PennOS

2.0

Generated by Doxygen 1.10.0

Chapter 1

PennOS

Welcome to PennOS, a project designed to simulate the functionalities of a UNIX-like operating system. This project was created by Aaron Tsui, Matt Park, Joesph Cho, and Maya Huizar.

Aaron Tsui: PennID 70714281 Matt Park: PennID Joseph Cho: PennID Maya Huizar: PennID

1.0.1 Overview

PennOS is a UNIX-like operating system that runs as a guest OS within a single process on a host OS. It includes implementations of a basic priority scheduler, a FAT file system (PennFAT), and user shell interactions.

1.0.2 Documentation

- **Kernel Documentation**: Here is an overview of the Kernel, Scheduler, and Shell, and its related functions and structure.
- **FAT Documentation**: Here is an overview of the structure of the filesystem, and its related functions.

Chapter 2

Filesystem Team: Aaron Tsui and Joseph Cho

2.1 List of Submitted Source Files for Standalone FAT:

- parser.c and parser.h
- pennfat.c and pennfat.h
- pennfat_kernel.c and pennfat_kernel.h
- standalonefat.c

2.2 List of Submitted Source Files for Integrated FAT with Kernel:

- parser.c and parser.h
- pennfat.c and pennfat.h
- pennfat_kernel.c and pennfat_kernel.h
- sys_call.c and sys_call.h
- shellbuiltins.c and shellbuiltins.h
- error.c and error.h
- bitmap.c and bitmap.h
- All other kernel files necessary for scheduling

2.3 Compilation Instructions:

2.4 For Standalone FAT:

- First, use the make commmand to compile the sttandalonefat.
- Then, type ./bin/standalonefat in the terminal and press enter

2.5 For Integrated FAT with Kernel:

- First, use the make commmand to compile the sttandalonefat.
- Then, type ./bin/standalonefat in the terminal and press enter
- Create a filesystem with command such as "mkfs minfs 1 0"
- Exit the standalone fat
- Use ./bin/pennos "fs_name" to start the pennos with "fs_name" mounted (e.g. ./bin/pennos minfs)

2.6 Overview of Work Accomplished:

2.6.1 Standalone FAT

Standalone FAT is based on FAT 16. It supports a root directory that can store multiple text files. It is interacts with the user using the routine commands which is described in the project description. The implementation of the Standalone FAT is abstracted away from the user. That is, it is not possible for the user to directly utilize the kernel level functions to interact with the FAT system. This reduces the overhead of the user that is using this FAT system so that they can focus on maintaining the data they are interested in.

In the high level, the Standalone FAT is maintained by the FAT region and the DATA region. FAT region is stored in memory so that the system can easily access and index the DATA region. DATA region is divided into the user-set block size. As the data is written or deleted from the DATA region, the Standalone FAT maintains the soundness of the system by modifying and updating both the FAT region and the DATA region.

2.6.2 Integrated FAT

The Integrated FAT is mounted on to the pennos over the terminal. Once it is mounted, the user can interact with the file system just like it would interact with the unix terminal. All of the commands that are FAT related are scheduled by the scheduler and logged accordingly.

2.7 Description of Code and Code Layout:

2.7.1 Standalone FAT

pennfat_kernel.c : All kernel level functions that directly interacts with the FAT system. It it also the only place we use system level functions such as write and read. It is abstracted away from the user, so that they don't have to worry about the actualy file descriptor table or numbers.

pennfat.c : Functions that call the kernel level functions to carry out the routine for the Standalone FAT. This level effectively abstracts the detail of the FAT implementation from the user.

standalone.c : "Shell" for the Standalone FAT. It continuously takes in user input through the terminal and carry them out.

2.7.2 Integrated FAT

pennfat_kernel.c : All kernel level functions that directly interacts with the FAT system. It it also the only place we use system level functions such as write and read. It is abstracted away from the user, so that they don't have to worry about the actualy file descriptor table or numbers.

sys_call.c : Functions that call the kernel level functions to carry out the routine for the Standalone FAT. This level effectively abstracts the detail of the FAT implementation from the user. Also, on error of the kernel level functions, all system level functions sets the errno, so that u_perror can successfully.

shellbuiltins.c : Functions that call the system level functions to carry out the built-in features. All built-in level functions aren't called "directly" by our host system. It is always scheduled by the scheduler in order for it to be executed.

Chapter 3

kernel

The kernel side of PennOS consists of three main aspects. They are described in more details on their own subpages below.

1. **The Kernel**: The kernel refers to the collection of system calls, as well as the overall datastructures and control mechanisms used by the scheduler.
2. **The Scheduler**: The scheduler is the main function that is in charge of deciding which process to schedule/run based on priorities, blocking and unblocking processes, and idling.
3. **The Shell**: The shell is simply a priority zero process that is instantiated at start, and continuously checks for user input to spawn new processes, or modify existing ones.

Chapter 4

scheduler

The scheduler is the main function in charge of mediating processes. The scheduler works in terms of quanta, of which each quantum lasts 100 ms. The general structure/timeline of a quantum looks like this.

The start of a new quantum is triggered by the scheduler receiving a SIGALRM signal, which it receives every 100 ms as set by a timer.

After receiving the alarm, the scheduler will suspend the currenly running process. It then will check the state and status of all blocked processes, unblocking, or updating them as needed. This may include unblocking a parent whose child has exited, or reducing the number of ticks to sleep for a process that called sleep. These events will be logged if necessary. After that, the process will determine the next process to run by checking the next priority to be run, that has a schedulable (read: running) process available. It will log this, and then unsuspend the processes spthread until the next SIGALRM at the end of the quantum, at which point this cycle will repeat.

Chapter 5

shell

The shell is the main process of pennos. It is declared and definen in shell. It is instatiated at tick 0, or the start of pennos, at priority 0. The shell is then scheduled by the scheduler to take in input from the user, and spawn in additional processes via s_spawn.

The commands available to be typed in the shell can be listed by typing "man" in the shell. These are the shell level commands that are specified in the PennOS assignment.

Chapter 6

system

For the system/kernel of PennOS, the most important notion is that of the process control block (PCB).

This can be seen in more detail here: pcb_t.

The main idea of the process control block it represents a process, or thread, that can be in several states. Processes can be running, stopped, blocked, or zombied.

The transitions between processes are mediated via signals sent via s_kill, s_exit, and s_sleep.

The kernel maintains circular linked lists CircularList's of processes in each state. There are 6 in total, one for each state: ZOMBIED, STOPPED, BLOCKED, and 3 for RUNNING, one for each priority level.

As a rule, user level functions should not need to and should not mediate process state transitions, as these will be handled by the scheduler or by system calls.

Chapter 7

Data Structure Index

7.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 8

File Index

8.1 File List

Here is a list of all files with brief descriptions:

Chapter 9

Data Structure Documentation

9.1 CircularList Struct Reference

This structure represents a circular linked list for managing processes.

