;
31     int fd_returned         = 0;
32     int proc_loc            = 0;
33     E_FILE_MODE mode        = 0;
34     mino_t* current         = NULL;
35     mino_t* target          = NULL;
36
37
38     if( !pr || !argc || !argv )
39         goto fail;
40
41     if( argc != 3 )
42     {
43         printf( "usage: open [file] [r|w|rw|a]\n" );
44         goto fail;
45     }
46
47     /*figure out the mode*/
48     if( !strcmp( argv[2], "r" ) )
49     {
50         mode = M_READ;
51     }
52     else if( !strcmp( argv[2], "w" ) )
53     {
54         mode = M_WRITE;
55     }
56     else if( !strcmp( argv[2], "rw" ) )
57     {
58         mode = M_READWRITE;
59     }
60     else if( !strcmp( argv[2], "a" ) )
61     {
62         mode = M_APPEND;
63     }
64     else
65     {
66         printf( "invalid mode\n" );
67         goto fail;
68     }
69
70
71
72     /*split into dirname and basename*/
73     strcpy( bn, loc_basename( argv[1] ) );
74     strcpy( dn, loc_dirname( argv[1] ) );
75
76     if( is_abs_path( bn ) )
77     {
78         current = kr->m_mino_tb[0];
79     }
80     else
81     {
82         current = pr->m_cwd;
83     }
84
85     if( !current )
```

```
86      {
87          printf( "error: couldn't find starting directory\n" );
88          goto fail;
89      }
90
91      if( !split_path( dn, MAX_DEPTH, split, &num_splits ) )
92      {
93          printf( "error splitting path\n" );
94          goto fail;
95      }
96
97      /*walk through the directories*/
98      for( i = 0; i < num_splits; i++ )
99      {
100         /*find the inode number for the current directory*/
101         inum = k_t_find_dir_num( current, split[i] );
102         if( inum != -1 )
103         {
104             /*get the next inode*/
105             to_free[i] = inum;
106             k_t_get_mino( current->m_fd, inum, &current );
107         }
108         else
109         {
110             printf( "could not find directory %s\n", split[i] );
111             goto fail;
112         }
113     }
114
115     /*free the split*/
116     for( i = 0; i < num_splits; i++ )
117     {
118         if( split[i] )
119         {
120             k_t_put_mino( to_free[i] );
121             free( split[i] );
122             split[i] = NULL;
123         }
124     }
125
126
127     /*current now points at the target directory*/
128     inum = k_t_find_child_by_name( current, bn );
129
130     if( inum == -1 )
131     {
132         printf( "error: file to open does not exist\n" );
133         goto fail;
134     }
135
136     /*now open the inode num*/
137     k_t_get_mino( current->m_fd, inum, &target );
138
139     if( !target )
140     {
141         printf( "error retrieveing m_inode %i\n", inum );
142         goto fail;
143     }
144
145     /*add it to the kft*/
146     if( !k_t_open_oft_entry( target, mode, &fd_returned, argv[1] ) )
147     {
148         printf( "error adding inode to oft\n" );
149         goto fail;
150     }
151
152     /*add it to the process*/
153     if( ( proc_loc = proc_t_get_next_fd_loc( pr ) ) == -1 )
154     {
155         printf( "error adding inode to oft\n" );
156         goto fail;
157     }
158
159     pr->m_fds[proc_loc] = fd_returned;
160
161     return 0;
162
163 fail:
164     return -1;
165 }
```

5.7 Print file descriptors

5.7.1 Definition

Listing 75: pfd.h

```
1  /*****************************************************************************
2  file        : pfd.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int pfd_cmd( proc_t*, int argc, char** argv );
```

5.7.2 Implementation

Listing 76: pfd.c

```
1  /*****************************************************************************
2  file        : pfd.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "pfd.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int pfd_cmd( proc_t* pr, int argc, char** argv )
23 {
24     int i                   = 0;
25
26     if( !kr )
27         goto fail;
28
29     printf( "open file descriptor information\n" );
30     printf( "\tfd num\t\tmino num\tmino ptr\tref count\toffset\t\tmode\t\tname\n" );
31
32     for( i = 0; i < MAX_OFS; i ++ )
33     {
34         if( kr->m_of_tb[i] && kr->m_of_tb[i]->m_minoptr )
35         {
36
37             printf( "\t%i\t\t%i\t\t%p\t%i\t\t%li\t\t%i\t\t%s\n",
38                     i,
39                     kr->m_of_tb[i]->m_minoptr->m_ino_num,
40                     kr->m_of_tb[i]->m_minoptr,
41                     kr->m_of_tb[i]->m_refc,
42                     kr->m_of_tb[i]->m_off,
43                     kr->m_of_tb[i]->m_mode,
44                     kr->m_of_tb[i]->m_name );
45         }
46     }
47
48     printf( "\n" );
49
50     return 0;
51
52 fail:
53     return -1;
54 }
```

5.8 Read

5.8.1 Definition

Listing 77: read.h

