PennOS Documentation 24sp

Generated by Doxygen 1.10.0

Chapter 1

PennOS

1.1 Author Information

Authors: Michael Alfano (malfano 48925487) Vedansh Goenka (vedanshg 27727621) Arjun Arasappan (arjaras 14138764) Rohan Shah (rohsha 65780142)

1.2 Submitted Source Files

1.2.1 Top Level

- src/logging.c
- src/logging.h
- src/pcb.h
- src/pennos.h
- src/sys_call.c
- src/sys_call.h

1.2.2 PennFAT

- src/pennfat.c
- src/pennfat.h
- src/pennfat/fs_utils.c
- src/pennfat/fs_utils.h
- src/pennfat/k_fd_structs.h
- src/pennfat/k_fs_structs.h
- src/pennfat/k_pennfat.c
- src/pennfat/k_pennfat.c

- src/pennfat/u_pennfat.h
- src/pennfat/s_pennfat.c
- src/pennfat/s_pennfat.h
- src/pennfat/u_pennfat.c
- src/pennfat/u_pennfat.h
- src/pennfat/u_test_suite.c
- src/pennfat/u_test_suite.h
- src/test_routines.c
- src/test_routines.h

1.2.3 Kernel

- src/kernel.c
- src/kernel.h
- src/pennkernel/pcb_deque.c
- src/pennkernel/pcb_deque.h
- src/pennkernel/pid_deque.c
- src/pennkernel/pid_deque.h

1.2.4 Shell

- src/shell/shell_func.c
- src/shell/shell_func.h
- src/shell/stress.c
- src/shell/stress.h

1.2.5 Util

- src/util/spthread.c

1.3 Compilation Instructions

Perform `make` at the top level in the code heirarchy to compile the code.

1.3.1 Penn OS

To execute Penn OS enter the following at the top level after making:

```
./bin/pennos
```

1.3.2 PennFAT

To execute PennFAT enter the following at the top level after making:

```
./bin/pennfat
```

1.3.3 Overview of Work Accomplished

Over the course of this project we designed and implemented PennOS, a simulated os that can handle many aspects of OS functionality. This includes process management, file system interactions (PennFAT), and basic shell operations.

1.3.3.1 Key accomplishments:

- **PennFAT Implementation**: Developed a fully functional file allocation table (FAT) based file system (Penn↩ FAT). PennFAT supports file creation, deletion, reading, and writing, along with permission handling, and other expected functionality of a file system.
- **Kernel Implementation**: Kernel capabilities to manage processes with PCB and PID management systems. This enables process scheduling and management.
- **Shell Functionality**: Built a shell that allows for process interaction, file system navigation, and testing with custom-built commands and scripts.
- **Testing Framework**: Developed testing suites for both k-level and u-level functions to make sure the OS components functioned properly.

This project gave us a solid foundation of operating systems.

1.3.4 Description of Code and Code Layout

Our codebase is structured into several sections:

1.3.4.1 PennFAT

- **Files**: Includes files such as `pennfat.c` and `pennfat.h` which have the high level logic for enabline file system operations.
- **Utilities**: Includes files such as `fs_utils.c` and `fs_utils.h` for lower-level file system operations and helper functions.
- **Data Structures**: Includes files such as `k_fd_structs.h` and `k_fs_structs.h` which define the data models for file descriptors and file system metadata.
- **Testing Suite**: Includes files such as `u_test_suite.c` and `u_test_suite.h` which are used to test file system integrity and functionality.

1.3.4.2 PennKernel

- **Process Management**: Includes `pcb_deque.c` and `pcb_deque.h` which manage the deque operations for process control blocks used for process scheduling. Refer to `kernel.c` for integration details.
- **PID Management**: Includes `pid_deque.c` and `pid_deque.h` which are used for handling process identifiers aiding in process management. See `kernel.c` for usage examples.
- **Logging**: The `logging.c` file is utilized within the kernel for logging various system events and process states, enhancing debugging and system monitoring.

1.3.4.3 Shell

- **Core Functionality**: The files `shell_func.c` and `shell_func.h` provide the essential framework and functions for shell operations, enabling command parsing, execution, and signal handling.
- **Command Execution**: The shell supports a variety of built-in commands and the ability to execute custom scripts as detailed in `pennos.c`. It handles command execution with process management, including foreground and background execution.
- **Signal Handling**: Signal management for processes is implemented in `sys_call.c`, allowing the shell to send signals like SIGSTOP, SIGCONT, and SIGTERM to control processes.
- **Error Handling**: Robust error handling mechanisms are integrated within the shell to manage and log errors during command execution and file operations.

1.3.4.4 Utility

- **Threads**: The `spthread.c` file.

1.3.4.5 Source Top Level

- **System Integration**: Includes files like `sys_call.c` and `sys_call.h` for handling system calls, and `logging.c` and `logging.h` for system logging.
- **Errors**: Includes centralized custom error codes with error desciption and outputing functionality.

1.3.5 General Comments

- **Documentation**: inline comments and module-level documentation are provided to help with clarity.
- **Contact**: if you have any questions please don't hesitate to email us (`malfano@seas.upenn.edu`, `vedanshg@seas.upenn.edu`, `arjaras@seas.upenn.edu`, `rohsha@sas.upenn.edu`)

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 DirectoryEntry Struct Reference

Structure representing a directory entry.

`#include <k_fs_structs.h>`

Data Fields

- char name [32]
- uint32_t size
- uint16_t firstBlock
- uint8_t type
- uint8_t perm
- time_t mtime
- char reserved [16]

4.1.1 Detailed Description

Structure representing a directory entry.

4.1.2 Field Documentation

4.1.2.1 firstBlock

`uint16_t DirectoryEntry::firstBlock`

The first block of the entry.

4.1.2.2 mtime

`time_t DirectoryEntry::mtime`

The modification time of the entry.

4.1.2.3 name

```
char DirectoryEntry::name[32]
```

The name of the entry.

4.1.2.4 perm

```
uint8_t DirectoryEntry::perm
```

The permissions of the entry.

4.1.2.5 reserved

```
char DirectoryEntry::reserved[16]
```

Reserved space for future use.

4.1.2.6 size

```
uint32_t DirectoryEntry::size
```

The size of the entry.

4.1.2.7 type

```
uint8_t DirectoryEntry::type
```

The type of the entry.

The documentation for this struct was generated from the following file:

- src/pennfat/k_fs_structs.h

4.2 KFileDescriptor Struct Reference

Represents a file descriptor in the file system.

```
#include <k_fd_structs.h>
```

Data Fields

- char name [32]
- uint16_t first_block_index
- uint32_t offset
- FileMode mode
- uint8_t ref_count

4.2.1 Detailed Description

Represents a file descriptor in the file system.

4.2.2 Field Documentation

4.2.2.1 first_block_index

```
uint16_t KFileDescriptor::first_block_index
```

Index to the file in the directory entries

4.2.2.2 mode

```
FileMode KFileDescriptor::mode
```

Mode as defined above

4.2.2.3 name

```
char KFileDescriptor::name[32]
```

Name of the file

4.2.2.4 offset

```
uint32_t KFileDescriptor::offset
```

Offset within the file

4.2.2.5 ref_count

```
uint8_t KFileDescriptor::ref_count
```

Reference count

The documentation for this struct was generated from the following file:

- src/pennfat/k_fd_structs.h

4.3 parsed_command Struct Reference

```
#include <parser.h>
```

Data Fields

- bool **is_background**
- bool **is_file_append**
- const char * **stdin_file**
- const char * **stdout_file**
- size_t **num_commands**
- char ** **commands** [ ]

4.3.1 Detailed Description

struct parsed_command stored all necessary information needed for penn-shell.

The documentation for this struct was generated from the following file:

- parser/parser.h

4.4 pcb Struct Reference

Data Fields

- spthread_t **thread**
- pthread_mutex_t **mutex**
- char * **process_name**
- int **pid**
- int **ppid**
- PIDDeque * **child_pids**
- int * **open_fds**
- int **num_open_fds**
- int **priority**
- int **state**
- SFileDescriptor * **file_descriptors**
- void ** **thread_args**
- int **num_args**
- int **continue_to**
- int **f0**
- int **f1**
- bool **is_foreground**
- int **signal_state**
- int **block_time**
- int **is_waiting_on**
- int **exit_status**

The documentation for this struct was generated from the following file:

- src/pcb.h

4.5 pcb_dq_node_st Struct Reference

A node in a double-ended queue that stores PCB (Process Control Block) structures.

`#include <pcb_deque.h>`

Data Fields

- pcb_t payload
- struct pcb_dq_node_st * next
- struct pcb_dq_node_st * prev

4.5.1 Detailed Description

A node in a double-ended queue that stores PCB (Process Control Block) structures.

4.5.2 Field Documentation

4.5.2.1 next

`pcb_dq_node_st::next`

Pointer to the next node in the deque.

4.5.2.2 payload

`pcb_dq_node_st::payload`

The PCB stored within the node.

4.5.2.3 prev

`pcb_dq_node_st::prev`

Pointer to the previous node in the deque.

The documentation for this struct was generated from the following file:

- src/pennkernel/pcb_deque.h

4.6 pcb_dq_st Struct Reference

A double-ended queue (deque) for managing PCBs.

`#include <pcb_deque.h>`

Data Fields

- int num_elements
- PCBDequeNode * front
- PCBDequeNode * back

4.6.1 Detailed Description

A double-ended queue (deque) for managing PCBs.

4.6.2 Field Documentation

4.6.2.1 back

`pcb_dq_st::back`

Pointer to the back node of the deque.

4.6.2.2 front

`pcb_dq_st::front`

Pointer to the front node of the deque.

4.6.2.3 num_elements

`pcb_dq_st::num_elements`

Number of elements currently in the deque.

The documentation for this struct was generated from the following file:

- src/pennkernel/pcb_deque.h

4.7 pid_dq_node_st Struct Reference

A node in a double-ended queue that stores process IDs.

`#include <pid_deque.h>`

Data Fields

- int pid
- struct pid_dq_node_st * next
- struct pid_dq_node_st * prev

4.7.1 Detailed Description

A node in a double-ended queue that stores process IDs.

4.7.2 Field Documentation

4.7.2.1 next

`pid_dq_node_st::next`

Pointer to the next node in the deque.

4.7.2.2 pid

`pid_dq_node_st::pid`

Process ID stored in the node.

4.7.2.3 prev

`pid_dq_node_st::prev`

Pointer to the previous node in the deque.

The documentation for this struct was generated from the following file:

- src/pennkernel/pid_deque.h

4.8 pid_dq_st Struct Reference

A double-ended queue (deque) for managing process IDs.

`#include <pid_deque.h>`

Data Fields

- int num_elements
- PIDDequeNode * front
- PIDDequeNode * back

4.8.1 Detailed Description

A double-ended queue (deque) for managing process IDs.

4.8.2 Field Documentation

4.8.2.1 back

`pid_dq_st::back`

Pointer to the back node of the deque.

4.8.2.2 front

`pid_dq_st::front`

Pointer to the front node of the deque.

4.8.2.3 num_elements

`pid_dq_st::num_elements`

Number of elements currently in the deque.

The documentation for this struct was generated from the following file:

- src/pennkernel/pid_deque.h

4.9 SFileDescriptor Struct Reference

Structure representing a collection of file descriptors.

`#include <local_fd_struct.h>`

Data Fields

- KFileDescriptor * fileDescriptors [MAX_P_FD]

4.9.1 Detailed Description

Structure representing a collection of file descriptors.

This structure holds an array of file descriptors.

4.9.2 Field Documentation

4.9.2.1 fileDescriptors

```
KFileDescriptor* SFileDescriptor::fileDescriptors[MAX_P_FD]
```

Array of file descriptors

The documentation for this struct was generated from the following file:

- src/pennfat/local_fd_struct.h

4.10 spthread_fwd_args_st Struct Reference

Data Fields

- pthread_fn **actual_routine**
- void * **actual_arg**
- bool **setup_done**
- pthread_mutex_t **setup_mutex**
- pthread_cond_t **setup_cond**
- spthread_meta_t * **child_meta**

The documentation for this struct was generated from the following file:

- src/util/spthread.c

4.11 spthread_meta_st Struct Reference

Data Fields

- sigset_t **suspend_set**
- volatile sig_atomic_t **state**
- pthread_mutex_t **meta_mutex**

The documentation for this struct was generated from the following file:

- src/util/spthread.c

4.12 spthread_signal_args_st Struct Reference

Data Fields

- const int **signal**
- volatile sig_atomic_t **ack**
- pthread_mutex_t **shutup_mutex**

The documentation for this struct was generated from the following file:

- src/util/spthread.c

4.13 spthread_st Struct Reference

Data Fields

- pthread_t **thread**
- spthread_meta_t * **meta**

The documentation for this struct was generated from the following file:

- src/util/spthread.h

Chapter 5

File Documentation

5.1 parser.h

```
00001 /* Penn-Shell Parser
00002    hanbangw, 21fa    */
00003
00004 #pragma once
00005
00006 #include <stdbool.h>
00007 #include <stddef.h>
00008 #include <stdio.h>
00009
00010 /* Here defines all possible parser errors */
00011 // parser encountered an unexpected file input token '<'
00012 #define UNEXPECTED_FILE_INPUT 1
00013
00014 // parser encountered an unexpected file output token '>'
00015 #define UNEXPECTED_FILE_OUTPUT 2
00016
00017 // parser encountered an unexpected pipeline token '|'
00018 #define UNEXPECTED_PIPELINE 3
00019
00020 // parser encountered an unexpected ampersand token '&'
00021 #define UNEXPECTED_AMPERSAND 4
00022
00023 // parser didn't find input filename following '<'
00024 #define EXPECT_INPUT_FILENAME 5
00025
00026 // parser didn't find output filename following '>' or '»'
00027 #define EXPECT_OUTPUT_FILENAME 6
00028
00029 // parser didn't find any commands or arguments where it expects one
00030 #define EXPECT_COMMANDS 7
00031
00036 struct parsed_command {
00037   // indicates the command shall be executed in background
00038   // (ends with an ampersand '&')
00039   bool is_background;
00040
00041   // indicates if the stdout_file shall be opened in append mode
00042   // ignore this value when stdout_file is NULL
00043   bool is_file_append;
00044
00045   // filename for redirecting input from
00046   const char* stdin_file;
00047
00048   // filename for redirecting output to
00049   const char* stdout_file;
00050
00051   // number of commands (pipeline stages)
00052   size_t num_commands;
00053
00054   // an array to a list of arguments
00055   // size of `commands` is `num_commands`
00056   char** commands[];
00057 };
00058
00086 int parse_command(const char* cmd_line, struct parsed_command** result);
00087
00088 /* This is a debugging function used for outputting a parsed command line. */
00089 void print_parsed_command(FILE* output, const struct parsed_command* cmd);
```

```
00090
00091 /* a debugging function for printing out what error was encountered */
00092 void print_parser_errcode(FILE* output, int err_code);
```

5.2 src/errors.c File Reference

Error handling functions.

```
#include "errors.h"
```

Functions

- const char * get_error_description (int error_code)

  Returns a description of the error code.
- void log_error (int error_code)

  Logs an error message to stderr.