`#include <clinkedlist.h>`

Data Fields

- Node * head
- Node * tail
- unsigned int size

9.1.1 Detailed Description

This structure represents a circular linked list for managing processes.

9.1.2 Field Documentation

9.1.2.1 head

```
Node* CircularList::head
```

Pointer to the head (first node) of the list.

9.1.2.2 size

```
unsigned int CircularList::size
```

Number of nodes in the list.

9.1.2.3 tail

```
Node* CircularList::tail
```

Pointer to the tail (last node) of the list.

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

- src/util/clinkedlist.h

9.2 directory_entries Struct Reference

This structure stores all required information about the directory entries that are stored in the root directory.

```
#include <pennfat_kernel.h>
```

Data Fields

- char name [32]
- uint32_t size

  32-byte null-terminated file name. name[0] also serves as a special marker 0: end of directory 1: deleted entry; the file is also deleted 2: deleted entry; the file is still being used

- uint16_t firstBlock

  4-byte number of bytes in file
- uint8_t type

  2-byte number indicating the first block number of the file (undefined if size is zero)
- uint8_t perm

  1-byte number for the type of the file, which will be one of the following: 0: unknown 1: a regular file 2: a directory file
- time_t mtime

  file permissions, which will be one of the following: 0: none 2: write only 4: read only 5: read and executable (shell scripts) 6: read and write 7: read, write, and executable
- uint8_t reserved [16]

  creation/modification time as returned by time(2) in Linux

9.2.1 Detailed Description

This structure stores all required information about the directory entries that are stored in the root directory.

9.2.2 Field Documentation

9.2.2.1 firstBlock

```
uint16_t directory_entries::firstBlock
```

4-byte number of bytes in file

9.2.2.2 mtime

```
time_t directory_entries::mtime
```

file permissions, which will be one of the following: 0: none 2: write only 4: read only 5: read and executable (shell scripts) 6: read and write 7: read, write, and executable

9.2.2.3 name

```
char directory_entries::name[32]
```

9.2.2.4 perm

```
uint8_t directory_entries::perm
```

1-byte number for the type of the file, which will be one of the following: 0: unknown 1: a regular file 2: a directory file

9.2.2.5 reserved

```
uint8_t directory_entries::reserved[16]
```

creation/modification time as returned by time(2) in Linux

9.2.2.6 size

```
uint32_t directory_entries::size
```

32-byte null-terminated file name. name[0] also serves as a special marker 0: end of directory 1: deleted entry; the file is also deleted 2: deleted entry; the file is still being used

9.2.2.7 type

```
uint8_t directory_entries::type
```

2-byte number indicating the first block number of the file (undefined if size is zero)

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

- src/util/pennfat_kernel.h

9.3 DynamicPIDArray Struct Reference

Structure for the dynamic array to store child PIDs.

```
#include <array.h>
```

Data Fields

- pid_t * array
- size_t used
- size_t size

9.3.1 Detailed Description

Structure for the dynamic array to store child PIDs.

9.3.2 Field Documentation

9.3.2.1 array

```
pid_t* DynamicPIDArray::array
```

Pointer to the array of child PIDs.

9.3.2.2 size

```
size_t DynamicPIDArray::size
```

Current allocated size of the array.

9.3.2.3 used

```
size_t DynamicPIDArray::used
```

Number of elements currently used.

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

- src/util/array.h

9.4 FD_Bitmap Struct Reference

Structure for managing open file descriptors using a bitmap.

```
#include <bitmap.h>
```

Data Fields

- uint8_t bits [FD_BITMAP_BYTES]

9.4.1 Detailed Description

Structure for managing open file descriptors using a bitmap.

9.4.2 Field Documentation

9.4.2.1 bits

```
uint8_t FD_Bitmap::bits[FD_BITMAP_BYTES]
```

Array of bytes to represent the bitmap.

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

- src/util/bitmap.h

9.5 file_descriptor_st Struct Reference

This structure stores all required information about the file descriptor.

```
#include <pennfat_kernel.h>
```

Data Fields

- int fd
- char * fname

  File descriptor number. This is also used as the index for the global_fd_table.
- int mode

  File name.
- int offset

  Either F_WRITE, F_READ, F_OPEN. Refer to k_open for more details.
- int ref_cnt

  Offset from the start of the file.

9.5.1 Detailed Description

This structure stores all required information about the file descriptor.

9.5.2 Field Documentation

9.5.2.1 fd

```
int file_descriptor_st::fd
```

9.5.2.2 fname

```
char* file_descriptor_st::fname
```

File descriptor number. This is also used as the index for the global_fd_table.

9.5.2.3 mode

```
int file_descriptor_st::mode
```

File name.

9.5.2.4 offset

```
int file_descriptor_st::offset
```

Either F_WRITE, F_READ, F_OPEN. Refer to k_open for more details.

9.5.2.5 ref_cnt

```
int file_descriptor_st::ref_cnt
```

Offset from the start of the file.

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

- src/util/pennfat_kernel.h

9.6 Node Struct Reference

This structure represents a node in the circular linked list.

```
#include <clinkedlist.h>
```

Data Fields

- pcb_t * process
- struct Node * next

9.6.1 Detailed Description

This structure represents a node in the circular linked list.

9.6.2 Field Documentation

9.6.2.1 next

```
struct Node* Node::next
```

Pointer to the next node in the list.

9.6.2.2 process

```
pcb_t* Node::process
```

Pointer to the process control block (pcb_t).

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

- src/util/clinkedlist.h

9.7 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 [ ]

9.7.1 Detailed Description

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

9.7.2 Field Documentation

9.7.2.1 commands

```
char** parsed_command::commands[ ]
```

9.7.2.2 is_background

```
bool parsed_command::is_background
```

9.7.2.3 is_file_append

```
bool parsed_command::is_file_append
```

9.7.2.4 num_commands

```
size_t parsed_command::num_commands
```

9.7.2.5 stdin_file

```
const char* parsed_command::stdin_file
```

9.7.2.6 stdout_file

```
const char* parsed_command::stdout_file
```

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

- src/parser.h

9.8 pcb_t Struct Reference

This structure stores all required information about a running process.

```
#include <kernel.h>
```

Data Fields

- spthread_t handle
- pid_t pid

  This stores a handle to the spthread.

- pid_t ppid

  This stores the PID of the process.
- DynamicPIDArray * child_pids

  This stores the PPID of the process.
- unsigned int priority: 2

  This stores a pointer to a dynamically sized array of child pid_t's.
- process_state_t state

  This the priority level of the process. (0, 1, or 2).
- process_state_t initial_state

  This is an enum storing the process's current state.
- FD_Bitmap * open_fds

  This is an enum storing the process's initial state for bg fg.
- int input_fd

  This stores a bitmap containg all open file descriptors.