```
1  /*****************************************************************************
2  file        : cd.h
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7  #pragma once
8
9  #include "../global_defs.h"
10 #include "../proc_t.h"
11
12 int read_cmd( proc_t*, int argc, char** argv );
```

5.8.2 Implementation

Listing 78: read.c

```
1  /*****************************************************************************
2  file        : cd.c
3  author      :
4  desc        :
5  date        : 04-01-13
6   *****************************************************************************/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11
12 #include "read.h"
13 #include "../string_funcs.h"
14 #include "../global_defs.h"
15 #include "../k_t.h"
16 #include "../mino_t.h"
17 #include "../proc_t.h"
18
19
20 extern k_t* kr;
21
22 int read_cmd( proc_t* pr, int argc, char** argv )
23 {
24     i32 ret             = 0;
25     char b[8*BLK_SZ]    = { '\0' };
26
27     if( argc != 3 )
28     {
29         printf( "usage: [fd number] [bytes to read]\n" );
30         goto fail;
31     }
32
33     if( atoi( argv[1] ) > 8 * BLK_SZ )
34     {
35         printf( "error: size too large, max is %i\n", 8 * BLK_SZ );
36         goto fail;
37     }
38
39     if( ( ret = k_t_read( atoi( argv[1] ), atoi( argv[2] ), b ) ) == -1 )
40     {
41         printf( "error reading file\n" );
42         goto fail;
43     }
44
45     printf( "bytes read: %i from fd %i\n", ret, atoi( argv[1] ) );
46
47     /*for( i = 0; i < ret; i ++ )
48     {
49         printf( "%c", b[i] );
50     }
51
52     printf( "\n\n" );*/
53
54     return 0;
55
```

```
56  fail:
57
58      return -1;
59  }
```

5.9 Write

5.9.1 Definition

Listing 79: write.h

```
1   /*****************************************************************************
2   file        : cd.h
3   author      :
4   desc        :
5   date        : 04-01-13
6    *****************************************************************************/
7   #pragma once
8
9   #include "../global_defs.h"
10  #include "../proc_t.h"
11
12  int write_cmd( proc_t*, int argc, char** argv );
```

5.9.2 Implementation

Listing 80: write.c

```
1   /*****************************************************************************
2   file        : cd.c
3   author      :
4   desc        :
5   date        : 04-01-13
6    *****************************************************************************/
7
8   #include <stdlib.h>
9   #include <stdio.h>
10  #include <string.h>
11
12  #include "write.h"
13  #include "../string_funcs.h"
14  #include "../global_defs.h"
15  #include "../k_t.h"
16  #include "../mino_t.h"
17  #include "../proc_t.h"
18
19
20  extern k_t* kr;
21
22  int write_cmd( proc_t* pr, int argc, char** argv )
23  {
24      int i               = 0;
25      int j               = 0;
26      i32 ret             = 0;
27      i32 remainder           = 0;
28      i32 iteration           = 0;
29      char b[8*BLK_SZ]    = { '\0' };
30
31      if( argc < 3 )
32      {
33          printf( "usage: [fd number] [string to write]\n" );
34          goto fail;
35      }
36
37      if( atoi( argv[1] ) > 8 * BLK_SZ )
38      {
39          printf( "error: size too large, max is %i\n", 8 * BLK_SZ );
40          goto fail;
41      }
42
43      iteration = (strlen(argv[2]) / (8 * BLK_SZ) ) + 1;
44      remainder = strlen(argv[2]) % (8 * BLK_SZ);
45
46      printf("%d rem %d it\n", remainder, iteration);
47      while(iteration != 0)
48      {
```

```
49          memset(b, 0 , 8*BLK_SZ);
50          if( iteration == 1)
51          {
52              for(i = 0; i < remainder; i++)
53                  b[i] = argv[2][(j * 8 * BLK_SZ) + i];
54              if( ( ret += k_t_write( atoi( argv[1] ), remainder , b ) ) == -1 )
55              {
56                  printf( "error writing file\n" );
57                  goto fail;
58              }
59              iteration--;
60          }
61          else
62          {
63              for(i = 0; i < 8 * BLK_SZ; i++)
64                  b[i] = argv[2][(j * 8 * BLK_SZ)+i];
65              if( ( ret += k_t_write( atoi( argv[1] ), 8*BLK_SZ , b ) ) == -1 )
66              {
67                  printf( "error writing file\n" );
68                  goto fail;
69              }
70              j++;
71              iteration--;
72          }
73      }
74
75      printf( "bytes written: %i from fd %i\n", ret, atoi( argv[1] ) );
76
77
78      return 0;
79
80  fail:
81
82      return -1;
83  }
```