5.2.1 Detailed Description

Error handling functions.

5.2.2 Function Documentation

5.2.2.1 get_error_description()

```
const char * get_error_description (
            int error_code )
```

Returns a description of the error code.

Parameters

error_code	The error code to describe.

Returns

A description of the error code.

5.2.2.2 log_error()

```
void log_error (
            int error_code )
```

Logs an error message to stderr.

Parameters

error_code	The error code to log.

5.3 errors.h

```
00001 #ifndef ERRORS_H
00002 #define ERRORS_H
00003
00004 #include <stdint.h>
00005 #include <stdio.h>
00006
00007 // General Errors
00008 #define ERROR -1
00009 #define FS_NOT_MOUNTED -2
00010 #define FS_ALREADY_MOUNTED -3
00011 #define FS_NOT_FOUND -4
00012 #define INVALID_OPERATION -5
00013 #define KERNEL_ERROR -6
00014
00015 // File Operation Errors
00016 #define FILE_NOT_FOUND -100
00017 #define FILE_ALREADY_EXISTS -101
00018 #define NO_FREE_FILE_DESCRIPTORS -102
00019 #define FILE_DESCRIPTOR_NOT_FOUND -103
00020 #define NO_PERMISSION -104
00021 #define INVALID_MODE -105
00022 #define NO_FREE_BLOCKS -107
00023 #define FILE_NOT_WRITABLE -108
00024 #define LOCAL_FILE_DESCRIPTOR_NOT_FOUND -109
00025 #define FILE_IN_USE -110
00026 #define FILE_UNABLE_TO_OPEN -111
00027 #define INVALID_STD_FILE_DESCRIPTOR -112
00028 #define FILE_UNABLE_TO_REMOVE -113
00029
00030 // Directory Operation Errors
00031 #define DIRECTORY_ENTRY_NOT_FOUND -200
00032 #define NO_SPACE_FOR_NEW_DIRECTORY -202
00033
00034 // Block Operation Errors
00035 #define INVALID_BLOCK_INDEX -300
00036
00037 // File Descriptor Errors
00038 #define FILE_DESCRIPTOR_IN_USE -403
00039 #define FILE_DESCRIPTOR_STD -404
00040
00041 // Input/Output Errors
00042 #define WRITE_ERROR -501
00043 #define SEEK_ERROR -502
00044
00045 // Process Errors
00046 #define LOCAL_FILE_DESCRIPTOR_TABLE_NOT_FOUND -601
00047 #define FILE_NAME_NOT_FOUND -602
00048 #define INVALID_READ_SIZE -603
00049 #define BUFFER_NOT_FOUND -604
00050 #define INVALID_WRITE_SIZE -605
00051 #define INVALID_OFFSET -606
00052 #define INVALID_WHENCE -607
00053 #define INVALID_SIGNAL -608
00054 #define NO_PROCESS_BLOCK_FOUND -609
00055 #define FAILED_TO_ALLOCATE_PROCESS -610
00056 #define FAILED_TO_CREATE_THREAD -611
00057 #define NO_RECENT_PROCESS -612
00058 #define PROCESS_NOT_FOUND -613
00059
00065 const char* get_error_description(int error_code);
00066
00071 void log_error(int error_code);
00072
00073 #endif // ERRORS_H
```

5.4 kernel.h

```
00001 #ifndef KERNEL_H
00002 #define KERNEL_H
00003
```

```
00004 #include <signal.h>
00005 #include <stdio.h>
00006 #include <stdlib.h>
00007 #include <string.h>
00008 #include <sys/time.h>
00009 #include <sys/types.h>
00010 #include <unistd.h>
00011
00012 #include "errors.h"
00013 #include "logging.h"
00014 #include "pcb.h"
00015 #include "pennkernel/pcb_deque.h"
00016 #include "util/spthread.h"
00017
00018 #define MAX_OPEN_FDS 64
00019 #define DEFAULT_PRIORITY 1
00020 #define MAX_PROCESSES 100
00021 #define QUANTUM_MSEC 100
00022
00023 #define PRIORITY_HIGH 0
00024 #define PRIORITY_MEDIUM 1
00025 #define PRIORITY_LOW 2
00026 #define NUM_PRIORITY_LEVELS 3
00027
00032 pid_t k_get_pid();
00033
00037 void k_init(void);
00038
00039 int k_number_of_pcbs();
00040
00051 pid_t k_proc_create(void* (*func)(void*),
00052                     void* argv[],
00053                     int fd0,
00054                     int fd1,
00055                     bool is_foreground,
00056                     int num_commands);
00057
00065 pid_t k_waitpid(pid_t pid, int* wstatus, bool nohang);
00066
00074 int k_signal(pid_t pid, enum signal_type signal, void* args);
00075
00079 void k_exit(void);
00080
00087 int k_nice(pid_t pid, int priority);
00088
00093 void k_sleep(unsigned int ticks);
00094
00099 void k_bring_to_front(int pid);
00100
00105 SFileDescriptor* k_get_fd_table();
00106
00111 void** k_get_args();
00112
00116 void setup_priority_queues();
00117
00121 void scheduler_tick();
00122
00127 pcb_t* choose_next_process();
00128
00133 void cleanup_pcb_resources(pcb_t* pcb);
00134
00140 pcb_t* find_pcb_by_pid(pid_t pid);
00141
00146 pcb_t** k_get_all_process_pcbs(int* count);
00147
00154 int k_bg(int pid);
00155
00162 int k_fg(int pid);
00163
00168 int k_get_f0();
00169
00174 int k_get_f1();
00175
00180 int k_get_num_args();
00181
00185 void setup_alt_queues();
00186
00191 void display_pcb_pids(PCBDeque* deque);
00192
00196 void display_all_pcb_pids();
00197
00203 bool block_pcb(pcb_t* pcb);
00204
00210 bool zombify_by_pcb(pcb_t* pcb);
00211
00215 void unblock_processes();
00216
```

```
00222 int schedule_process();
00223
00224 #endif  // KERNEL_H
```

5.5 logging.h

```
00001 #ifndef _LOGGING_H
00002 #define _LOGGING_H
00003
00004 #include <stdarg.h>
00005 #include <stdio.h>
00006 #include <stdlib.h>
00007 #include <time.h>
00008
00009 #define LOG_FILE_PATH "system_log.txt"
00010
00015 void log_init_logging(char* log_file_path);
00016
00020 void log_close_logging();
00021
00027 void log_event(const char* format, ...);
00028
00035 void log_schedule_process(int pid, int priority, const char* queue);
00036
00043 void log_create_process(int pid, int priority, const char* process_name);
00044
00051 void log_process_exit(int pid, int priority, const char* process_name);
00052
00059 void log_process_signal(int pid, int priority, const char* process_name);
00060
00067 void log_zombie_process(int pid, int priority, const char* process_name);
00068
00075 void log_orphan_process(int pid, int priority, const char* process_name);
00076
00083 void log_wait_process(int pid, int priority, const char* process_name);
00084
00092 void log_adjust_nice(int pid, int old_nice, int new_nice, const char* process_name);
00093
00100 void log_blocked_process(int pid, int priority, const char* process_name);
00101
00108 void log_unblocked_process(int pid, int priority, const char* process_name);
00109
00116 void log_stopped_process(int pid, int priority, const char* process_name);
00117
00124 void log_continue_process(int pid, int priority, const char* process_name);
00125
00126
00127 #endif  // _LOGGING_H
```

5.6 src/pcb.h File Reference

Process control block structure and related functions.

```
#include <stdint.h>
#include <util/spthread.h>
#include "./pennfat/k_fd_structs.h"
#include "./pennfat/local_fd_struct.h"
#include "pennkernel/pid_deque.h"
```

Data Structures

- struct pcb

Typedefs

- typedef struct pcb **pcb_t**

Enumerations

- enum **signal_type** { **P_SIGSTOP** , **P_SIGCONT** , **P_SIGTERM** }
- enum **process_state** {
  RUNNING , **TERMINATED** , **BLOCKED** , **ZOMBIED** ,
  STOPPED }

5.6.1 Detailed Description

Process control block structure and related functions.

5.7 pcb.h

Go to the documentation of this file.

```
00001
00006 #ifndef PCB_H
00007 #define PCB_H
00008
00009 #include <stdint.h>
00010 #include <util/spthread.h>
00011 #include "./pennfat/k_fd_structs.h"
00012 #include "./pennfat/local_fd_struct.h"
00013 #include "pennkernel/pid_deque.h"
00014
00015 // Enumeration for signal types used within the kernel
00016
00017 enum signal_type { P_SIGSTOP, P_SIGCONT, P_SIGTERM };
00018
00019 enum process_state { RUNNING, TERMINATED, BLOCKED, ZOMBIED, STOPPED };
00020
00021 typedef struct pcb {
00022   spthread_t thread;      // handle to the spthread
00023   pthread_mutex_t mutex;  // Mutex for synchronizing access
00024   char* process_name;
00025   int pid;               // process id
00026   int ppid;              // parent process id
00027   PIDDeque* child_pids;  // dynamically sized array of child process ids
00028   int* open_fds;         // array of open file descriptors
00029   int num_open_fds;      // number of open file descriptors
00030   int priority;          // priority of the process
00031   int state;             // state of the process
00032   SFileDescriptor* file_descriptors;  // array of file descriptors
00033   void** thread_args;
00034   int num_args;
00035   int continue_to;
00036   int f0;
00037   int f1;
00038   bool is_foreground;
00039   int signal_state;
00040   int block_time;     //
00041   int is_waiting_on;  // 0 if not waiting on any child, -1 if waiting on any
00042                       // child, > 0 if waiting on that child pid
00043   int exit_status;  // 0 if not exited, 1 if exited normally by calling exit(),
00044                     // 2 if stopped by signal, 3, if terminated by signal
00045
00046 } pcb_t;
00047
00048 #endif  // PCB_H
```

5.8 src/pennfat.c File Reference

The main file for the PennFAT file system.

```
#include "pennfat.h"
```

Functions

- void **setup_signal_handlers_pf** ()

  Set up signal handlers to ignore SIGINT and SIGTSTP.
- int main (int argc, char ∗argv[ ])

  Main function to initiate the PennFAT shell.
- void print_parser_errcode_new (FILE ∗output, int err_code)

  Print the error message associated with a parser error code.

5.8.1 Detailed Description

The main file for the PennFAT file system.

5.8.2 Function Documentation

5.8.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function to initiate the PennFAT shell.

Parameters

argc	The number of command line arguments.
argv	The argument vector.

Returns

Returns EXIT_SUCCESS upon the termination of the program.

5.8.2.2 print_parser_errcode_new()

```
void print_parser_errcode_new (
            FILE * output,
            int err_code )
```

Print the error message associated with a parser error code.

Parameters

output	The output stream to which the error message is written.
err_code	The error code returned by the parser function.

5.9 pennfat.h

```
00001 #ifndef PENNFAT_MAIN_H
00002 #define PENNFAT_MAIN_H
00003
00004 #include "../parser/parser.h"
00005 #include "./pennfat/k_routines.h"
00006 #include "./pennfat/s_pennfat.h"
00007 #include "./test_routines.h"
00008
00009 #include <signal.h>
00010 #include <stdio.h>
00011 #include <stdlib.h>
00012 #include <string.h>
00013 #include <unistd.h>
00014
00015 #define MAX_ARGS 32
00016 #define EXIT_SUCCESS 0
00017 #define PROMPT "pennfat> "
00018
00019 struct parsed_command;
00020
00024 void setup_signal_handlers_pf(void);
00025
00032 int main(int argc, char* argv[]);
00033
00038 char* read_line_from_terminal(void);
00039
00045 void print_parser_errcode_new(FILE* output, int err_code);
00046
00047 #endif  // PENNFAT_MAIN_H
```

5.10 src/pennfat/fs_utils.c File Reference

Create a filesystem path in the fs directory.

```
#include "fs_utils.h"
```

Functions

- char * create_fs_path (char *fs_name)

  Creates a file system path based on the given file system name.
- char * create_os_path (char *os_name)

  Creates an operating system path by concatenating the OS_PATH and the given OS name.
- void mkfs (char *fs_name, int blocks_in_fat, int block_size_config)

  Creates a PennFAT filesystem in the file named fs_name.
- uint32_t calculate_fs_size (char *fs_name)

  Calculates the size of the filesystem.
- void mount (char *fs_name)

  Mounts the filesystem by loading its FAT into memory.
- void unmount (void)

  Unmounts the currently mounted filesystem.

5.10.1 Detailed Description

Create a filesystem path in the fs directory.

Parameters

fs_name	The name of the filesystem

Returns

char∗ The path to the filesystem

5.10.2 Function Documentation

5.10.2.1 calculate_fs_size()

```
uint32_t calculate_fs_size (
            char * fs_name )
```

Calculates the size of the filesystem.

This function calculates the size of the specified filesystem file. It takes the name of the filesystem file as input and returns the size of the filesystem in bytes.

Parameters

fs_name	Name of the filesystem file to calculate the size of.

Returns

Size of the filesystem.

5.10.2.2 create_fs_path()

```
char * create_fs_path (
            char * fs_name )
```

Creates a file system path based on the given file system name.

This function takes a file system name as input and returns a dynamically allocated string representing the file system path. The returned string should be freed by the caller when it is no longer needed.

Parameters

fs_name	The name of the file system.

Returns

A dynamically allocated string representing the file system path.

5.10.2.3 create_os_path()

```
char * create_os_path (
            char * os_name )
```

Creates an operating system path by concatenating the OS_PATH and the given OS name.

This function dynamically allocates memory for the resulting path and returns it. The caller is responsible for freeing the memory when it is no longer needed.

Parameters

os_name	The name of the operating system.

Returns

A dynamically allocated string representing the operating system path.

5.10.2.4 mkfs()

```
void mkfs (
            char * fs_name,
            int blocks_in_fat,
            int block_size_config )
```

Creates a PennFAT filesystem in the file named fs_name.

This function creates a PennFAT filesystem in the specified file. It takes the name of the filesystem file, the number of blocks in the FAT region, and the configuration value for the first_block_index size as input.

Parameters

fs_name	Name of the filesystem file to create.
blocks_in_fat	Number of blocks in the FAT region.
block_size_config	Configuration value for first_block_index size.

5.10.2.5 mount()

```
void mount (
            char * fs_name )
```

Mounts the filesystem by loading its FAT into memory.

This function mounts the specified filesystem file by loading its FAT (File Allocation Table) into memory. It takes the name of the filesystem file as input.

Parameters

fs_name	Name of the filesystem file to mount.

5.10.2.6 unmount()

```
void unmount (
            void  )
```

Unmounts the currently mounted filesystem.