- int output_fd

  The input i/o that his process reads data from.
- bool statechanged

  The out i/o that his process writes data to.
- int exit_status

  This contains a bool that keeps track of whether or not the process state has changed.
- int term_signal

  Exit status of process, 0 if exited , -1 if not exited.
- bool waiting_for_change

  Signal number that caused process to terminate, -1 if not terminated.
- pid_t waiting_on_pid

  Bool describing whether or not the process is currently waiting on a process.
- unsigned int ticks_to_wait

  PID of the child the process is currently waiting on, or -1 if none.
- char * processname

  Ticks remaining to wait, used only for s_sleep calls.
- bool is_bg

  Name of process, to be used for logging and ps.
- int job_num

  To signal whether or not the processor in background is terminated.
- char * cmd_name

  The job number if process is in background or stopped.

9.8.1 Detailed Description

This structure stores all required information about a running process.

9.8.2 Field Documentation

9.8.2.1 child_pids

```
DynamicPIDArray* pcb_t::child_pids
```

This stores the PPID of the process.

9.8.2.2 cmd_name

```
char* pcb_t::cmd_name
```

The job number if process is in background or stopped.

9.8.2.3 exit_status

```
int pcb_t::exit_status
```

This contains a bool that keeps track of whether or not the process state has changed.

9.8.2.4 handle

```
spthread_t pcb_t::handle
```

9.8.2.5 initial_state

```
process_state_t pcb_t::initial_state
```

This is an enum storing the process's current state.

9.8.2.6 input_fd

```
int pcb_t::input_fd
```

This stores a bitmap containg all open file descriptors.

9.8.2.7 is_bg

```
bool pcb_t::is_bg
```

Name of process, to be used for logging and ps.

9.8.2.8 job_num

```
int pcb_t::job_num
```

To signal whether or not the processor in background is terminated.

9.8.2.9 open_fds

```
FD_Bitmap* pcb_t::open_fds
```

This is an enum storing the process's initial state for bg fg.

9.8.2.10 output_fd

```
int pcb_t::output_fd
```

The input i/o that his process reads data from.

9.8.2.11 pid

```
pid_t pcb_t::pid
```

This stores a handle to the spthread.

9.8.2.12 ppid

```
pid_t pcb_t::ppid
```

This stores the PID of the process.

9.8.2.13 priority

```
unsigned int pcb_t::priority
```

This stores a pointer to a dynamically sized array of child pid_t's.

9.8.2.14 processname

```
char* pcb_t::processname
```

Ticks remaining to wait, used only for s_sleep calls.

9.8.2.15 state

```
process_state_t pcb_t::state
```

This the priority level of the process. (0, 1, or 2).

9.8.2.16 statechanged

```
bool pcb_t::statechanged
```

The out i/o that his process writes data to.

9.8.2.17 term_signal

```
int pcb_t::term_signal
```

Exit status of process, 0 if exited , -1 if not exited.

9.8.2.18 ticks_to_wait

```
unsigned int pcb_t::ticks_to_wait
```

PID of the child the process is currently waiting on, or -1 if none.

9.8.2.19 waiting_for_change

```
bool pcb_t::waiting_for_change
```

Signal number that caused process to terminate, -1 if not terminated.

9.8.2.20 waiting_on_pid

```
pid_t pcb_t::waiting_on_pid
```

Bool describing whether or not the process is currently waiting on a process.

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

- src/util/kernel.h

9.9 PList Struct Reference

```
#include <prioritylist.h>
```

Data Fields

- PNode * head
- unsigned int size

9.9.1 Field Documentation

9.9.1.1 head

```
PNode* PList::head
```

Pointer to the head (first node) of the list.

9.9.1.2 size

```
unsigned int PList::size
```

Number of nodes in the list.

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

- src/util/prioritylist.h

9.10 PNode Struct Reference

```
#include <prioritylist.h>
```

Data Fields

- unsigned int priority: 2
- struct PNode * next

9.10.1 Field Documentation

9.10.1.1 next

```
struct PNode* PNode::next
```

Pointer to the next node in the list.

9.10.1.2 priority

```
unsigned int PNode::priority
```

Pointer to the process control block (pcb_t).

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

- src/util/prioritylist.h

9.11 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

9.11.1 Field Documentation

9.11.1.1 actual_arg

```
void* spthread_fwd_args_st::actual_arg
```

9.11.1.2 actual_routine

```
pthread_fn spthread_fwd_args_st::actual_routine
```

9.11.1.3 child_meta

```
spthread_meta_t* spthread_fwd_args_st::child_meta
```

9.11.1.4 setup_cond

```
pthread_cond_t spthread_fwd_args_st::setup_cond
```

9.11.1.5 setup_done

```
bool spthread_fwd_args_st::setup_done
```

9.11.1.6 setup_mutex

```
pthread_mutex_t spthread_fwd_args_st::setup_mutex
```

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

- src/util/spthread.c

9.12 spthread_meta_st Struct Reference

Data Fields

- sigset_t suspend_set
- volatile sig_atomic_t state
- pthread_mutex_t meta_mutex

9.12.1 Field Documentation

9.12.1.1 meta_mutex

```
pthread_mutex_t spthread_meta_st::meta_mutex
```

9.12.1.2 state

```
volatile sig_atomic_t spthread_meta_st::state
```

9.12.1.3 suspend_set

```
sigset_t spthread_meta_st::suspend_set
```

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

- src/util/spthread.c

9.13 spthread_signal_args_st Struct Reference

Data Fields

- const int signal
- volatile sig_atomic_t ack
- pthread_mutex_t shutup_mutex

9.13.1 Field Documentation

9.13.1.1 ack

```
volatile sig_atomic_t spthread_signal_args_st::ack
```

9.13.1.2 shutup_mutex

```
pthread_mutex_t spthread_signal_args_st::shutup_mutex
```

9.13.1.3 signal

```
const int spthread_signal_args_st::signal
```

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

- src/util/spthread.c

9.14 spthread_st Struct Reference

```
#include <spthread.h>
```

Data Fields

- pthread_t thread
- spthread_meta_t * meta

9.14.1 Field Documentation

9.14.1.1 meta

```
spthread_meta_t* spthread_st::meta
```

9.14.1.2 thread

```
pthread_t spthread_st::thread
```

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

- src/util/spthread.h

Chapter 10

File Documentation

10.1 doc/fat.md File Reference

10.2 doc/kernel.md File Reference

10.3 doc/README.md File Reference

10.4 doc/scheduler.md File Reference

10.5 doc/shell.md File Reference

10.6 doc/system.md File Reference

10.7 src/parser.h File Reference

```
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
```

Data Structures

- struct parsed_command

Macros

- #define UNEXPECTED_FILE_INPUT 1
- #define UNEXPECTED_FILE_OUTPUT 2
- #define UNEXPECTED_PIPELINE 3
- #define UNEXPECTED_AMPERSAND 4
- #define EXPECT_INPUT_FILENAME 5
- #define EXPECT_OUTPUT_FILENAME 6
- #define EXPECT_COMMANDS 7

Functions

- int parse_command (const char *cmd_line, struct parsed_command **result)
- void print_parsed_command (FILE *output, const struct parsed_command *cmd)
- void print_parser_errcode (FILE *output, int err_code)

10.7.1 Macro Definition Documentation

10.7.1.1 EXPECT_COMMANDS

```
#define EXPECT_COMMANDS 7
```

10.7.1.2 EXPECT_INPUT_FILENAME

```
#define EXPECT_INPUT_FILENAME 5
```

10.7.1.3 EXPECT_OUTPUT_FILENAME

```
#define EXPECT_OUTPUT_FILENAME 6
```

10.7.1.4 UNEXPECTED_AMPERSAND

```
#define UNEXPECTED_AMPERSAND 4
```

10.7.1.5 UNEXPECTED_FILE_INPUT

```
#define UNEXPECTED_FILE_INPUT 1
```

10.7.1.6 UNEXPECTED_FILE_OUTPUT

```
#define UNEXPECTED_FILE_OUTPUT 2
```

10.7.1.7 UNEXPECTED_PIPELINE

```
#define UNEXPECTED_PIPELINE 3
```

10.7.2 Function Documentation

10.7.2.1 parse_command()

```
int parse_command (
            const char * cmd_line,
            struct parsed_command ** result )
```

Arguments: cmd_line: a null-terminated string that is the command line result: a non-null pointer to a `struct parsed_command *`

Return value (int): an error code which can be, 0: parser finished succesfully -1: parser encountered a system call error 1-7: parser specific error, see error type above

This function will parse the given `cmd_line` and store the parsed information into a `struct parsed_command`. The memory needed for the struct will be allocated by this function, and the pointer to the memory will be stored into the given `*result`.

You can directly use the result in system calls. See demo for more information.

If the function returns a successful value (0), a `struct parsed_command` is guareenteed to be allocated and stored in the given `*result`. It is the caller's responsibility to free the given pointer using `free(3)`.

Otherwise, no `struct parsed_command` is allocated and `*result` is unchanged. If a system call error (-1) is returned, the caller can use `errno(3)` or `perror(3)` to gain more information about the error.

10.7.2.2 print_parsed_command()

```
void print_parsed_command (
            FILE * output,
            const struct parsed_command * cmd )
```

10.7.2.3 print_parser_errcode()

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

10.8 parser.h

Go to the documentation of this file.

```
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);
```

10.9 src/pennfat.c File Reference

```
#include "pennfat.h"
#include <unistd.h>
```

Functions

- void prompt (bool isShell)

  Helper function to display the prompt to the user.
- int read_command (char **cmds)

  Helper function that reads user input and handles CTRL-D.
- void int_handler (int signo)

  Helper function that handles CTRL-Z.
- void initialize_global_fd_table ()

  Initializes global fd table with stdin, stdout, and stderr.
- void free_global_fd_table ()
- void mkfs (const char *fs_name, int blocks_in_fat, int block_size_config)

  Creates a "filesytem" (file on host device) with name `fs_name` with each of the `blocks_in_fat` blocks of size specified via `block_size_config`.
- int mount (const char *fs_name)

Mounts the file system specified by `fs_name` via mmap(2)

- int unmount ()

  Unmounts the currently mounted filesystem.

- int get_block_size (int block_size_config)

  Converts config number to actual blocks size in bytes.

- int get_fat_size (int block_size, int blocks_in_fat)

  Computes fat size via block_size and number of blocks in fat.

- int get_num_fat_entries (int block_size, int blocks_in_fat)

  Computes number of fat entries.

- int get_data_size (int block_size, int num_fat_entries)

  Computes size of data region in bytes.

- int get_offset_size (int block_num, int offset)

  Gets offset size from start of filesystem to `block_num` with `offset`.

- void touch (char **args)

  Implements touch function via k_open. Opens all files specified in user command.

- void rm (char **args)

  Implements rm function via k_unlink. Removes files specific by user. Errors if file to be removed doesn't exist.

- void mv (char **args)

  Implements mv function via k_rename. For command mv f1 f2, f1 is renamed to f2 f2 is removed if it already exists.

- void chmod (char **args)

  Implements chmod function via k_change_mode. Changes mode (file permission) of file directory entry Errors if resulting file permission is invalid.

- void cat_file_wa (char **args)

  Implements cat functions E.g. cat f1 -w f2 writes contents of f1 into f2, cat f1 f2 prints contents of f1 and f2 into stdout, cat f1 -a f2, appends contents of f1 into f2.

- void cat_w (char *output)

  Implements cat function that reads from terminal and writes to `output` file. Creates `output` file if it doesn't exist.

- void cat_a (char *output)

  Implements cat function that reads from terminal and appends to `output` file. Creates `output` file if it doesn't exist.

- int ls (char *filename)

  Implements standard ls function.

- int cp_within_fat (char *source, char *dest)

  Implements cp function of copying within the fat by calling k_cp_within_fat Creates `dest` file if it doesn't exist, `source` must exist.

- int cp_to_host (char *source, char *host_dest)

  Implements cp function of copying from fat to host by calling k_cp_to_host Creates `host_dest` if it doesn't exist, `source` must exist.

- int cp_from_host (char *host_source, char *dest)

  Implements cp function of copying from the host to fat by calling k_cp_from_host Creates `dest` file if it doesn't exist, `host_source` nust exist.

10.9.1 Function Documentation

10.9.1.1 cat_a()

```
void cat_a (
            char * output )
```

Implements cat function that reads from terminal and appends to `output` file. Creates `output` file if it doesn't exist.

Parameters

output	File to be appended to

10.9.1.2 cat_file_wa()

```
void cat_file_wa (
            char ** args )
```

Implements cat functions E.g. cat f1 -w f2 writes contents of f1 into f2, cat f1 f2 prints contents of f1 and f2 into stdout, cat f1 -a f2, appends contents of f1 into f2.

Parameters

args	user command

10.9.1.3 cat_w()

```
void cat_w (
            char * output )
```

Implements cat function that reads from terminal and writes to `output` file. Creates `output` file if it doesn't exist.

Parameters

output	File to be written to

10.9.1.4 chmod()

```
void chmod (
            char ** args )
```

Implements chmod function via k_change_mode. Changes mode (file permission) of file directory entry Errors if resulting file permission is invalid.

Parameters

args	user command

10.9.1.5 cp_from_host()

```
int cp_from_host (
            char * host_source,
            char * dest )
```

Implements cp function of copying from the host to fat by calling k_cp_from_host Creates `dest` file if it doesn't exist, `host_source` nust exist.