This function unmounts the currently mounted filesystem by releasing the memory occupied by its FAT.

5.11 fs_utils.h

```
00001 #ifndef FS_UTILS_H
00002 #define FS_UTILS_H
00003
00004 #include "k_routines.h"
00005 #include "k_fd_structs.h"
00006 #include <stdint.h>
00007
00008 #define BUFFER_SIZE 65536  // 64KB buffer
00009
00010 #define FS_PATH "./src/fs/"
00011 #define OS_PATH "./src/os/"
00012
00023 char *create_fs_path(char *fs_name);
00024
00037 char *create_os_path(char *os_name);
00038
00050 void mkfs(char *fs_name, int blocks_in_fat, int block_size_config);
00051
00062 uint32_t calculate_fs_size(char *fs_name);
00063
00073 void mount(char *fs_name);
00074
00081 void unmount(void);
00082
00083 #endif  // FS_UTILS_H
```

5.12 src/pennfat/k_fd_structs.h File Reference

File containing structures for file descriptors.

```
#include <stdint.h>
#include "k_fs_structs.h"
```

Data Structures

- struct KFileDescriptor

  Represents a file descriptor in the file system.

Macros

- #define **MAX_FD** 1024

Enumerations

- enum FileMode { F_WRITE = 0 , F_READ = 1 , F_APPEND = 2 }

  Enumeration representing different file modes.

5.12.1 Detailed Description

File containing structures for file descriptors.

5.12.2 Enumeration Type Documentation

5.12.2.1 FileMode

```
enum FileMode
```

Enumeration representing different file modes.

Enumerator

F_WRITE	Write mode.
F_READ	Read mode.
F_APPEND	Append mode.

5.13 k_fd_structs.h

Go to the documentation of this file.

```
00001
00006 #ifndef K_FD_STRUCTS_H
00007 #define K_FD_STRUCTS_H
00008
00009 #include <stdint.h>
00010 #include "k_fs_structs.h"
00011
00012 #define MAX_FD 1024
00013
00017 typedef enum {
00018   F_WRITE = 0,
00019   F_READ = 1,
00020   F_APPEND = 2
00021 } FileMode;
00022
00023
00028 typedef struct {
00029   char name[32];
00030   uint16_t first_block_index;
00031   uint32_t offset;
00032   FileMode mode;
00033   uint8_t ref_count;
00034 } KFileDescriptor;
00035
00036
00037 #endif  // K_FD_STRUCTS_H
```

5.14 src/pennfat/k_fs_structs.h File Reference

Structures for the PennFAT filesystem.

```
#include <stdint.h>
#include <time.h>
```

Data Structures

- struct DirectoryEntry

  Structure representing a directory entry.

Macros

- #define **MAX_ENTRIES** 64
- #define **MAX_BLOCKS** 32

5.14.1 Detailed Description

Structures for the PennFAT filesystem.

5.15 k_fs_structs.h

Go to the documentation of this file.

```
00001
00006 #ifndef K_FS_STRUCTS_H
00007 #define K_FS_STRUCTS_H
00008 #define MAX_ENTRIES 64
00009 #define MAX_BLOCKS 32
00010
00011 #include <stdint.h>
00012 #include <time.h>
00013
00014 /*
00015 Permissions:
00016 0: none
00017 2: write only
00018 4: read only
00019 5: read and executable (shell scripts)
00020 6: read and write
00021 7: read, write, and executable
00022 */
00023
00027 typedef struct {
00028   char name[32];
00029   uint32_t size;
00030   uint16_t firstBlock;
00031   uint8_t type;
00032   uint8_t perm;
00033   time_t mtime;
00034   char reserved[16];
00035 } DirectoryEntry;
00036
00037 #endif  // K_FS_STRUCTS_H
```

5.16 src/pennfat/k_pennfat.c File Reference

Implementation of the PennFAT kernel module.

```
#include "k_pennfat.h"
```

Functions

- uint16_t get_blocks_in_fat ()

  Retrieves the number of blocks in the FAT.
- uint16_t get_block_size_config ()

  Retrieves the block size configuration from the FAT.
- int get_block_size ()

  Calculates and retrieves the block size based on the block size configuration.
- void copy_string (char *dest, const char *src, int size)

  Copies a string from source to destination with a specified size.
- uint32_t max (uint32_t a, uint32_t b)

  Returns the maximum value between two given values.
- uint16_t combine_bytes (char lowByte, char highByte)

  Combine two bytes into one 16-bit unit to write.
- uint16_t * get_data_block_address (uint16_t fat_index)

  Get the data block at the specified index.
- uint16_t get_next_free_data_block_index ()

  Get the next free data block in the filesystem.
- uint16_t get_block_index_from_count (uint16_t first_block, uint16_t block_number)

  Retrieves the block index from the given block number in the filesystem.
- uint16_t get_last_block (uint16_t first_block)

  Retrieves the last block index in the filesystem.
- void initialize_fd_table (KFileDescriptor fdTable[ ])

  Initializes the file_desc descriptor table.
- KFileDescriptor * get_fd_by_index (uint16_t index)

  Get the next free file_desc descriptor in the filesystem.
- KFileDescriptor * get_fd_by_name (const char *name, int skip_count)

  Get the file descriptor by name.
- int get_fd_index (KFileDescriptor *fd)

  Get the index of the file descriptor.
- KFileDescriptor * get_next_free_fd ()

  Get the next free file_desc descriptor in the filesystem.
- KFileDescriptor * create_fd_from_entry (DirectoryEntry *entry, int mode)

  Create a file descriptor given the directory entry.
- int allocate_new_block (int fd)

  Allocate a new block for the file descriptor.
- uint16_t get_fd_block_count_from_offset (KFileDescriptor *fd)

  Get the block count from the offset of the file_desc descriptor.
- uint16_t get_blocks_allocated (KFileDescriptor *fd)

  Retrieves the number of blocks allocated for a given file descriptor.
- uint16_t get_current_fd_fat_index (KFileDescriptor *fd)

  Get the current block index of the file descriptor.
- uint16_t get_current_fd_block_offset (KFileDescriptor *fd)

  Get the current block offset of the file_desc descriptor.
- int close_fd (KFileDescriptor *fd)

  Close a file_desc descriptor.
- bool writeable_file (const char *name)

  Check if a file is writable.
- bool deletable_file (const char *name)

  Check if a file is deletable.
- DirectoryEntry * get_directory_entry_by_filename (const char *name)

Get the directory entry for a file_desc with the specified name.

- void clear_block (uint16_t block_index)

  Clear block.
- void recover_blocks (uint16_t first_block)

  recover blocks from the FAT.
- bool add_directory_entry (DirectoryEntry *entry)

  Add a directory entry to the filesystem.
- bool remove_directory_entry (DirectoryEntry *entry)

  Remove a directory entry from the filesystem.
- int create_file (const char *filename, int perm)

  Create a new file in the filesystem with specified permissions.
- int validate_permission_and_mode (uint8_t perm, int mode)

  Validate the permissions and mode for a file_desc descriptor.
- int k_open (const char *fname, int mode)

  Open a file_desc with the specified name and mode.
- int k_read (int fd, int n, char *buf)

  Open a file_desc with the specified name and mode.
- int k_write (int fd, const char *buf, int n)

  Write n bytes from a string to a file_desc.
- int k_close (int fd)

  Close a file_desc descriptor.
- int k_unlink (const char *fname)

  Remove a file.
- int k_lseek (int fd, int offset, int whence)

  Reposition the file pointer of a file descriptor.
- char * get_permission (uint16_t permission)

  Get the permission string based on the given permission value.
- void k_ls (const char *filename, int fd)

  List a file or all files in the current directory.
- int k_append (int fd, const char *buf, int n)

  Appends data to a file specified by the file descriptor.
- int convert_STD (int std)

  Converts the given value from STD to another format.
- int read_std (int std, int n, char *buf)

  Reads data from the specified standard input.
- int write_std (int std, int n, const char *buf)

  Writes the specified buffer to the standard output.
- char * read_line_from_terminal (void)

  Read a line of input from the terminal.
- int k_is_executable (const char *filename)

  Check if a file is executable.

5.16.1 Detailed Description

Implementation of the PennFAT kernel module.

5.16.2 Function Documentation

5.16.2.1 add_directory_entry()

```
bool add_directory_entry (
            DirectoryEntry * entry )
```

Add a directory entry to the filesystem.

Parameters

entry	The directory entry to add.

Returns

error code or 0 on success.

5.16.2.2 allocate_new_block()

```
int allocate_new_block (
            int fd )
```

Allocate a new block for the file descriptor.

This function allocates a new block for the given file descriptor. It checks if the filesystem is mounted and if the file descriptor exists. If a new block is available, it updates the FAT (File Allocation Table) to link the current block to the new block.

Parameters

fd	The file descriptor to allocate a block for.

Returns

int Returns 0 on success, a negative value on failure. -1: Filesystem not mounted. -2: No free blocks available.

5.16.2.3 clear_block()

```
void clear_block (
            uint16_t block_index )
```

Clear block.

Parameters

block_index	The block index to clear.

5.16.2.4 close_fd()

```
int close_fd (
            KFileDescriptor * fd )
```

Close a file_desc descriptor.

Parameters

fd	The file_desc descriptor to close.

5.16.2.5 combine_bytes()

```
uint16_t combine_bytes (
            char lowByte,
            char highByte )
```

Combine two bytes into one 16-bit unit to write.

Parameters

lowByte	The low byte to write.
highByte	The high byte to write.

5.16.2.6 convert_STD()

```
int convert_STD (
            int std )
```

Converts the given value from STD to another format.

This method takes an integer value in STD format and converts it to another format.

Parameters

std	The value to be converted from STD format.

Returns

The converted value in the desired format.

5.16.2.7 copy_string()

```
void copy_string (
            char * dest,
            const char * src,
            int size )
```

Copies a string from source to destination with a specified size.

Parameters

dest	The destination string.
src	The source string.
size	The size of the string to be copied.

5.16.2.8 create_fd_from_entry()

```
KFileDescriptor * create_fd_from_entry (
            DirectoryEntry * entry,
            int mode )
```

Create a file descriptor given the directory entry.

This function creates a file descriptor from a directory entry. The file descriptor contains information about the file, such as its name, first block index, mode, and offset.

Parameters

entry	The directory entry to create the file descriptor from.
mode	The mode to open the file descriptor in.

Returns

A pointer to the created file descriptor, or NULL if the filesystem is not mounted or there are no free file descriptors available.

5.16.2.9 create_file()

```
int create_file (
            const char * filename,
            int perm )
```

Create a new file in the filesystem with specified permissions.

Parameters

filename	The name of the new file.
perm	The permissions for the new file.

Returns

int 0 on success, negative value on failure.

5.16.2.10 deletable_file()

```
bool deletable_file (
            const char * name )
```

Check if a file is deletable.

This function checks if a file is deletable by verifying that no file descriptors are open for the file. It also checks for any garbage file and closes it.

Parameters

name	The name of the file to check.

Returns

true if the file is deletable, false otherwise.

5.16.2.11 get_block_index_from_count()

```
uint16_t get_block_index_from_count (
            uint16_t first_block,
            uint16_t block_number )
```

Retrieves the block index from the given block number in the filesystem.

This function takes the first block index and the block number as input and returns the corresponding block index. It traverses the filesystem's FAT (File Allocation Table) to find the block index associated with the given block number.

Parameters

first_block	The first block index in the filesystem.
block_number	The block number for which the block index needs to be retrieved.

Returns

The block index corresponding to the given block number, or -1 if the filesystem is not mounted or an invalid block index is encountered.

5.16.2.12 get_block_size()

```
int get_block_size ( )
```

Calculates and retrieves the block size based on the block size configuration.

Returns

The calculated block size.

5.16.2.13 get_block_size_config()

```
uint16_t get_block_size_config ( )
```

Retrieves the block size configuration from the FAT.

Returns

The block size configuration.

5.16.2.14 get_blocks_allocated()

```
uint16_t get_blocks_allocated (
            KFileDescriptor * fd )
```

Retrieves the number of blocks allocated for a given file descriptor.

This function returns the number of blocks allocated for the file associated with the given file descriptor.

Parameters

fd	A pointer to the file descriptor.

Returns

The number of blocks allocated for the file.

5.16.2.15 get_blocks_in_fat()

```
uint16_t get_blocks_in_fat ( )
```

Retrieves the number of blocks in the FAT.

Returns

The number of blocks in the FAT.

5.16.2.16 get_current_fd_block_offset()

```
uint16_t get_current_fd_block_offset (
            KFileDescriptor * fd )
```

Get the current block offset of the file_desc descriptor.

Parameters

fd	The file_desc descriptor to get the offset of.

Returns

uint16_t The offset of the file_desc descriptor.

5.16.2.17 get_current_fd_fat_index()

```
uint16_t get_current_fd_fat_index (
            KFileDescriptor * fd )
```

Get the current block index of the file descriptor.

This function returns the block index of the given file descriptor. The block index represents the current position of the file descriptor within the file system.

Parameters

fd	The file descriptor to get the block index of.

Returns

uint16_t The block index of the file descriptor. Returns -1 if the file system is not mounted or if an invalid block index is encountered.

5.16.2.18 get_data_block_address()

```
uint16_t * get_data_block_address (
            uint16_t fat_index )
```

Get the data block at the specified index.

This function returns a pointer to the start of the data block at the specified index.

Parameters

index	The index of the data block to get.

Returns

uint16_t* A pointer to the start of the data block.

5.16.2.19 get_directory_entry_by_filename()

```
DirectoryEntry * get_directory_entry_by_filename (
            const char * name )
```

Get the directory entry for a file_desc with the specified name.

Parameters

name	The name of the file_desc to search for.

5.16.2.20 get_fd_block_count_from_offset()

```
uint16_t get_fd_block_count_from_offset (
            KFileDescriptor * fd )
```

Get the block count from the offset of the file_desc descriptor.

This function calculates the block count of a file descriptor based on the given offset. The block count is determined by dividing the offset by the block size.

Parameters

fd	The file_desc descriptor to get the block count of.

Returns

uint16_t The block count of the file_desc descriptor. Returns -1 if the filesystem is not mounted.

5.16.2.21 get_fd_by_index()

```
KFileDescriptor * get_fd_by_index (
            uint16_t index )
```

Get the next free file_desc descriptor in the filesystem.

This function returns a pointer to the next free file_desc descriptor in the filesystem. If no free descriptors are available, it returns NULL.

Parameters

index	The index of the file_desc descriptor to retrieve.

Returns

KFileDescriptor* Pointer to the next free file_desc descriptor, or NULL if none are available.

5.16.2.22 get_fd_by_name()

```
KFileDescriptor * get_fd_by_name (
            const char * name,
            int skip_count )
```

Get the file descriptor by name.

This function searches for a file descriptor with the specified name in the filesystem's file descriptor table. If the file descriptor is found, it returns a pointer to the file descriptor. If the file descriptor is not found, it returns NULL.

Parameters

name	The name of the file descriptor to search for.

Returns

KFileDescriptor* Pointer to the file descriptor, or NULL if not found.

5.16.2.23 get_fd_index()

```
int get_fd_index (
            KFileDescriptor * fd )
```

Get the index of the file descriptor.

This function returns the index of the file descriptor in the filesystem's file descriptor table.

Parameters

fd	The file descriptor to get the index of.

Returns

int The index of the file descriptor, or -1 if the filesystem is not mounted or the file descriptor is not found.

5.16.2.24 get_last_block()

```
uint16_t get_last_block (
            uint16_t first_block )
```

Retrieves the last block index in the filesystem.

This function takes the first block index as input and returns the last block index in the filesystem. It traverses the filesystem's FAT (File Allocation Table) until it reaches the end of the chain, i.e., the block index with a value of 0xFFFF.

Parameters

first_block	The first block index in the filesystem.

Returns

The last block index in the filesystem, or -1 if the filesystem is not mounted.

5.16.2.25 get_next_free_data_block_index()

```
uint16_t get_next_free_data_block_index ( )
```

Get the next free data block in the filesystem.

This function searches for the next free data block in the filesystem and returns its index.

Returns

uint16_t The index of the next free data block, or -1 if none are available.

5.16.2.26 get_next_free_fd()

```
KFileDescriptor * get_next_free_fd ( )
```

Get the next free file_desc descriptor in the filesystem.

This function searches for the next available file_desc descriptor in the filesystem's file descriptor table.

Returns

KFileDescriptor* Pointer to the next free file_desc descriptor, or NULL if none are available.

5.16.2.27 get_permission()

```
char * get_permission (
            uint16_t permission )
```

Get the permission string based on the given permission value.

This function takes a permission value and returns a string representation of the permissions. The permission value is a 16-bit integer where each bit represents a specific permission. The returned string consists of three characters: 'r' for read permission, 'w' for write permission, and 'x' for execute permission. If a permission is not granted, a '-' character is used instead.

Parameters

permission	The permission value to convert to a string.

Returns

A dynamically allocated string representing the permissions. The caller is responsible for freeing the memory. If the allocation fails, NULL is returned.

5.16.2.28 initialize_fd_table()

```
void initialize_fd_table (
            KFileDescriptor fdTable[] )
```

Initializes the file_desc descriptor table.

This function initializes the file descriptor table by setting the name, first block index, mode, and offset for each entry. It also sets up the standard file descriptors for stdin, stdout, and stderr.

Parameters

g_fdTable	The file descriptor table to initialize.

5.16.2.29 k_append()

```
int k_append (
            int fd,
            const char * buf,
            int n )
```

Appends data to a file specified by the file descriptor.

Parameters

fd	The file descriptor of the file to append to.
buf	The buffer containing data to append.
n	The number of bytes to append from the buffer.

Returns

int The number of bytes appended, or negative on error.

5.16.2.30 k_close()

```
int k_close (
             int fd )
```

Close a file_desc descriptor.

Parameters

fd	File descriptor to close.

Returns

int 0 on success, negative value on failure.

5.16.2.31 k_is_executable()

```
int k_is_executable (
             const char * filename )
```

Check if a file is executable.

Parameters

fd	File descriptor

Returns

int 1 if executable, 0 if not, -1 if error

5.16.2.32 k_ls()

```
void k_ls (
             const char * filename,
             int fd )
```

List a file or all files in the current directory.

If filename is NULL, lists all fis first first_block_index, permissions, size, modification timestamp, and filename.

Parameters

filename	Name of the file to list, or NULL to list all files.

5.16.2.33 k_lseek()

```
int k_lseek (
             int fd,
             int offset,
             int whence )
```

Reposition the file pointer of a file descriptor.

Whence:

- F_SEEK_SET: Set file pointer to offset.
- F_SEEK_CUR: Set file pointer to current location plus offset.
- F_SEEK_END: Set file pointer to end of file plus offset.

Parameters

fd	File descriptor to seek.
offset	Offset for the file pointer.
whence	Determines how the offset affects the file pointer.

Returns

int 0 on success, negative value on failure.

5.16.2.34 k_open()

```
int k_open (
             const char * fname,
             int mode )
```

Open a file_desc with the specified name and mode.

Parameters

fname	Name of the file_desc to open.
mode	Mode to open the file_desc in.

Returns

int File descriptor on success, negative value on failure.

Mode:

F_WRITE - writing and reading, truncates if the file_desc exists, or creates it if it does not exist. Only one instance of a file_desc can be opened in F_WRITE mode; error if attempted to open a file_desc in F_WRITE mode more than once

F_READ - open the file_desc for reading only, return an error if the file_desc does not exist

F_APPEND - open the file_desc for reading and writing but does not truncate the file_desc if exists; additionally, the file_desc pointer references the end of the file_desc.

5.16.2.35 k_read()

```
int k_read (
            int fd,
            int n,
            char * buf )
```

Open a file_desc with the specified name and mode.

Parameters

fd	File descriptor to read from.
n	Number of bytes to read.
buf	Buffer to store read data.

Returns

int Number of bytes read, 0 if EOF, or negative on error.

5.16.2.36 k_unlink()

```
int k_unlink (
            const char * fname )
```

Remove a file.

Note: Cannot delete a file in use by another process. Updates necessary on file system metadata.

Parameters

fname	Name of the file to remove.

Returns

1 on success, negative value on failure.

5.16.2.37 k_write()

```
int k_write (
            int fd,
            const char * buf,
            int n )
```

Write n bytes from a string to a file_desc.

Parameters

fd	File descriptor to write to.
buf	String from which to write bytes.
n	Number of bytes to write.

Returns

int Number of bytes written, or negative on error.

5.16.2.38 max()

```
uint32_t max (
            uint32_t a,
            uint32_t b )
```

Returns the maximum value between two given values.

Parameters

a	The first value.
b	The second value.

Returns

The maximum value between a and b.

5.16.2.39 read_line_from_terminal()

```
char * read_line_from_terminal (
            void  )
```

Read a line of input from the terminal.

Reads a line of text from the terminal.

Returns

A pointer to the dynamically allocated line of input.

5.16.2.40 read_std()

```
int read_std (
            int std,
            int n,
            char * buf )
```

Reads data from the specified standard input.

This function reads `n` bytes from the standard input and stores them in the buffer pointed to by `buf`.

Parameters

std	The standard input file descriptor.
n	The number of bytes to read.
buf	Pointer to the buffer where the read data will be stored.

Returns

The number of bytes read on success, or -1 on failure.

5.16.2.41 recover_blocks()

```
void recover_blocks (
            uint16_t first_block )
```

recover blocks from the FAT.

Parameters

first_block	The first block to recover.

5.16.2.42 remove_directory_entry()

```
bool remove_directory_entry (
            DirectoryEntry * entry )
```

Remove a directory entry from the filesystem.

Parameters

entry	The directory entry to remove.

Returns

error code or 0 on success.

5.16.2.43 validate_permission_and_mode()

```
int validate_permission_and_mode (
            uint8_t perm,
            int mode )
```

Validate the permissions and mode for a file_desc descriptor.

Parameters

perm	The permissions of the file_desc descriptor.
mode	The mode to open the file_desc descriptor in.

Returns

int 0 if valid, negative value if invalid.

5.16.2.44 write_std()

```
int write_std (
            int std,
            int n,
            const char * buf )
```

Writes the specified buffer to the standard output.

This function writes the contents of the buffer pointed to by `buf` to the standard output.

Parameters

std	The standard output file descriptor.
n	The number of bytes to write.
buf	Pointer to the buffer containing the data to be written.

Returns

On success, the number of bytes written is returned. On error, -1 is returned.

5.16.2.45 writeable_file()

```
bool writeable_file (
            const char * name )
```

Check if a file is writable.

This function checks if a file is writable by verifying that no file descriptors are open for the file that are not in read mode.

Parameters

name	The name of the file to check.

Returns

true if the file is writeable, false otherwise.

5.17 k_pennfat.h

```
00001
00002 #ifndef _POSIX_C_SOURCE
00003 #define _POSIX_C_SOURCE 200809L
00004 #endif
00005
00006 #ifndef _DEFAULT_SOURCE
00007 #define _DEFAULT_SOURCE 1
00008 #endif
00009
00010 #define _XOPEN_SOURCE 700
00011
00012 #ifndef K_PENNFAT_H
00013 #define K_PENNFAT_H
00014
```

```
00015 #include "k_fd_structs.h"
00016 #include "k_fs_structs.h"
00017 #include "../errors.h"
00018
00019 #include <stdbool.h>
00020 #include <stdlib.h>
00021 #include <string.h>
00022 #include <time.h>
00023 #include "fcntl.h"
00024 #include "stdio.h"
00025 #include "unistd.h"
00026
00033 extern uint16_t* fs_fat;  // External FAT
00034
00041 extern KFileDescriptor g_fdTable[MAX_FD];     // External file_desc descriptor table
00042
00048 extern uint32_t fs_size;   // External filesystem size. (== 0 if not mounted)
00052 extern bool fs_mounted;    // External flag for filesystem mounted
00053
00054 // Define seek constants
00055 #define F_SEEK_SET 0
00056 #define F_SEEK_CUR 1
00057 #define F_SEEK_END 2
00058
00059 // Define standard file descriptors
00060 #define STDIN 0
00061 #define STDOUT 1
00062 #define STDERR 2
00063
00064 #define MAX_INPUT_SIZE 1024
00065
00066 #define ROOT_DIRECTORY_BLOCK_INDEX 1
00067
00068 // Kernel Level Functions for PennFAT
00069
00070 /*
00071 ==========================================
00072
00073              Utility Functions
00074
00075 ==========================================
00076 */
00077
00083 uint16_t get_blocks_in_fat();
00084
00090 uint16_t get_block_size_config();
00091
00098 int get_block_size();
00099
00107 void copy_string(char *dest, const char *src, int size);
00108
00116 uint32_t max(uint32_t a, uint32_t b);
00117
00124 uint16_t combine_bytes(char lowByte, char highByte);
00125
00126 /*
00127 ==========================================
00128
00129              Block Functions
00130
00131 ==========================================
00132 */
00133
00143 uint16_t *get_data_block_address(uint16_t fat_index);
00144
00154 uint16_t get_next_free_data_block_index();
00155
00171 uint16_t get_block_index_from_count(uint16_t first_block,
00172                                     uint16_t block_number);
00173
00186 uint16_t get_last_block(uint16_t first_block);
00187
00188 /*
00189 ==========================================
00190
00191        File Descriptor Functions
00192
00193 ==========================================
00194 */
00195
00205 void initialize_fd_table(KFileDescriptor fdTable[]);
00206
00222 KFileDescriptor *get_fd_by_index(uint16_t index);
00223
00236 KFileDescriptor *get_fd_by_name(const char *name, int skip_count);
00237
00248 int get_fd_index(KFileDescriptor *fd);
```

```
00249
00259 KFileDescriptor *get_next_free_fd();
00260
00273 KFileDescriptor *create_fd_from_entry(DirectoryEntry *entry, int mode);
00274
00288 int allocate_new_block(int fd);
00289
00301 uint16_t get_fd_block_count_from_offset(KFileDescriptor *fd);
00302
00311 uint16_t get_blocks_allocated(KFileDescriptor *fd);
00312
00325 uint16_t get_current_fd_fat_index(KFileDescriptor *fd);
00326
00333 uint16_t get_current_fd_block_offset(KFileDescriptor *fd);
00334
00340 int close_fd(KFileDescriptor *fd);
00341
00351 bool writeable_file(const char *name);
00352
00363 bool deletable_file(const char *name);
00364
00365 /*
00366 ============================================
00367
00368             Directory Functions
00369
00370 ============================================
00371 */
00372
00378 DirectoryEntry *get_directory_entry_by_filename(const char *name);
00379
00385 void clear_block(uint16_t block_index);
00386
00392 void recover_blocks(uint16_t first_block);
00393
00400 bool add_directory_entry(DirectoryEntry *entry);
00401
00408 bool remove_directory_entry(DirectoryEntry *entry);
00409
00410 /*
00411 ============================================
00412
00413               File Functions
00414
00415 ============================================
00416 */
00417
00425 int create_file(const char *filename, int perm);
00426
00427 /*
00428 ============================================
00429
00430           Kernel Level Functions
00431
00432 ============================================
00433 */
00434
00442 int validate_permission_and_mode(uint8_t perm, int mode);
00443
00465 int k_open(const char *fname, int mode);
00466
00475 int k_read(int fd, int n, char *buf);
00476
00485 int k_write(int fd, const char *buf, int n);
00486
00493 int k_close(int fd);
00494
00504 int k_unlink(const char *fname);
00505
00519 int k_lseek(int fd, int offset, int whence);
00520
00536 char *get_permission(uint16_t permission);
00537
00546 void k_ls(const char *filename, int fd);
00547
00556 int k_append(int fd, const char *buf, int n);
00557
00566 int convert_STD(int std);
00567
00578 int read_std(int std, int n, char *buf);
00579
00590 int write_std(int std, int n, const char *buf);
00591
00600 char *read_line_from_terminal(void);
00601
00608 int k_is_executable(const char *filename);
00609
```

```
00610 #endif  // K_PENNFAT_H
```

5.18 src/pennfat/k_routines.c File Reference

This file contains the implementation of the kernel routines.

```
#include "k_routines.h"
```

Functions

- void pf_touch (char **files)
  Creates files if they do not exist, or updates their timestamp.
- void pf_mv (char *source, char *dest)
  Renames SOURCE to DEST.
- void pf_rm (char **args)
  Removes the specified files.
- void pf_cat (char **args)
  Concatenates files and outputs them to stdout or a file.
- void pf_cp (char **args)
  Copies a file either within the PennFAT filesystem or between the filesystem and the host OS.
- void pf_chmod (char *permissions, char *filename)
  Change permissions of a file. There's no need to error if a permission being added already exists, or if a permission being removed is already not granted.
- void **pf_ls** (void)
  Lists all files in the current directory, similar to "ls -il" in bash.

Variables

- uint16_t * fs_fat
  FAT.
- uint32_t fs_size = 0
  Filesystem size.
- bool fs_mounted = false
  Flag indicating whether the filesystem is mounted.
- KFileDescriptor g_fdTable [MAX_FD]
  Structure representing a file descriptor in the kernel.

5.18.1 Detailed Description

This file contains the implementation of the kernel routines.

5.18.2 Function Documentation

5.18.2.1 pf_cat()

```
void pf_cat (
            char ** args )
```

Concatenates files and outputs them to stdout or a file.

Parameters

files	Null-terminated array of file paths to concatenate.
output_file	Optional path to the output file (NULL for stdout).
append	Whether to append to the output file (true) or overwrite it (false).

5.18.2.2 pf_chmod()

```
void pf_chmod (
            char * permissions,
            char * filename )
```

Change permissions of a file. There's no need to error if a permission being added already exists, or if a permission being removed is already not granted.