Parameters

host_source	source file to copy from host
dest	destination file to copy into in fat

Returns

-1 on error, 1 if successful

10.9.1.6 cp_to_host()

```
int cp_to_host (
            char * source,
            char * host_dest )
```

Implements cp function of copying from fat to host by calling k_cp_to_host Creates `host_dest` if it doesn't exist, `source` must exist.

Parameters

source	source file to copy from in fat
host_dest	destination file to copy into in host filesystem

Returns

-1 on error, 1 if successful

10.9.1.7 cp_within_fat()

```
int cp_within_fat (
            char * source,
            char * dest )
```

Implements cp function of copying within the fat by calling k_cp_within_fat Creates `dest` file if it doesn't exist, `source` must exist.

Parameters

source	source file to copy from
dest	destination file to copy into

Returns

-1 on error, 0 if successful

10.9.1.8 free_global_fd_table()

```
void free_global_fd_table ( )
```

10.9.1.9 get_block_size()

```
int get_block_size (
            int block_size_config )
```

Converts config number to actual blocks size in bytes.

Parameters

block_size_config	The block size config number

Returns

integer representing block size in bytes

10.9.1.10 get_data_size()

```
int get_data_size (
            int block_size,
            int num_fat_entries )
```

Computes size of data region in bytes.

Parameters

block_size	the size in bytes of each block
num_fat_entries	number of fat entries

Returns

integer representing data size in bytes

10.9.1.11 get_fat_size()

```
int get_fat_size (
            int block_size,
            int blocks_in_fat )
```

Computes fat size via block_size and number of blocks in fat.

Parameters

block_size	the size in bytes of each block
blocks_in_fat	number of blocks in fat

Returns

integer representing size of fat in bytes

10.9.1.12 get_num_fat_entries()

```
int get_num_fat_entries (
            int block_size,
            int blocks_in_fat )
```

Computes number of fat entries.

Parameters

block_size	the size in bytes of each block
blocks_in_fat	number of blocks in fat

Returns

integer representing number of fat entries

10.9.1.13 get_offset_size()

```
int get_offset_size (
            int block_num,
            int offset )
```

Gets offset size from start of filesystem to `block_num` with `offset`.

Parameters

block_num	the block number
offset	offset from start if `block_num`

Returns

total offset size in bytes

10.9.1.14 initialize_global_fd_table()

```
void initialize_global_fd_table ( )
```

Initializes global fd table with stdin, stdout, and stderr.

10.9.1.15 int_handler()

```
void int_handler (
            int signo )
```

Helper function that handles CTRL-Z.

Parameters

signo	Signal for int handler

10.9.1.16 ls()

```
int ls (
            char * filename )
```

Implements standard ls function.

10.9.1.17 mkfs()

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

Creates a "filesytem" (file on host device) with name `fs_name` with each of the `blocks_in_fat` blocks of size specified via `block_size_config` .

Parameters

fs_name	Name of the file system to be created
blocks_in_fat	Number of blocks in the fat
block_size_config	Configuration specifying size of each block in fat

10.9.1.18 mount()

```
int mount (
            const char * fs_name )
```

Mounts the file system specified by `fs_name` via mmap(2)

Parameters

fs_name	Name of the file system to be mounted

Returns

Returns 0 if successful and -1 if error

10.9.1.19 mv()

```
void mv (
            char ** args )
```

Implements mv function via k_rename. For command mv f1 f2, f1 is renamed to f2 f2 is removed if it already exists.

Parameters

args	user command

10.9.1.20 prompt()

```
void prompt (
            bool shell )
```

Helper function to display the prompt to the user.

Parameters

shell	true if shell prompt, false otherwise

10.9.1.21 read_command()

```
int read_command (
            char ** cmds )
```

Helper function that reads user input and handles CTRL-D.

10.9.1.22 rm()

```
void rm (
            char ** args )
```

Implements rm function via k_unlink. Removes files specific by user. Errors if file to be removed doesn't exist.

Parameters

args	user command

10.9.1.23 touch()

```
void touch (
            char ** args )
```

Implements touch function via k_open. Opens all files specified in user command.

Parameters

args	user command

10.9.1.24 unmount()

```
int unmount ( )
```

Unmounts the currently mounted filesystem.

Returns

Returns 0 if successful and -1 if error

10.10 src/pennfat.h File Reference

```
#include <fcntl.h>
#include <signal.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>
#include "parser.h"
#include "util/pennfat_kernel.h"
```

Macros

- #define MAX_LEN 4096

Functions

- void free_global_fd_table ()
- void mkfs (const char *fs_name, int blocks_in_fat, int block_size_config)

  Creates a "filesytem" (file on host device) with name `fs_name` with each of the `blocks_in_fat` blocks of size specified via `block_size_config`.
- int mount (const char *fs_name)

  Mounts the file system specified by `fs_name` via mmap(2)
- int unmount ()

  Unmounts the currently mounted filesystem.
- void prompt (bool shell)

  Helper function to display the prompt to the user.
- int read_command (char **cmds)

  Helper function that reads user input and handles CTRL-D.
- void int_handler (int signo)

  Helper function that handles CTRL-Z.
- int get_block_size (int block_size_config)

  Converts config number to actual blocks size in bytes.
- int get_fat_size (int block_size, int blocks_in_fat)

  Computes fat size via block_size and number of blocks in fat.

- int get_num_fat_entries (int block_size, int blocks_in_fat)

  Computes number of fat entries.
- int get_data_size (int block_size, int num_fat_entries)

  Computes size of data region in bytes.
- int get_offset_size (int block_num, int offset)

  Gets offset size from start of filesystem to `block_num` with `offset`.
- void initialize_global_fd_table ()

  Initializes global fd table with stdin, stdout, and stderr.
- void touch (char **args)

  Implements touch function via k_open. Opens all files specified in user command.
- void rm (char **args)

  Implements rm function via k_unlink. Removes files specific by user. Errors if file to be removed doesn't exist.
- void mv (char **args)

  Implements mv function via k_rename. For command mv f1 f2, f1 is renamed to f2 f2 is removed if it already exists.
- void chmod (char **args)

  Implements chmod function via k_change_mode. Changes mode (file permission) of file directory entry Errors if resulting file permission is invalid.
- void cat_file_wa (char **args)

  Implements cat functions E.g. cat f1 -w f2 writes contents of f1 into f2, cat f1 f2 prints contents of f1 and f2 into stdout, cat f1 -a f2, appends contents of f1 into f2.
- int ls (char *filename)

  Implements standard ls function.
- int cp_within_fat (char *source, char *dest)

  Implements cp function of copying within the fat by calling k_cp_within_fat Creates `dest` file if it doesn't exist, `source` must exist.
- int cp_to_host (char *source, char *host_dest)

  Implements cp function of copying from fat to host by calling k_cp_to_host Creates `host_dest` if it doesn't exist, `source` must exist.
- int cp_from_host (char *host_source, char *dest)

  Implements cp function of copying from the host to fat by calling k_cp_from_host Creates `dest` file if it doesn't exist, `host_source` nust exist.
- void cat_w (char *output)

  Implements cat function that reads from terminal and writes to `output` file. Creates `output` file if it doesn't exist.
- void cat_a (char *output)

  Implements cat function that reads from terminal and appends to `output` file. Creates `output` file if it doesn't exist.