Print appropriate error message if:

- `file` is not a file that exists
- `perms` is invalid

Example Usage: chmod +x file (adds executable permission to file) Example Usage: chmod +rw file (adds read + write permissions to file) Example Usage: chmod -wx file (removes write + executable permissions from file)

5.18.2.3 pf_cp()

```
void pf_cp (
            char ** args )
```

Copies a file either within the PennFAT filesystem or between the filesystem and the host OS.

Parameters

args	Array of arguments: [source, destination, -h]. PennFAT-to-PennFAT or PennFAT-to-host.

5.18.2.4 pf_mv()

```
void pf_mv (
            char * source,
            char * dest )
```

Renames SOURCE to DEST.

Parameters

source	Path of the source file.
dest	New path (name) for the source file.

5.18.2.5 pf_rm()

```
void pf_rm (
            char ** args )
```

Removes the specified files.

Parameters

args	Null-terminated array of args including file paths to remove.

5.18.2.6 pf_touch()

```
void pf_touch (
            char ** files )
```

Creates files if they do not exist, or updates their timestamp.

Parameters

files	Null-terminated array of file paths.

5.18.3 Variable Documentation

5.18.3.1 fs_fat

```
uint16_t* fs_fat
```

FAT.

External FAT.

This variable represents the FAT used in the program.

5.18.3.2 fs_mounted

```
bool fs_mounted = false
```

Flag indicating whether the filesystem is mounted.

External flag indicating whether the filesystem is mounted.

5.18.3.3 fs_size

```
uint32_t fs_size = 0
```

Filesystem size.

External filesystem size.

This variable represents the size of the external filesystem. This includes both FAT + DATA. If the filesystem is not mounted, its value will be 0.

5.18.3.4 g_fdTable

```
KFileDescriptor g_fdTable[MAX_FD]
```

Structure representing a file descriptor in the kernel.

This structure holds information about a file descriptor in the kernel. It is used to track open files and manage file operations.

5.19 k_routines.h

```
00001 #ifndef S_PENNFAT_ROUTINES_H
00002 #define S_PENNFAT_ROUTINES_H
00003
00004 #include "fs_utils.h"
00005 #include "k_fd_structs.h"
00006 #include "k_pennfat.h"
00007 #include "s_pennfat.h"
00008
00009 #include <fcntl.h>
00010 #include <stdbool.h>
00011 #include <stdint.h>
00012 #include <stdio.h>
00013 #include <stdlib.h>
00014 #include <string.h>
00015 #include <sys/mman.h>
00016 #include <unistd.h>
00017
00022 void pf_touch(char** files);
00023
00029 void pf_mv(char* source, char* dest);
00030
00035 void pf_rm(char** args);
00036
00044 void pf_cat(char** args);
00045
00052 void pf_cp(char** args);
00053
00068 void pf_chmod(char* permissions, char* filename);
00069
00073 void pf_ls(void);
00074
00075 #endif  // S_PENNFAT_H
```

5.20 src/pennfat/local_fd_struct.h File Reference

This file contains the definition of the SFileDescriptor structure.

```
#include "k_fd_structs.h"
```

Data Structures

- struct SFileDescriptor

  Structure representing a collection of file descriptors.

Macros

- #define **MAX_P_FD** 32

5.20.1 Detailed Description

This file contains the definition of the SFileDescriptor structure.

5.21 local_fd_struct.h

Go to the documentation of this file.

```
00001
00006 #ifndef LOCAL_FD_STRUCT_H
00007 #define LOCAL_FD_STRUCT_H
00008
00009 #include "k_fd_structs.h"
00010
00011 #define MAX_P_FD 32
00012
00018 typedef struct
00019 {
00020   KFileDescriptor *fileDescriptors[MAX_P_FD];
00021 } SFileDescriptor;
00022
00023
00024 #endif // LOCAL_FD_STRUCT_H
```

5.22 src/pennfat/s_pennfat.c File Reference

PennFAT system call wrappers.

```
#include "s_pennfat.h"
```

Functions

- int get_fd_index_from_address (KFileDescriptor *fd)

  Retrieves the index of a file descriptor from its address.
- int s_open (const char *fname, int mode)

  Opens a file with the specified mode, using kernel-level functions.
- int s_read (int fd, int n, char *buf)

  Reads n bytes from a file into a buffer.
- int s_write (int fd, const char *str, int n)

  Writes n bytes from a string to a file.
- int s_append (int fd, const char *str, int n)

  Appends n bytes from a string to the end of the file.
- int s_close (int fd)

  Closes a file descriptor, cleaning up process-specific resources.
- int s_unlink (const char *fname)

  Removes a file from the filesystem.
- int s_lseek (int fd, int offset, int whence)

  Repositions the file pointer of an open file descriptor.
- void s_ls (const char *filename, int fd)

  Lists files in the current directory or a specified file's details.
- int s_is_executable (const char *filename)

  Checks if a file is executable.

5.22.1 Detailed Description

PennFAT system call wrappers.

5.22.2 Function Documentation

5.22.2.1 get_fd_index_from_address()

```
int get_fd_index_from_address (
            KFileDescriptor * fd )
```

Retrieves the index of a file descriptor from its address.

This function takes a pointer to a file descriptor and returns the index of the file descriptor in the global file descriptor table. The file descriptor table is represented by the array `g_fdTable`.

Parameters

fd	A pointer to the file descriptor.

Returns

The index of the file descriptor in the file descriptor table, or an error code if the file descriptor is NULL.

5.22.2.2 s_append()

```
int s_append (
            int fd,
            const char * str,
            int n )
```

Appends n bytes from a string to the end of the file.

Parameters

fd	File descriptor of the file to write to.
str	String containing the data to write.
n	Number of bytes to write.

Returns

int Number of bytes written, or negative on error.

5.22.2.3 s_close()

```
int s_close (
            int fd )
```

Closes a file descriptor, cleaning up process-specific resources.

Parameters

fd	File descriptor to close.

Returns

int 0 on success, negative on failure.

5.22.2.4 s_is_executable()

```
int s_is_executable (
            const char * filename )
```

Checks if a file is executable.

Parameters

filename	Name of the file to check.

Returns

int 1 if the file is executable, 0 if not.

5.22.2.5 s_ls()

```
void s_ls (
            const char * filename,
            int fd )
```

Lists files in the current directory or a specified file's details.

Parameters

filename	Name of the file to list details for, or NULL to list all files.

5.22.2.6 s_lseek()

```
int s_lseek (
            int fd,
            int offset,
            int whence )
```

Repositions the file pointer of an open file descriptor.

Parameters

fd	File descriptor to reposition.
offset	Offset for the new position.
whence	Reference point for the offset (F_SEEK_SET, F_SEEK_CUR, F_SEEK_END).

Returns

int New position on success, negative on failure.

5.22.2.7 s_open()

```
int s_open (
            const char * fname,
            int mode )
```

Opens a file with the specified mode, using kernel-level functions.

Parameters

fname	Name of the file to open.
mode	Mode in which to open the file (F_WRITE, F_READ, F_APPEND).

Returns

int File descriptor on success, negative value on error.

5.22.2.8 s_read()

```
int s_read (
            int fd,
            int n,
            char * buf )
```

Reads n bytes from a file into a buffer.

Parameters

fd	File descriptor of the file to read from.
n	Number of bytes to read.
buf	Buffer to store read data.

Returns

int Number of bytes read, 0 if EOF, negative on error.

5.22.2.9 s_unlink()

```
int s_unlink (
            const char * fname )
```

Removes a file from the filesystem.

Parameters

fname	Name of the file to remove.

Returns

int 0 on success, negative on failure.

5.22.2.10 s_write()

```
int s_write (
            int fd,
            const char * str,
            int n )
```

Writes n bytes from a string to a file.

Parameters

fd	File descriptor of the file to write to.
str	String containing the data to write.
n	Number of bytes to write.

Returns

int Number of bytes written, or negative on error.

5.23 s_pennfat.h

```
00001 #ifndef S_PENNFAT_H
00002 #define S_PENNFAT_H
00003
00004 #include "k_pennfat.h"
00005
00006 #include <stdbool.h>
00007 #include <stdint.h>
00008 #include <stdio.h>
00009 #include <sys/mman.h>
00010
00020 int get_fd_index_from_address(KFileDescriptor* fd);
00021
00029 int s_open(const char* fname, int mode);
00030
00039 int s_read(int fd, int n, char* buf);
00040
00049 int s_write(int fd, const char* str, int n);
00050
00059 int s_append(int fd, const char* str, int n);
00060
00067 int s_close(int fd);
00068
00075 int s_unlink(const char* fname);
00076
00086 int s_lseek(int fd, int offset, int whence);
00093 void s_ls(const char* filename, int fd);
00094
00101 int s_is_executable(const char* filename);
00102
00103 #endif  // S_PENNFAT_H
```

5.24 src/pennfat/u_pennfat.c File Reference

User level functions for PennFAT.

```
#include "u_pennfat.h"
```

Functions

- bool is_fd_within_table_local (int fd, SFileDescriptor *table)
  Checks if the file descriptor is within the local table.
- void add_fd_to_table (KFileDescriptor *fd, SFileDescriptor *table)
  Adds a file descriptor to the local table.
- void remove_fd_from_table (int fd, SFileDescriptor *table)
  Removes a file descriptor from the local table.
- KFileDescriptor * get_fd_address_from_index_global (int index)
  Retrieves a pointer to a global file descriptor based on its index.
- int get_fd_index_from_address_global (KFileDescriptor *fd)
  Computes the index of a global file descriptor based on its address.
- int u_open (void *argv[])
  Opens a file specified by path and mode.
- int u_read (void *argv[])
  Reads data from a file.
- int u_write (void *argv[])
  Writes data to a file.
- int u_append (void *argv[])
  Appends data to a file.
- int u_close (void *argv[])
  Closes a file descriptor.
- int u_unlink (void *argv[])
  Unlinks (deletes) a file.
- int u_lseek (void *argv[])
  Seeks to a specific position in a file based on offset and origin.
- void u_ls (void *argv[])
  Lists the contents of a directory.
- int u_is_executable (void *argv[])
  Checks if the specified file is executable.

5.24.1 Detailed Description

User level functions for PennFAT.

5.24.2 Function Documentation

5.24.2.1 add_fd_to_table()

```
void add_fd_to_table (
            KFileDescriptor * fd,
            SFileDescriptor * table )
```

Adds a file descriptor to the local table.

Parameters

fd	Pointer to the KFileDescriptor to be added.
table	Pointer to the SFileDescriptor structure where the file descriptor should be added.

5.24.2.2 get_fd_address_from_index_global()

```
KFileDescriptor * get_fd_address_from_index_global (
            int index )
```

Retrieves a pointer to a global file descriptor based on its index.

Parameters

index	The index of the file descriptor in the global table.

Returns

Pointer to the KFileDescriptor if valid index, NULL otherwise.

5.24.2.3 get_fd_index_from_address_global()

```
int get_fd_index_from_address_global (
            KFileDescriptor * fd )
```

Computes the index of a global file descriptor based on its address.

Parameters

fd	Pointer to the KFileDescriptor whose index is to be determined.

Returns

Index of the file descriptor on success, ERROR on failure.

5.24.2.4 is_fd_within_table_local()

```
bool is_fd_within_table_local (
            int fd,
            SFileDescriptor * table )
```

Checks if the file descriptor is within the local table.

Parameters

fd	The file descriptor to check.
table	Pointer to the SFileDescriptor structure representing the local file descriptor table.

Returns

true if the file descriptor is within the local table, false otherwise.

5.24.2.5 remove_fd_from_table()

```
void remove_fd_from_table (
            int fd,
            SFileDescriptor * table )
```

Removes a file descriptor from the local table.

Parameters

fd	The index of the file descriptor to be removed.
table	Pointer to the SFileDescriptor structure from which the file descriptor should be removed.

5.24.2.6 u_append()

```
int u_append (
            void * argv[] )
```

Appends data to a file.

Parameters

argv	Array containing pointers to the local file descriptor table, file descriptor, buffer containing data, and number of bytes to append.

Returns

Number of bytes appended on success, ERROR on failure.

5.24.2.7 u_close()

```
int u_close (
            void * argv[] )
```

Closes a file descriptor.

Parameters

argv	Array containing pointers to the local file descriptor table and the file descriptor to be closed.

Returns

Result code, ERROR on failure.

5.24.2.8 u_is_executable()

```
int u_is_executable (
            void * argv[] )
```

Checks if the specified file is executable.

Parameters

argv	Array containing the file name to be checked.

Returns

true if file is executable, ERROR otherwise.

5.24.2.9 u_ls()

```
void u_ls (
            void * argv[] )
```

Lists the contents of a directory.

Parameters

argv	Array containing the directory descriptor.

5.24.2.10 u_lseek()

```
int u_lseek (
            void * argv[] )
```

Seeks to a specific position in a file based on offset and origin.

Parameters

argv	Array containing pointers to the local file descriptor table, file descriptor, offset, and whence indicator.

Returns

New position from the beginning of the file on success, ERROR on failure.

5.24.2.11 u_open()

```
int u_open (
            void * argv[] )
```

Opens a file specified by path and mode.

Parameters

argv	Array containing pointers to the local file descriptor table, file name, and mode.

Returns

File descriptor on success, ERROR on failure.

5.24.2.12 u_read()

```
int u_read (
            void * argv[] )
```

Reads data from a file.

Parameters

argv	Array containing pointers to the local file descriptor table, file descriptor, number of bytes to read, and buffer to store the data.

Returns

Number of bytes read on success, ERROR on failure.

5.24.2.13 u_unlink()

```
int u_unlink (
            void * argv[] )
```

Unlinks (deletes) a file.

Parameters

argv	Array containing the file name to be deleted.

Returns

Result code, ERROR on failure.

5.24.2.14 u_write()

```
int u_write (
            void * argv[] )
```

Writes data to a file.

Parameters

argv	Array containing pointers to the local file descriptor table, file descriptor, buffer containing data, and number of bytes to write.

Returns

Number of bytes written on success, ERROR on failure.

5.25 u_pennfat.h

```
00001 #ifndef U_PENNFAT_H
00002 #define U_PENNFAT_H
00003
00004 #include <stdbool.h>
00005 #include "../pcb.h"
00006 #include "k_pennfat.h"
00007 #include "local_fd_struct.h"
00008 #include "s_pennfat.h"
00009 #include "../sys_call.h"
00010
00017 bool is_fd_within_table_local(int fd, SFileDescriptor *table);
00018
00024 void add_fd_to_table(KFileDescriptor *fd, SFileDescriptor *table);
00025
00031 void remove_fd_from_table(int fd, SFileDescriptor *table);
00032
00038 KFileDescriptor *get_fd_address_from_index_global(int index);
00039
00045 int get_fd_index_from_address_global(KFileDescriptor *fd);
00046
00052 int u_open(void *argv[]);
00053
00059 int u_read(void *argv[]);
00060
00066 int u_write(void *argv[]);
00067
00073 int u_append(void *argv[]);
00074
00080 int u_close(void *argv[]);
00081
00087 int u_unlink(void *argv[]);
00088
00094 int u_lseek(void *argv[]);
00095
00100 void u_ls(void *argv[]);
00101
00107 int u_is_executable(void *argv[]);
00108
00109 #endif  // U_PENNFAT_H
```

5.26 src/pennfat/u_test_suite.c File Reference

Implementation of U-level test suite for PennFAT.

```
#include "u_test_suite.h"
```

Functions

- void **test_u_simple_file_operations** ()

  Test simple file operations like open, write, and close.

- void **test_u_complex_file_operations** ()

  Test complex file operations including multiple files and error handling.

- void **test_u_open_close** ()

  Test the open and close operations on files.
- void **test_u_read_write** ()

  Test read and write operations on files.
- void **test_u_append** ()

  Test append operation on an existing file.
- void **test_u_unlink** ()

  Test the unlink operation to remove a file.
- void **test_u_lseek** ()

  Test the lseek operation to change the file offset.
- void **test_u_ls** ()

  Test listing files in the directory.
- int test_u_functions ()

  Run all the test functions related to file operations.

5.26.1 Detailed Description

Implementation of U-level test suite for PennFAT.

5.26.2 Function Documentation

5.26.2.1 test_u_functions()

```
int test_u_functions ( )
```

Run all the test functions related to file operations.

Returns

int Returns 0 if all tests pass, non-zero for failures.

5.27 u_test_suite.h

```
00001 #ifndef U_TEST_ROUTINES_H
00002 #define U_TEST_ROUTINES_H
00003
00004 #include "u_pennfat.h"
00005 #include "test_routines.h"
00006
00010 void test_u_simple_file_operations();
00011
00015 void test_u_complex_file_operations();
00016
00020 void test_u_open_close();
00021
00025 void test_u_read_write();
00026
00030 void test_u_append();
00031
00035 void test_u_unlink();
00036
00040 void test_u_lseek();
00041
00045 void test_u_ls();
00046
00052 int test_u_functions();
00053
00054 #endif // U_TEST_ROUTINES_H
```

5.28 src/pennkernel/pcb_deque.c File Reference

This file contains the implementation of the PCB deque.

```
#include "pcb_deque.h"
#include <stdlib.h>
#include <string.h>
```

Functions

- int PCBDeque_Size (PCBDeque *deque)

  Returns the number of elements in the PCB deque.
- PCBDeque * PCBDeque_Allocate (void)

  Allocates and initializes a new PCB deque.
- void PCBDeque_Free (PCBDeque *deque)

  Frees a PCB deque and all its contained nodes.
- void PCBDeque_Push_Front (PCBDeque *deque, pcb_t payload)

  Inserts a new PCB at the front of the deque.
- void PCBDeque_Push_Back (PCBDeque *deque, pcb_t payload)

  Inserts a new PCB at the back of the deque.
- bool PCBDeque_Peek_Front (PCBDeque *deque, pcb_t *payload_ptr)

  Retrieves the PCB at the front of the deque without removing it.
- bool PCBDeque_Peek_Back (PCBDeque *deque, pcb_t *payload_ptr)

  Retrieves the PCB at the back of the deque without removing it.
- bool PCBDeque_Pop_Front (PCBDeque *deque, pcb_t *payload_ptr)

  Removes and returns the PCB at the front of the deque.
- bool PCBDeque_Pop_Back (PCBDeque *deque, pcb_t *payload_ptr)

  Removes and returns the PCB at the back of the deque.
- bool PCBDeque_Remove_By_PID (PCBDeque *deque, pid_t pid, pcb_t *payload_ptr)

  Removes a PCB from the deque by its PID and returns the removed PCB.
- pcb_t ** PCBDeque_Get_All_PCBs (PCBDeque *deque, size_t *count)

  Retrieves all PCBs stored in the deque as an array of pointers.
- bool PCBDeque_Find_By_PID (PCBDeque *deque, pid_t pid, pcb_t *payload_ptr)

  Finds a PCB in the deque by its PID and returns the found PCB.
- void **copy_pcb_fields** (pcb_t *dest, const pcb_t *src)
- bool PCBDeque_Update_By_PID (PCBDeque *deque, const pcb_t input_pcb)

  Updates a PCB in the deque by its PID with new data.

5.28.1 Detailed Description

This file contains the implementation of the PCB deque.

5.28.2 Function Documentation

5.28.2.1 PCBDeque_Allocate()

```
PCBDeque * PCBDeque_Allocate (
            void  )
```

Allocates and initializes a new PCB deque.

Returns

Pointer to the allocated PCB deque, or NULL if allocation fails.

5.28.2.2 PCBDeque_Find_By_PID()

```
bool PCBDeque_Find_By_PID (
            PCBDeque * deque,
            pid_t pid,
            pcb_t * payload_ptr )
```

Finds a PCB in the deque by its PID and returns the found PCB.

Parameters

deque	The PCB deque.
pid	The PID of the PCB to find.
payload_ptr	Pointer to store the found PCB.

Returns

True if the PCB was found, false otherwise.

5.28.2.3 PCBDeque_Free()

```
void PCBDeque_Free (
            PCBDeque * deque )
```

Frees a PCB deque and all its contained nodes.

Parameters

deque	The PCB deque to free.

5.28.2.4 PCBDeque_Get_All_PCBs()

```
pcb_t ** PCBDeque_Get_All_PCBs (
```

```
            PCBDeque * deque,
            size_t * count )
```

Retrieves all PCBs stored in the deque as an array of pointers.

Parameters

deque	The PCB deque.
count	Pointer to store the number of PCBs retrieved.

Returns

Array of pointers to PCBs, or NULL if an error occurred.

5.28.2.5 PCBDeque_Peek_Back()

```
bool PCBDeque_Peek_Back (
            PCBDeque * deque,
            pcb_t * payload_ptr )
```

Retrieves the PCB at the back of the deque without removing it.

Parameters

deque	The PCB deque.
payload_ptr	Pointer to store the peeked PCB.

Returns

True if a PCB was successfully retrieved, false otherwise.

5.28.2.6 PCBDeque_Peek_Front()

```
bool PCBDeque_Peek_Front (
            PCBDeque * deque,
            pcb_t * payload_ptr )
```

Retrieves the PCB at the front of the deque without removing it.

Parameters

deque	The PCB deque.
payload_ptr	Pointer to store the peeked PCB.

Returns

True if a PCB was successfully retrieved, false otherwise.

5.28.2.7 PCBDeque_Pop_Back()

```
bool PCBDeque_Pop_Back (
            PCBDeque * deque,
            pcb_t * payload_ptr )
```

Removes and returns the PCB at the back of the deque.

Parameters

deque	The PCB deque.
payload_ptr	Pointer to store the popped PCB.

Returns

True if a PCB was successfully popped, false otherwise.

5.28.2.8 PCBDeque_Pop_Front()

```
bool PCBDeque_Pop_Front (
            PCBDeque * deque,
            pcb_t * payload_ptr )
```

Removes and returns the PCB at the front of the deque.

Parameters

deque	The PCB deque.
payload_ptr	Pointer to store the popped PCB.

Returns

True if a PCB was successfully popped, false otherwise.

5.28.2.9 PCBDeque_Push_Back()

```
void PCBDeque_Push_Back (
            PCBDeque * deque,
            pcb_t payload )
```

Inserts a new PCB at the back of the deque.

Parameters

deque	The PCB deque.
payload	The PCB to insert.

5.28.2.10 PCBDeque_Push_Front()

```
void PCBDeque_Push_Front (
            PCBDeque * deque,
            pcb_t payload )
```

Inserts a new PCB at the front of the deque.

Parameters

deque	The PCB deque.
payload	The PCB to insert.

5.28.2.11 PCBDeque_Remove_By_PID()

```
bool PCBDeque_Remove_By_PID (
            PCBDeque * deque,
            pid_t pid,
            pcb_t * payload_ptr )
```

Removes a PCB from the deque by its PID and returns the removed PCB.

Parameters

deque	The PCB deque.
pid	The PID of the PCB to remove.
payload_ptr	Pointer to store the removed PCB.

Returns

True if the PCB was successfully removed, false otherwise.

5.28.2.12 PCBDeque_Size()

```
int PCBDeque_Size (
            PCBDeque * deque )
```

Returns the number of elements in the PCB deque.

Parameters

deque	The PCB deque to query.

Returns

Number of elements in the deque.

5.28.2.13 PCBDeque_Update_By_PID()

```
bool PCBDeque_Update_By_PID (
            PCBDeque * deque,
            const pcb_t input_pcb )
```

Updates a PCB in the deque by its PID with new data.

Parameters

deque	The PCB deque.
input_pcb	The new PCB data to update.

Returns

True if the PCB was successfully updated, false otherwise.

5.29 pcb_deque.h

```
00001 #ifndef PCB_DEQUE_H
00002 #define PCB_DEQUE_H
00003
00004 #include <stdbool.h>
00005 #include <pthread.h>
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008
00009 #include "pcb.h"
00010
00022 typedef struct pcb_dq_node_st {
00023     pcb_t payload;
00024     struct pcb_dq_node_st *next;
00025     struct pcb_dq_node_st *prev;
00026 } PCBDequeNode;
00027
00039 typedef struct pcb_dq_st {
00040     int num_elements;
00041     PCBDequeNode *front;
00042     PCBDequeNode *back;
00043 } PCBDeque;
00044
00045 // Function declarations
00046
00051 PCBDeque* PCBDeque_Allocate(void);
00052
00057 void PCBDeque_Free(PCBDeque *deque);
00058
00064 int PCBDeque_Size(PCBDeque *deque);
00065
00071 void PCBDeque_Push_Front(PCBDeque *deque, pcb_t payload);
00072
00079 bool PCBDeque_Pop_Front(PCBDeque *deque, pcb_t *payload_ptr);
00080
00087 bool PCBDeque_Peek_Front(PCBDeque *deque, pcb_t *payload_ptr);
00088
00094 void PCBDeque_Push_Back(PCBDeque *deque, pcb_t payload);
00095
00102 bool PCBDeque_Pop_Back(PCBDeque *deque, pcb_t* payload_ptr);
00103
00110 bool PCBDeque_Peek_Back(PCBDeque *deque, pcb_t *payload_ptr);
00111
00119 bool PCBDeque_Remove_By_PID(PCBDeque *deque, pid_t pid, pcb_t *payload_ptr);
00120
00127 pcb_t** PCBDeque_Get_All_PCBs(PCBDeque* deque, size_t* count);
00128
00136 bool PCBDeque_Find_By_PID(PCBDeque* deque, pid_t pid, pcb_t* payload_ptr);
00137
00144 bool PCBDeque_Update_By_PID(PCBDeque* deque, const pcb_t input_pcb);
00145
00146 #endif  // PCB_DEQUE_H
```

5.30 src/pennkernel/pid_deque.c File Reference

Implementation of a deque data structure for process IDs.

```
#include "pid_deque.h"
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
```

Functions

- int PIDDeque_Size (PIDDeque *deque)
  Returns the number of elements in the PID deque.
- PIDDeque * PIDDeque_Allocate (void)
  Allocates and initializes a new PID deque.
- void PIDDeque_Free (PIDDeque *deque)
  Frees a PID deque and all its contained nodes.
- void PIDDeque_Push_Front (PIDDeque *deque, int pid)
  Inserts a new PID at the front of the deque.
- void PIDDeque_Push_Back (PIDDeque *deque, int pid)
  Inserts a new PID at the back of the deque.
- bool PIDDeque_Peek_Front (PIDDeque *deque, int *pid_ptr)
  Retrieves the PID at the front of the deque without removing it.
- bool PIDDeque_Peek_Back (PIDDeque *deque, int *pid_ptr)
  Retrieves the PID at the back of the deque without removing it.
- bool PIDDeque_Pop_Front (PIDDeque *deque, int *pid_ptr)
  Removes and returns the PID at the front of the deque.
- bool PIDDeque_Pop_Back (PIDDeque *deque, int *pid_ptr)
  Removes and returns the PID at the back of the deque.
- bool PIDDeque_Remove_By_PID (PIDDeque *deque, pid_t pid, int *pid_ptr)
  Removes a PID from the deque by its process ID.
- int * PIDDeque_Get_All_pids (PIDDeque *deque, size_t *count)
  Retrieves all PIDs stored in the deque.
- bool PIDDeque_Find_By_PID (PIDDeque *deque, pid_t pid, int *pid_ptr)
  Finds a PID in the deque by its process ID.

5.30.1 Detailed Description

Implementation of a deque data structure for process IDs.

5.30.2 Function Documentation

5.30.2.1 PIDDeque_Allocate()

```
PIDDeque * PIDDeque_Allocate (
            void  )
```

Allocates and initializes a new PID deque.

Returns

Pointer to the allocated PID deque, or NULL if allocation fails.

5.30.2.2 PIDDeque_Find_By_PID()

```
bool PIDDeque_Find_By_PID (
            PIDDeque * deque,
            pid_t pid,
            int * pid_ptr )
```

Finds a PID in the deque by its process ID.

Parameters

deque	The PID deque.
pid	The process ID to find.
pid_ptr	Pointer to store the found PID.

Returns

True if the PID was found, false otherwise.

5.30.2.3 PIDDeque_Free()

```
void PIDDeque_Free (
            PIDDeque * deque )
```

Frees a PID deque and all its contained nodes.

Parameters

deque	The PID deque to free.

5.30.2.4 PIDDeque_Get_All_pids()

```
int * PIDDeque_Get_All_pids (
            PIDDeque * deque,
            size_t * count )
```

Retrieves all PIDs stored in the deque.

Parameters

deque	The PID deque.
count	Pointer to store the number of PIDs retrieved.

Returns

Array of PIDs, or NULL if an error occurred.

5.30.2.5 PIDDeque_Peek_Back()

```
bool PIDDeque_Peek_Back (
            PIDDeque * deque,
            int * pid_ptr )
```

Retrieves the PID at the back of the deque without removing it.

Parameters

deque	The PID deque.
pid_ptr	Pointer to store the peeked PID.

Returns

True if a PID was successfully retrieved, false otherwise.

5.30.2.6 PIDDeque_Peek_Front()

```
bool PIDDeque_Peek_Front (
            PIDDeque * deque,
            int * pid_ptr )
```

Retrieves the PID at the front of the deque without removing it.

Parameters

deque	The PID deque.
pid_ptr	Pointer to store the peeked PID.

Returns

True if a PID was successfully retrieved, false otherwise.

5.30.2.7 PIDDeque_Pop_Back()

```
bool PIDDeque_Pop_Back (
            PIDDeque * deque,
            int * pid_ptr )
```

Removes and returns the PID at the back of the deque.

Parameters

deque	The PID deque.
pid_ptr	Pointer to store the popped PID.