10.10.1 Macro Definition Documentation

10.10.1.1 MAX_LEN

```
#define MAX_LEN 4096
```

10.10.2 Function Documentation

10.10.2.1 cat_a()

```
void cat_a (
            char * output )
```

Implements cat function that reads from terminal and appends to `output` file. Creates `output` file if it doesn't exist.

Parameters

output	File to be appended to

10.10.2.2 cat_file_wa()

```
void cat_file_wa (
            char ** args )
```

Implements cat functions E.g. cat f1 -w f2 writes contents of f1 into f2, cat f1 f2 prints contents of f1 and f2 into stdout, cat f1 -a f2, appends contents of f1 into f2.

Parameters

args	user command

10.10.2.3 cat_w()

```
void cat_w (
            char * output )
```

Implements cat function that reads from terminal and writes to `output` file. Creates `output` file if it doesn't exist.

Parameters

output	File to be written to

10.10.2.4 chmod()

```
void chmod (
            char ** args )
```

Implements chmod function via k_change_mode. Changes mode (file permission) of file directory entry Errors if resulting file permission is invalid.

Parameters

args	user command

10.10.2.5 cp_from_host()

```
int cp_from_host (
            char * host_source,
            char * dest )
```

Implements cp function of copying from the host to fat by calling k_cp_from_host Creates `dest` file if it doesn't exist, `host_source` nust exist.

Parameters

host_source	source file to copy from host
dest	destination file to copy into in fat

Returns

-1 on error, 1 if successful

10.10.2.6 cp_to_host()

```
int cp_to_host (
            char * source,
            char * host_dest )
```

Implements cp function of copying from fat to host by calling k_cp_to_host Creates `host_dest` if it doesn't exist, `source` must exist.

Parameters

source	source file to copy from in fat
host_dest	destination file to copy into in host filesystem

Returns

-1 on error, 1 if successful

10.10.2.7 cp_within_fat()

```
int cp_within_fat (
            char * source,
            char * dest )
```

Implements cp function of copying within the fat by calling k_cp_within_fat Creates `dest` file if it doesn't exist, `source` must exist.

Parameters

source	source file to copy from
dest	destination file to copy into

Returns

-1 on error, 0 if successful

10.10.2.8 free_global_fd_table()

```
void free_global_fd_table ( )
```

10.10.2.9 get_block_size()

```
int get_block_size (
            int block_size_config )
```

Converts config number to actual blocks size in bytes.

Parameters

block_size_config	The block size config number

Returns

integer representing block size in bytes

10.10.2.10 get_data_size()

```
int get_data_size (
            int block_size,
            int num_fat_entries )
```

Computes size of data region in bytes.

Parameters

block_size	the size in bytes of each block
num_fat_entries	number of fat entries

Returns

integer representing data size in bytes

10.10.2.11 get_fat_size()

```
int get_fat_size (
            int block_size,
            int blocks_in_fat )
```

Computes fat size via block_size and number of blocks in fat.

Parameters

block_size	the size in bytes of each block
blocks_in_fat	number of blocks in fat

Returns

integer representing size of fat in bytes

10.10.2.12 get_num_fat_entries()

```
int get_num_fat_entries (
            int block_size,
            int blocks_in_fat )
```

Computes number of fat entries.

Parameters

block_size	the size in bytes of each block
blocks_in_fat	number of blocks in fat

Returns

integer representing number of fat entries

10.10.2.13 get_offset_size()

```
int get_offset_size (
            int block_num,
            int offset )
```

Gets offset size from start of filesystem to `block_num` with `offset`.

Parameters

block_num	the block number
offset	offset from start if `block_num`

Returns

total offset size in bytes

10.10.2.14 initialize_global_fd_table()

```
void initialize_global_fd_table ( )
```

Initializes global fd table with stdin, stdout, and stderr.

10.10.2.15 int_handler()

```
void int_handler (
            int signo )
```

Helper function that handles CTRL-Z.

Parameters

signo	Signal for int handler

10.10.2.16 ls()

```
int ls (
            char * filename )
```

Implements standard ls function.

10.10.2.17 mkfs()

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

Creates a "filesytem" (file on host device) with name `fs_name` with each of the `blocks_in_fat` blocks of size specified via `block_size_config` .

Parameters

fs_name	Name of the file system to be created
blocks_in_fat	Number of blocks in the fat
block_size_config	Configuration specifying size of each block in fat

10.10.2.18 mount()

```
int mount (
            const char * fs_name )
```

Mounts the file system specified by `fs_name` via mmap(2)

Parameters

fs_name	Name of the file system to be mounted

Returns

Returns 0 if successful and -1 if error

10.10.2.19 mv()

```
void mv (
            char ** args )
```

Implements mv function via k_rename. For command mv f1 f2, f1 is renamed to f2 f2 is removed if it already exists.

Parameters

args	user command

10.10.2.20 prompt()

```
void prompt (
            bool shell )
```

Helper function to display the prompt to the user.

Parameters

shell	true if shell prompt, false otherwise

10.10.2.21 read_command()

```
int read_command (
            char ** cmds )
```

Helper function that reads user input and handles CTRL-D.

10.10.2.22 rm()

```
void rm (
            char ** args )
```

Implements rm function via k_unlink. Removes files specific by user. Errors if file to be removed doesn't exist.

Parameters

args	user command

10.10.2.23 touch()

```
void touch (
            char ** args )
```

Implements touch function via k_open. Opens all files specified in user command.

Parameters

args	user command

10.10.2.24 unmount()

```
int unmount ( )
```

Unmounts the currently mounted filesystem.