Returns

True if a PID was successfully popped, false otherwise.

5.30.2.8 PIDDeque_Pop_Front()

```
bool PIDDeque_Pop_Front (
            PIDDeque * deque,
            int * pid_ptr )
```

Removes and returns the PID at the front of the deque.

Parameters

deque	The PID deque.
pid_ptr	Pointer to store the popped PID.

Returns

True if a PID was successfully popped, false otherwise.

5.30.2.9 PIDDeque_Push_Back()

```
void PIDDeque_Push_Back (
            PIDDeque * deque,
            int pid )
```

Inserts a new PID at the back of the deque.

Parameters

deque	The PID deque.
pid	The process ID to insert.

5.30.2.10 PIDDeque_Push_Front()

```
void PIDDeque_Push_Front (
            PIDDeque * deque,
            int pid )
```

Inserts a new PID at the front of the deque.

Parameters

deque	The PID deque.
pid	The process ID to insert.

5.30.2.11 PIDDeque_Remove_By_PID()

```
bool PIDDeque_Remove_By_PID (
            PIDDeque * deque,
            pid_t pid,
            int * pid_ptr )
```

Removes a PID from the deque by its process ID.

Parameters

deque	The PID deque.
pid	The process ID to remove.
pid_ptr	Pointer to store the removed PID.

Returns

True if the PID was successfully removed, false otherwise.

5.30.2.12 PIDDeque_Size()

```
int PIDDeque_Size (
            PIDDeque * deque )
```

Returns the number of elements in the PID deque.

Parameters

deque	The PID deque to query.

Returns

Number of elements in the deque.

5.31 pid_deque.h

```
00001 #ifndef PID_DEQUE_H
00002 #define PID_DEQUE_H
00003
00004 #include <stdbool.h>
00005 #include <stddef.h>
00006 #include <sys/types.h>
00007
00019 typedef struct pid_dq_node_st {
00020     int pid;
00021     struct pid_dq_node_st *next;
00022     struct pid_dq_node_st *prev;
00023 } PIDDequeNode;
00024
00036 typedef struct pid_dq_st {
00037     int num_elements;
00038     PIDDequeNode *front;
00039     PIDDequeNode *back;
00040 } PIDDeque;
00041
00042 // Function declarations
00043
```

```
00048 PIDDeque* PIDDeque_Allocate(void);
00049
00054 void PIDDeque_Free(PIDDeque *deque);
00055
00061 int PIDDeque_Size(PIDDeque *deque);
00062
00068 void PIDDeque_Push_Front(PIDDeque *deque, int pid);
00069
00076 bool PIDDeque_Pop_Front(PIDDeque *deque, int *pid_ptr);
00077
00084 bool PIDDeque_Peek_Front(PIDDeque *deque, int *pid_ptr);
00085
00091 void PIDDeque_Push_Back(PIDDeque *deque, int pid);
00092
00099 bool PIDDeque_Pop_Back(PIDDeque *deque, int* pid_ptr);
00100
00107 bool PIDDeque_Peek_Back(PIDDeque *deque, int *pid_ptr);
00108
00116 bool PIDDeque_Remove_By_PID(PIDDeque* deque, pid_t pid, int* pid_ptr);
00117
00124 int* PIDDeque_Get_All_pids(PIDDeque* deque, size_t* count);
00125
00133 bool PIDDeque_Find_By_PID(PIDDeque* deque, pid_t pid, int* pid_ptr);
00134
00135 #endif  // PID_DEQUE_H
```

5.32 src/pennos.c File Reference

The main file for the PennOS kernel.

```
#include <fcntl.h>
#include <kernel.h>
#include <logging.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys_call.h>
#include "../parser/parser.h"
#include "./pennfat/u_test_suite.h"
#include "./pennfat/u_pennfat.h"
#include "shell/shell_func.h"
#include "shell/stress.h"
```

Functions

- void **execute_commands** (char *command, char ***commands, void **args, int f0, int f1, bool is_↩ background, int num_args)
- void **custom_script** (char *filename, SFileDescriptor *fd_table, int f0, int f1, int is_background)
- void * shell_process (void *arg)
- int **main** (int argc, char **argv)

Variables

- bool **multiple_commands**

5.32.1 Detailed Description

The main file for the PennOS kernel.

5.32.2 Function Documentation

5.32.2.1 shell_process()

```
void * shell_process (
            void * arg )
```

Create the local fd table Assign f0 and f1 into the table through u_open If append, then lseek to the end of f1 (if not stdout) char* command name is [0] - pointer Pass table as [1] to the function - pointer args as [2] onwards - string pointers

5.33 src/shell/shell_func.c File Reference

Implementation of shell functions.

```
#include "shell_func.h"
```

Functions

- void **print_fd_table** (SFileDescriptor *fd_table)
- void **print_fd_table_names** (SFileDescriptor *fd_table)
- int **copy_string_array_into_buffer** (char *buffer, void **args, int num_args)
- void **close_file_not_stdout** (SFileDescriptor *fd_table, int fd)
- void * cat (void *arg)

  The ususal `cat` program.
- void * sleep_travis (void *arg)

  Sleep for `n` seconds.
- void * busy (void *arg)

  Busy wait indefinitely. It can only be interrupted via signals.
- void * echo (void *arg)

  Echo back an input string.
- void * ls (void *arg)

  Lists all files in the working directory. For extra credit, it should support relative and absolute file paths.
- void * touch (void *arg)

  For each file, create an empty file if it doesn't exist, else update its timestamp.
- void * mv (void *arg)

  Rename a file. If the `dst_file` file already exists, overwrite it.
- void * cp (void *arg)
- void * rm (void *arg)

  Remove a list of files. Treat each file in the list as a separate transaction. (i.e. if removing file1 fails, still attempt to remove file2, file3, etc.)
- void * chmod (void *arg)

  Change permissions of a file. There's no need to error if a permission being added already exists, or if a permission being removed is already not granted.
- void * ps (void *arg)

  List all processes on PennOS, displaying PID, PPID, priority, status, and command name.
- void * shell_kill (void *arg)

  Sends a specified signal to a list of processes. If a signal name is not specified, default to "term". Valid signals are -term, -stop, and -cont.

- void * nice_travis (void *arg)

  Spawn a new process for `command` and set its priority to `priority`.
- void * **nice_pid** (void *arg)
- void * man (void *arg)

  Adjust the priority level of an existing process.
- void * bg (void *args)

  Resumes the most recently stopped job in the background, or the job specified by job_id.
- void * fg (void *args)

  Brings the most recently stopped or background job to the foreground, or the job specified by job_id.
- void * jobs (void *arg)

  Lists all jobs.
- void * logout (void *arg)

  Exits the shell and shutsdown PennOS.

Variables

- bool **multiple_commands** = false

5.33.1 Detailed Description

Implementation of shell functions.

5.33.2 Function Documentation

5.33.2.1 bg()

```
void * bg (
            void * arg )
```

Resumes the most recently stopped job in the background, or the job specified by job_id.

Example Usage: bg Example Usage: bg 2 (job_id is 2)

5.33.2.2 busy()

```
void * busy (
            void * arg )
```

Busy wait indefinitely. It can only be interrupted via signals.

Example Usage: busy

5.33.2.3 cat()

```
void * cat (
            void * arg )
```

The ususal `cat` program.

If `files` arg is provided, concatenate these files and print to stdout If `files` arg is *not* provided, read from stdin and print back to stdout

Example Usage: cat f1 f2 (concatenates f1 and f2 and print to stdout) Example Usage: cat f1 f2 < f3 (concatenates f1 and f2 and prints to stdout, ignores f3) Example Usage: cat < f3 (concatenates f3, prints to stdout)

5.33.2.4 chmod()

```
void * chmod (
            void * arg )
```

Change permissions of a file. There's no need to error if a permission being added already exists, or if a permission being removed is already not granted.

Print appropriate error message if:

- `file` is not a file that exists
- `perms` is invalid

Example Usage: chmod +x file (adds executable permission to file) Example Usage: chmod +rw file (adds read + write permissions to file) Example Usage: chmod -wx file (removes write + executable permissions from file)

5.33.2.5 cp()

```
void * cp (
            void * arg )
```

Copy a file. If the `dst_file` file already exists, overwrite it.

Print appropriate error message if:

- `src_file` is not a file that exists
- `src_file` does not have read permissions
- `dst_file` file already exists but does not have write permissions

Example Usage: cp src_file dst_file

5.33.2.6 echo()

```
void * echo (
            void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

5.33.2.7 fg()

```
void * fg (
            void * arg )
```

Brings the most recently stopped or background job to the foreground, or the job specified by job_id.

Example Usage: fg Example Usage: fg 2 (job_id is 2)

5.33.2.8 jobs()

```
void * jobs (
            void * arg )
```

Lists all jobs.

Example Usage: jobs

5.33.2.9 logout()

```
void * logout (
            void * arg )
```

Exits the shell and shutsdown PennOS.

Example Usage: logout

5.33.2.10 ls()

```
void * ls (
            void * arg )
```

Lists all files in the working directory. For extra credit, it should support relative and absolute file paths.

Example Usage: ls (regular credit) Example Usage: ls ../../foo/./bar/sample (only for EC)

5.33.2.11 man()

```
void * man (
            void * arg )
```

Adjust the priority level of an existing process.

Example Usage: nice_pid 0 123 (sets priority 0 to PID 123)

Lists all available commands.

Example Usage: man

5.33.2.12 mv()

```
void * mv (
            void * arg )
```

Rename a file. If the `dst_file` file already exists, overwrite it.

Print appropriate error message if:

- `src_file` is not a file that exists
- `src_file` does not have read permissions
- `dst_file` file already exists but does not have write permissions

Example Usage: mv src_file dst_file

5.33.2.13 nice_travis()

```
void * nice_travis (
            void * arg )
```

Spawn a new process for `command` and set its priority to `priority`.

1. Adjust the priority level of an existing process.

Example Usage: nice 2 cat f1 f2 f3 (spawns cat with priority 2)

5.33.2.14 ps()

```
void * ps (
            void * arg )
```

List all processes on PennOS, displaying PID, PPID, priority, status, and command name.

Example Usage: ps

5.33.2.15 rm()

```
void * rm (
            void * arg )
```

Remove a list of files. Treat each file in the list as a separate transaction. (i.e. if removing file1 fails, still attempt to remove file2, file3, etc.)

Print appropriate error message if:

- `file` is not a file that exists

Example Usage: rm f1 f2 f3 f4 f5

5.33.2.16 shell_kill()

```
void * shell_kill (
            void * arg )
```

Sends a specified signal to a list of processes. If a signal name is not specified, default to "term". Valid signals are -term, -stop, and -cont.

Example Usage: kill 1 2 3 (sends term to processes 1, 2, and 3) Example Usage: kill -term 1 2 (sends term to processes 1 and 2) Example Usage: kill -stop 1 2 (sends stop to processes 1 and 2) Example Usage: kill -cont 1 (sends cont to process 1)

5.33.2.17 sleep_travis()

```
void * sleep_travis (
            void * arg )
```

Sleep for `n` seconds.

Note that you'll have to convert the number of seconds to the correct number of ticks.

Example Usage: sleep 10

5.33.2.18 touch()

```
void * touch (
            void * arg )
```

For each file, create an empty file if it doesn't exist, else update its timestamp.

Example Usage: touch f1 f2 f3 f4 f5

5.34 shell_func.h

```
00001 #include <fcntl.h>
00002 #include <kernel.h>
00003 #include <logging.h>
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006 #include <string.h>
00007 #include <sys_call.h>
00008 #include "../parser/parser.h"
00009 #include "../pennfat/u_pennfat.h"
00010 #include "../pennfat/k_routines.h"
00011
00012 #define BUF_SIZE 65536 // Unreasonably large buffer size solves all problems
00013
00024 void* cat(void* arg);
00025
00034 void* sleep_travis(void* arg);
00035
00042 void* busy(void* arg);
00043
00049 void* echo(void* arg);
00050
00058 void* ls(void* arg);
00059
00066 void* touch(void* arg);
00067
00078 void* mv(void* arg);
00079
00090 void* cp(void* arg);
```

```
00091
00102 void* rm(void* arg);
00103
00118 void* chmod(void* arg);
00119
00126 void* ps(void* arg);
00127
00138 void* shell_kill(void* arg);
00139
00146 void* nice_travis(void* arg);
00147
00153 // void* nice_pid(void* arg);
00154
00160 void* man(void* arg);
00161
00169 void* bg(void* arg);
00170
00178 void* fg(void* arg);
00179
00185 void* jobs(void* arg);
00186
00192 void* logout(void* arg);
```

5.35 stress.h

```
00001 #ifndef STRESS_H
00002 #define STRESS_H
00003
00004 void* hang(void*);
00005 void* nohang(void*);
00006 void* recur(void*);
00007 void* zombify(void*);
00008 void* orphanify(void*);
00009
00010 #endif
```

5.36 src/sys_call.c File Reference

Initialize the system process management.

```
#include "sys_call.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "kernel.h"
#include "logging.h"
```

Functions

- void **s_init** ()

  Initialize the system process management.
- pid_t s_spawn (void ∗(∗func)(void ∗), void ∗argv[ ], int fd0, int fd1, bool is_foreground, int num_args)

  Spawn a new process.
- pid_t s_get_pid ()

  Retrieve the process ID of the current process.
- void s_fg (int pid)

  Bring a process to the foreground.
- void s_bg (int pid)

  Send a process to the background.
- void ∗∗ s_get_args ()

Retrieve the argument list of the current process.

- SFileDescriptor * s_get_fd_table ()

  Get the file descriptor table.

- int s_get_f0 ()

  Get the file descriptor for standard input.

- int s_get_f1 ()

  Get the file descriptor for standard output.

- int s_get_num_args ()

  Get the number of arguments passed to the current process.

- pid_t s_waitpid (pid_t pid, int *wstatus, bool nohang)

  Wait for a process to change state.

- int s_kill (pid_t pid, int signal, void *args)

  Send a signal to a process.

- void **s_exit** (void)

  Terminate the calling process.

- pcb_t ** s_get_all_process_pcbs (int *count)

  Retrieve a list of all process control blocks.

- int s_nice (pid_t pid, int priority)

  Change the priority of the current process.

- void s_sleep (unsigned int ticks)

  Suspend execution for an interval of time.

- void **s_stop** ()

  Stop the system or process.

- void **s_cont** ()

  Continue a stopped process.

- int **s_number_of_pcbs** ()

Variables

- int **count** = 0

5.36.1 Detailed Description

Initialize the system process management.

5.36.2 Function Documentation

5.36.2.1 s_bg()

```
void s_bg (
            int pid )
```

Send a process to the background.

Parameters

pid	Process ID to send to the background.

5.36.2.2 s_fg()

```
void s_fg (
            int pid )
```

Bring a process to the foreground.

Parameters

pid	Process ID to bring to the foreground.

5.36.2.3 s_get_all_process_pcbs()

```
pcb_t ** s_get_all_process_pcbs (
            int * count )
```

Retrieve a list of all process control blocks.

Returns

Pointer to an array of PCB pointers.

5.36.2.4 s_get_args()

```
void ** s_get_args ( )
```

Retrieve the argument list of the current process.

Returns

Pointer to the argument list.

5.36.2.5 s_get_f0()

```
int s_get_f0 ( )
```

Get the file descriptor for standard input.

Returns

File descriptor for standard input.

5.36.2.6 s_get_f1()

```
int s_get_f1 ( )
```

Get the file descriptor for standard output.

Returns

File descriptor for standard output.

5.36.2.7 s_get_fd_table()

```
SFileDescriptor * s_get_fd_table ( )
```

Get the file descriptor table.

Returns

Pointer to the start of the file descriptor table.

5.36.2.8 s_get_num_args()

```
int s_get_num_args ( )
```

Get the number of arguments passed to the current process.

Returns

Number of arguments.

5.36.2.9 s_get_pid()

```
pid_t s_get_pid ( )
```

Retrieve the process ID of the current process.

Returns

The process ID (PID) of the calling process.

5.36.2.10 s_kill()

```
int s_kill (
            pid_t pid,
            int signal,
            void * args )
```

Send a signal to a process.

Parameters

pid	Process ID to which the signal should be sent.
signal	Signal number to send.
args	Additional arguments if needed.

Returns

Status of operation (0 for success, -1 for failure).

5.36.2.11 s_nice()

```
int s_nice (
            pid_t pid,
            int priority )
```

Change the priority of the current process.

Parameters

pid	Process ID to change priority of.
priority	New priority value.

Returns

Status of operation (0 for success, -1 for failure).

5.36.2.12 s_sleep()

```
void s_sleep (
            unsigned int ticks )
```

Suspend execution for an interval of time.

Parameters

ticks	The number of ticks to sleep.

5.36.2.13 s_spawn()

```
pid_t s_spawn (
            void *(*)(void *) func,
            void * argv[],
            int fd0,
            int fd1,
            bool is_foreground,
            int num_commands )
```

Spawn a new process.

Parameters

func	The function to be executed by the new process.
argv	Arguments to pass to the function.

Parameters

fd0	File descriptor for standard input.
fd1	File descriptor for standard output.
is_foreground	Boolean indicating if the process is in the foreground.
num_commands	Number of commands to execute.

Returns

The process ID (PID) of the spawned process.

5.36.2.14 s_waitpid()

```
pid_t s_waitpid (
            pid_t pid,
            int * wstatus,
            bool nohang )
```

Wait for a process to change state.

Parameters

pid	Process ID to wait on.
wstatus	Pointer to store the exit status of the waited-for process.
nohang	Boolean to not hang if no status is available.

Returns

The process ID of the terminated process.

5.37 sys_call.h

```
00001 #ifndef _POSIX_C_SOURCE
00002 #define _POSIX_C_SOURCE 200809L
00003 #endif
00004
00005 #ifndef _DEFAULT_SOURCE
00006 #define _DEFAULT_SOURCE 1
00007 #endif
00008
00009 #define _XOPEN_SOURCE 700
00010
00011 #ifndef _SYS_CALL_H
00012 #define _SYS_CALL_H
00013
00014 #include <pcb.h>
00015 #include <stdbool.h>
00016 #include <sys/types.h>
00017
00022 pid_t s_get_pid();
00023
00027 void s_init();
00028
00039 pid_t s_spawn(void* (*func)(void*), void* argv[], int fd0, int fd1, bool is_foreground, int
      num_commands);
00040
00048 pid_t s_waitpid(pid_t pid, int* wstatus, bool nohang);
00049
00057 int s_kill(pid_t pid, int signal, void* args);
```

```
00058
00062 void s_exit(void);
00063
00070 int s_nice(pid_t pid, int priority);
00071
00076 void s_sleep(unsigned int ticks);
00077
00082 pcb_t** s_get_all_process_pcbs(int* count);
00083
00087 void s_stop();
00088
00092 void s_cont();
00093
00098 void s_fg(int pid);
00099
00104 void s_bg(int pid);
00105
00110 SFileDescriptor* s_get_fd_table();
00111
00116 int s_get_f0();
00117
00122 int s_get_f1();
00123
00128 void** s_get_args();
00129
00134 int s_get_num_args();
00135
00136 int s_number_of_pcbs();
00137
00138 #endif  // _SYS_CALL_H
```

5.38 src/test_routines.c File Reference

Test k functions.

```
#include "test_routines.h"
```

Functions

- void initialize_file_system (int fs_number)

  Initialize the file system with a specific test file system number.
- void **cleanup_file_system** ()

  Clean up and unmount the file system.
- void **test_simple_file_operations** ()
- void **test_complex_file_operations** ()
- void **test_list_files** ()
- void **simpleTest** ()
- void **test_k_read** ()
- void **test_k_write** ()
- void **test_k_open_close** ()
- void **test_k_unlink** ()
- void **test_k_lseek** ()
- void **test_k_ls** ()
- void **test_write_to_same_file** ()
- void **test_write_to_stdout** ()
- void **test_write_to_stderr** ()
- void **test_read_from_stdin_and_write_to_stderr** ()
- void **test_demo** ()
- int run_demo ()

  Run a demonstration of specific file system operations.
- int run_test ()

  Run a set of tests to validate file system operations.

5.38.1 Detailed Description

Test k functions.

5.38.2 Function Documentation

5.38.2.1 initialize_file_system()

```
void initialize_file_system (
            int fs_number )
```

Initialize the file system with a specific test file system number.

Parameters

fs_number	The file system number to initialize.

5.38.2.2 run_demo()

```
int run_demo ( )
```

Run a demonstration of specific file system operations.

Returns

Integer status code of the demonstration run (0 for success).

5.38.2.3 run_test()

```
int run_test ( )
```

Run a set of tests to validate file system operations.

Returns

Integer status code of the test run (0 for success).

5.39 test_routines.h

```
00001 #ifndef PENNFAT_TESTS_H
00002 #define PENNFAT_TESTS_H
00003
00004 #include <assert.h>
00005 #include <stdint.h>
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008 #include <string.h>
00009 #include <unistd.h>
00010
00011 #include "pennfat.h"
00012 #include "pennfat/k_fd_structs.h"
00013 #include "pennfat/k_pennfat.h"
```

```
00014 #include "pennfat/k_routines.h"
00015 #include "errors.h"
00016
00017 extern uint16_t* fs_fat;
00018 extern KFileDescriptor g_fdTable[MAX_FD];
00019 extern uint32_t fs_size;
00020
00025 void initialize_file_system(int fs_number);
00026
00030 void cleanup_file_system();
00031
00036 int run_demo();
00037
00042 int run_test();
00043
00044 #endif  // PENNFAT_TESTS_H
```

5.40 spthread.h

```
00001 #ifndef _POSIX_C_SOURCE
00002 #define _POSIX_C_SOURCE 200809L
00003 #endif
00004
00005 #ifndef _DEFAULT_SOURCE
00006 #define _DEFAULT_SOURCE 1
00007 #endif
00008
00009 #define _XOPEN_SOURCE 700
00010
00011 #ifndef SPTHREAD_H_
00012 #define SPTHREAD_H_
00013
00014 #include <pthread.h>
00015 #include <stdbool.h>
00016
00017 // CAUTION: according to `man 7 pthread`:
00018 //
00019 //   On older Linux kernels, SIGUSR1 and SIGUSR2
00020 //   are used.  Applications must avoid the use of whichever set of
00021 //   signals is employed by the implementation.
00022 //
00023 // This may not work on other linux versions
00024
00025 // SIGNAL PTHREAD
00026 // NOTE: if within a created spthread you change
00027 // the behaviour of SIGUSR1, then you will not be able
00028 // to suspend and continue a spthread
00029 #define SIGPTHD SIGUSR1
00030
00031 // declares a struct, but the internals of the
00032 // struct cannot be seen by functions outside of spthread.c
00033 typedef struct spthread_meta_st spthread_meta_t;
00034
00035 // The spthread wrapper struct.
00036 // Sometimes you may have to access the inner pthread member
00037 // but you shouldn't need to do that
00038 typedef struct spthread_st {
00039   pthread_t thread;
00040   spthread_meta_t* meta;
00041 } spthread_t;
00042
00043 // NOTE:
00044 // None of these are signal safe
00045 // Also note that most of these functions are not safe to suspension,
00046 // meaning that if the thread calling these is an spthread and is suspended
00047 // in the middle of spthread_continue or spthread_suspend, then it may not work.
00048 //
00049 // Make sure that the calling thread cannot be suspended before calling these
00050 // functions. Exceptions to this are spthread_exit(), spthread_self() and if a
00051 // thread is continuing or suspending itself.
00052
00053 // spthread_create:
00054 // this function works similar to pthread_create, except for two differences.
00055 // 1) the created pthread is able to be asychronously suspended, and continued
00056 //    using the functions:
00057 //      - spthread_suspend
00058 //      - spthread_continue
00059 // 2) The created pthread will be suspended before it executes the specified
00060 //    routine. It must first be continued with `spthread_continue` before
00061 //    it will start executing.
00062 //
00063 // It is worth noting that this function is not signal safe.
00064 // In other words, it should not be called from a signal handler.
```

```
00065 //
00066 // to avoid repetition, see pthread_create(3) for details
00067 // on arguments and return values as they are the same here.
00068 int spthread_create(spthread_t* thread,
00069                     const pthread_attr_t* attr,
00070                     void* (*start_routine)(void*),
00071                     void* arg);
00072
00073 // The spthread_suspend function will signal to the
00074 // specified thread to suspend execution.
00075 //
00076 // Calling spthread_suspend on an already suspended
00077 // thread does not do anything.
00078 //
00079 // It is worth noting that this function is not signal safe.
00080 // In other words, it should not be called from a signal handler.
00081 //
00082 // args:
00083 // - pthread_t thread: the thread we want to suspend
00084 //   This thread must be created using the spthread_create() function,
00085 //   if created by some other function, the behaviour is undefined.
00086 //
00087 // returns:
00088 // - 0 on success
00089 // - EAGAIN if the thread could not be signaled
00090 // - ENOSYS if not supported on this system
00091 // - ESRCH if the thread specified is not a valid pthread
00092 int spthread_suspend(spthread_t thread);
00093
00094 // The spthread_suspend_self function will cause the calling
00095 // thread (which should be created by spthread_create) to suspend
00096 // itself.
00097 //
00098 // returns:
00099 // - 0 on success
00100 // - EAGAIN if the thread could not be signaled
00101 // - ENOSYS if not supported on this system
00102 // - ESRCH if the calling thread is not an spthread
00103 int spthread_suspend_self();
00104
00105 // The spthread_continue function will signal to the
00106 // specified thread to resume execution if suspended.
00107 //
00108 // Calling spthread_continue on an already non-suspended
00109 // thread does not do anything.
00110 //
00111 // It is worth noting that this function is not signal safe.
00112 // In other words, it should not be called from a signal handler.
00113 //
00114 // args:
00115 // - spthread_t thread: the thread we want to continue
00116 //   This thread must be created using the spthread_create() function,
00117 //   if created by some other function, the behaviour is undefined.
00118 //
00119 // returns:
00120 // - 0 on success
00121 // - EAGAIN if the thread could not be signaled
00122 // - ENOSYS if not supported on this system
00123 // - ESRCH if the thread specified is not a valid pthread
00124 int spthread_continue(spthread_t thread);
00125
00126 // The spthread_cancel function will send a
00127 // cancellation request to the specified thread.
00128 //
00129 // as of now, this function is identical to pthread_cancel(3)
00130 // so to avoid repitition, you should look there.
00131 //
00132 // Here are a few things that are worth highlighting:
00133 // - it is worth noting that it is a cancellation __request__
00134 //   the thread may not terminate immediately, instead the
00135 //   thread is checked whenever it calls a function that is
00136 //   marked as a cancellation point. At those points, it will
00137 //   start the cancellation procedure
00138 // - to make sure all things are de-allocated properly on
00139 //   normal exiting of the thread and when it is cancelled,
00140 //   you should mark a defered de-allocation with
00141 //   pthread_cleanup_push(3).
00142 //   consider the following example:
00143 //
00144 //     void* thread_routine(void* arg) {
00145 //        int* num = malloc(sizeof(int));
00146 //        pthread_cleanup_push(&free, num);
00147 //        return NULL;
00148 //     }
00149 //
00150 //    this program will allocate an integer on the heap
00151 //    and mark that data to be de-allocated on cleanup.
```

```
00152 //    This means that when the thread returns from the
00153 //    routine specified in spthread_create, free will
00154 //    be called on num. This will also happen if the thread
00155 //    is cancelled and not able to be exited normally.
00156 //
00157 //    Another function that should be used in conjunction
00158 //    is pthread_cleanup_pop(3). I will leave that
00159 //    to you to read more on.
00160 //
00161 // It is worth noting that this function is not signal safe.
00162 // In other words, it should not be called from a signal handler.
00163 //
00164 // args:
00165 // - spthread_t thread: the thread we want to cancel.
00166 //   This thread must be created using the spthread_create() function,
00167 //   if created by some other function, the behaviour is undefined.
00168 //
00169 // returns:
00170 // - 0 on success
00171 // - ESRCH if the thread specified is not a valid pthread
00172 int spthread_cancel(spthread_t thread);
00173
00174 // Can be called by a thread to get two peices of information:
00175 // 1. Whether or not the calling thread is an spthread (true or false)
00176 // 2. The spthread_t of the calling thread, if it is an spthread_t
00177 //
00178 // almost always the function will be called like this:
00179 // spthread_t self;
00180 // bool i_am_spthread = spthread_self(&self);
00181 //
00182 // args:
00183 // - spthread_t* thread: the output parameter to get the spthread_t
00184 //   representing the calling thread, if it is an spthread
00185 //
00186 // returns:
00187 // - true if the calling thread is an spthread_t
00188 // - false otherwise.
00189 bool spthread_self(spthread_t* thread);
00190
00191 // The equivalent of pthread_join but for spthread
00192 // To make sure all resources are cleaned up appropriately
00193 // spthreads that are created must at some ppoint have spthread_join
00194 // called on them. Do not use pthread_join on an spthread.
00195 //
00196 // to avoid repetition, see pthread_join(3) for details
00197 // on arguments and return values as they are the same as this function.
00198 int spthread_join(spthread_t thread, void** retval);
00199
00200 // The equivalent of pthread_exit but for spthread
00201 // spthread_exit must be used by spthreads instead of pthread_exit.
00202 // Otherwise, calls to spthread_join or other functions (like spthread_suspend)
00203 // may not work as intended.
00204 //
00205 // to avoid repetition, see pthread_exit(3) for details
00206 // on arguments and return values as they are the same as this function.
00207 void spthread_exit(void* status);
00208
00209 #endif  // SPTHREAD_H_
```

Index