Returns

Returns 0 if successful and -1 if error

10.11 pennfat.h

Go to the documentation of this file.

```
00001 #ifndef PENNFAT_H
00002 #define PENNFAT_H
00003
00004 #include <fcntl.h>
00005 #include <signal.h>
00006 #include <stdarg.h>
00007 #include <stdbool.h>
00008 #include <stdint.h>
00009 #include <stdio.h>
00010 #include <stdlib.h>
00011 #include <string.h>
00012 #include <sys/mman.h>
00013 #include <time.h>
00014
00015 #include "parser.h"
00016 #include "util/pennfat_kernel.h"
00017 #ifndef PROMPT_PENN_FAT
00018 #define PROMPT_PENN_FAT "\033[31mpenn-fat>\033[0m "
00019 #endif
00020 #ifndef PROMPT_SHELL
00021 #define PROMPT_SHELL "$ "
00022 #endif
00023 #define MAX_LEN 4096
00024 #endif
00025
00026 void free_global_fd_table();
00027
00039 void mkfs(const char* fs_name, int blocks_in_fat, int block_size_config);
00040
00049 int mount(const char* fs_name);
00050
00057 int unmount();
00058
00065 void prompt(bool shell);
00066
00071 int read_command(char** cmds);
00072
00079 void int_handler(int signo);
00080
00089 int get_block_size(int block_size_config);
00090
00100 int get_fat_size(int block_size, int blocks_in_fat);
00101
00111 int get_num_fat_entries(int block_size, int blocks_in_fat);
00112
00122 int get_data_size(int block_size, int num_fat_entries);
00123
00134 int get_offset_size(int block_num, int offset);
00135
00140 void initialize_global_fd_table();
00141
00149 void touch(char** args);
00150
00158 void rm(char** args);
00159
00168 void mv(char** args);
00169
00178 void chmod(char** args);
00179
00189 void cat_file_wa(char** args);
00190
00195 int ls(char* filename);
00196
```

```
00208 int cp_within_fat(char* source, char* dest);
00209
00221 int cp_to_host(char* source, char* host_dest);
00222
00234 int cp_from_host(char* host_source, char* dest);
00235
00242 void cat_w(char* output);
00243
00250 void cat_a(char* output);
```

10.12 src/pennos.c File Reference

```
#include "pennos.h"
#include <signal.h>
#include "fcntl.h"
#include "parser.h"
#include "pennfat.h"
#include "sys/time.h"
#include "unistd.h"
#include "util/kernel.h"
#include "util/prioritylist.h"
```

Functions

- int b_output_redir (struct parsed_command *parsed)

  handles edge case output redirection
- void scheduler (char *logfile)
- void cancel_and_join (spthread_t thread)
- int main (int argc, char **argv)

Variables

- PList * priority

10.12.1 Function Documentation

10.12.1.1 b_output_redir()

```
int b_output_redir (
            struct parsed_command * parsed )
```

handles edge case output redirection

Parameters

parsed	parsed user command

Returns

-1 if not output redirection, fd number if file is opened

10.12.1.2 cancel_and_join()

```
void cancel_and_join (
            spthread_t thread )
```

10.12.1.3 main()

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

10.12.1.4 scheduler()

```
void scheduler (
            char * logfile )
```

10.12.2 Variable Documentation

10.12.2.1 priority

```
PList* priority
```

10.13 src/pennos.h File Reference

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include "parser.h"
#include "pennfat.h"
#include "util/globals.h"
#include "util/kernel.h"
#include "util/shellbuiltins.h"
#include "util/sys_call.h"
#include "util/stress.h"
```

Macros

- #define _POSIX_C_SOURCE 200809L
- #define _DEFAULT_SOURCE 1
- #define PROMPT "penn-os> "
- #define _XOPEN_SOURCE 700
- #define MAX_LEN 4096
- #define STDIN_FILENO 0
- #define STDOUT_FILENO 1
- #define STDERR_FILENO 2

Functions

- int b_output_redir (struct parsed_command *parsed)
  handles edge case output redirection

10.13.1 Macro Definition Documentation

10.13.1.1 _DEFAULT_SOURCE

```
#define _DEFAULT_SOURCE 1
```

10.13.1.2 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

10.13.1.3 _XOPEN_SOURCE

```
#define _XOPEN_SOURCE 700
```

10.13.1.4 MAX_LEN

```
#define MAX_LEN 4096
```

10.13.1.5 PROMPT

```
#define PROMPT "penn-os> "
```

10.13.1.6 STDERR_FILENO

```
#define STDERR_FILENO 2
```

10.13.1.7 STDIN_FILENO

```
#define STDIN_FILENO 0
```

10.13.1.8 STDOUT_FILENO

```
#define STDOUT_FILENO 1
```

10.13.2 Function Documentation

10.13.2.1 b_output_redir()

```
int b_output_redir (
            struct parsed_command * parsed )
```

handles edge case output redirection

Parameters

parsed	parsed user command

Returns

-1 if not output redirection, fd number if file is opened

10.14 pennos.h

Go to the documentation of this file.

```
00001 #ifndef PENNOS_H
00002 #define PENNOS_H
00003
00004 #ifndef _POSIX_C_SOURCE
00005 #define _POSIX_C_SOURCE 200809L
00006 #endif
00007
00008 #ifndef _DEFAULT_SOURCE
00009 #define _DEFAULT_SOURCE 1
00010 #endif
00011
00012 #undef PROMPT
00013
00014 #ifndef PROMPT
00015 #define PROMPT "penn-os> "
00016 #endif
00017
00018 #define _XOPEN_SOURCE 700
00019
00020 #ifndef MAX_LEN
00021 #define MAX_LEN 4096
00022 #endif
00023
00024 #define STDIN_FILENO 0
00025 #define STDOUT_FILENO 1
00026 #define STDERR_FILENO 2
00027
00028 #include <fcntl.h>
00029 #include <stdio.h>
00030 #include <stdlib.h>
00031 #include "parser.h"
00032 #include "pennfat.h"
00033 #include "util/globals.h"
00034 #include "util/kernel.h"
00035 #include "util/shellbuiltins.h"
00036 #include "util/sys_call.h"
00037 #include "util/stress.h"
00038
00039 static bool done = false;
00040
00041 static pthread_mutex_t done_lock;
00042
00051 int b_output_redir(struct parsed_command* parsed);
00052
00053 #endif
```

10.15 src/standalonefat.c File Reference

```
#include <errno.h>
#include "pennfat.h"
```

Functions

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

10.15.1 Function Documentation

10.15.1.1 main()

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

10.16 src/util/array.c File Reference

```
#include "array.h"
#include <stdlib.h>
```

Functions

- DynamicPIDArray * dynamic_pid_array_create (size_t initial_size)

  Initializes a new dynamic array for PIDs with an initial size.
- void dynamic_pid_array_destroy (DynamicPIDArray *array)

  Destroys a dynamic PID array, freeing its resources.
- bool dynamic_pid_array_add (DynamicPIDArray *array, pid_t pid)

  Adds a PID to the dynamic array, resizing if necessary.
- bool dynamic_pid_array_remove (DynamicPIDArray *array, pid_t pid)

  Removes a PID from the dynamic array.
- bool dynamic_pid_array_contains (const DynamicPIDArray *array, pid_t pid)

  Checks if a PID exists in the dynamic array.

10.16.1 Function Documentation

10.16.1.1 dynamic_pid_array_add()

```
bool dynamic_pid_array_add (
            DynamicPIDArray * array,
            pid_t pid )
```

Adds a PID to the dynamic array, resizing if necessary.

Parameters

array	A pointer to the dynamic PID array.
pid	The PID to add to the array.

Returns

true on success, false on failure (e.g., if memory allocation fails).

10.16.1.2 dynamic_pid_array_contains()

```
bool dynamic_pid_array_contains (
            const DynamicPIDArray * array,
            pid_t pid )
```

Checks if a PID exists in the dynamic array.

Parameters

array	A pointer to the dynamic PID array.
pid	The PID to check for in the array.

Returns

true if the PID exists in the array, false otherwise.

10.16.1.3 dynamic_pid_array_create()

```
DynamicPIDArray * dynamic_pid_array_create (
            size_t initial_size )
```

Initializes a new dynamic array for PIDs with an initial size.

Parameters

initial_size	The initial size of the dynamic array.

Returns

DynamicPIDArray∗ A pointer to the newly created dynamic array structure.

10.16.1.4 dynamic_pid_array_destroy()

```
void dynamic_pid_array_destroy (
            DynamicPIDArray * array )
```

Destroys a dynamic PID array, freeing its resources.

Parameters

array	A pointer to the dynamic PID array to be destroyed.

10.16.1.5 dynamic_pid_array_remove()

```
bool dynamic_pid_array_remove (
```

```
            DynamicPIDArray * array,
            pid_t pid )
```

Removes a PID from the dynamic array.

Parameters

array	A pointer to the dynamic PID array.
pid	The PID to remove from the array.

Returns

true if the PID was successfully removed, false if the PID was not found.

10.17 src/util/array.h File Reference

```
#include <stdbool.h>
#include <stddef.h>
#include <sys/types.h>
```

Data Structures

- struct DynamicPIDArray

  Structure for the dynamic array to store child PIDs.

Functions

- DynamicPIDArray * dynamic_pid_array_create (size_t initial_size)

  Initializes a new dynamic array for PIDs with an initial size.
- void dynamic_pid_array_destroy (DynamicPIDArray *array)

  Destroys a dynamic PID array, freeing its resources.
- bool dynamic_pid_array_add (DynamicPIDArray *array, pid_t pid)

  Adds a PID to the dynamic array, resizing if necessary.
- bool dynamic_pid_array_remove (DynamicPIDArray *array, pid_t pid)

  Removes a PID from the dynamic array.
- bool dynamic_pid_array_contains (const DynamicPIDArray *array, pid_t pid)

  Checks if a PID exists in the dynamic array.

10.17.1 Function Documentation

10.17.1.1 dynamic_pid_array_add()

```
bool dynamic_pid_array_add (
            DynamicPIDArray * array,
            pid_t pid )
```

Adds a PID to the dynamic array, resizing if necessary.

Parameters

array	A pointer to the dynamic PID array.
pid	The PID to add to the array.

Returns

true on success, false on failure (e.g., if memory allocation fails).

10.17.1.2 dynamic_pid_array_contains()

```
bool dynamic_pid_array_contains (
            const DynamicPIDArray * array,
            pid_t pid )
```

Checks if a PID exists in the dynamic array.

Parameters

array	A pointer to the dynamic PID array.
pid	The PID to check for in the array.

Returns

true if the PID exists in the array, false otherwise.

10.17.1.3 dynamic_pid_array_create()

```
DynamicPIDArray * dynamic_pid_array_create (
            size_t initial_size )
```

Initializes a new dynamic array for PIDs with an initial size.

Parameters

initial_size	The initial size of the dynamic array.

Returns

DynamicPIDArray∗ A pointer to the newly created dynamic array structure.

10.17.1.4 dynamic_pid_array_destroy()

```
void dynamic_pid_array_destroy (
            DynamicPIDArray * array )
```

Destroys a dynamic PID array, freeing its resources.

Parameters

array	A pointer to the dynamic PID array to be destroyed.

10.17.1.5 dynamic_pid_array_remove()

```
bool dynamic_pid_array_remove (
            DynamicPIDArray * array,
            pid_t pid )
```

Removes a PID from the dynamic array.

Parameters

array	A pointer to the dynamic PID array.
pid	The PID to remove from the array.

Returns

true if the PID was successfully removed, false if the PID was not found.

10.18 array.h

Go to the documentation of this file.

```
00001 #ifndef ARRAY_H
00002 #define ARRAY_H
00003
00004 #include <stdbool.h>    // For bool type
00005 #include <stddef.h>     // For size_t
00006 #include <sys/types.h>  //needed for ssize_t, if we use ints, can remove
00007
00011 typedef struct {
00012   pid_t* array;
00013   size_t used;
00014   size_t size;
00015 } DynamicPIDArray;
00016
00024 DynamicPIDArray* dynamic_pid_array_create(size_t initial_size);
00025
00031 void dynamic_pid_array_destroy(DynamicPIDArray* array);
00032
00040 bool dynamic_pid_array_add(DynamicPIDArray* array, pid_t pid);
00041
00050 bool dynamic_pid_array_remove(DynamicPIDArray* array, pid_t pid);
00051
00059 bool dynamic_pid_array_contains(const DynamicPIDArray* array, pid_t pid);
00060
00061 #endif
```

10.19 src/util/bitmap.c File Reference

```
#include "bitmap.h"
#include <string.h>
```

Functions

- void fd_bitmap_initialize (FD_Bitmap *bitmap)

  Initializes the bitmap to all zeros, indicating that no file descriptors are in use.
- bool fd_bitmap_set (FD_Bitmap *bitmap, uint32_t fd)

  Sets the bit for a given file descriptor, indicating it is now in use.
- bool fd_bitmap_clear (FD_Bitmap *bitmap, uint32_t fd)

  Clears the bit for a given file descriptor, indicating it is no longer in use.
- bool fd_bitmap_test (const FD_Bitmap *bitmap, uint32_t fd)

  Tests whether the bit for a given file descriptor is set, indicating whether it is in use.

10.19.1 Function Documentation

10.19.1.1 fd_bitmap_clear()

```
bool fd_bitmap_clear (
            FD_Bitmap * bitmap,
            uint32_t fd )
```

Clears the bit for a given file descriptor, indicating it is no longer in use.

Parameters

bitmap	A pointer to the file descriptor bitmap.
fd	The file descriptor to mark as not in use.

Returns

bool True if the operation was successful, false if the fd is out of range.

10.19.1.2 fd_bitmap_initialize()

```
void fd_bitmap_initialize (
            FD_Bitmap * bitmap )
```

Initializes the bitmap to all zeros, indicating that no file descriptors are in use.

Parameters

bitmap	A pointer to the file descriptor bitmap to initialize.

10.19.1.3 fd_bitmap_set()

```
bool fd_bitmap_set (
            FD_Bitmap * bitmap,
            uint32_t fd )
```

Sets the bit for a given file descriptor, indicating it is now in use.

Parameters

bitmap	A pointer to the file descriptor bitmap.
fd	The file descriptor to mark as in use.

Returns

bool True if the operation was successful, false if the fd is out of range.

10.19.1.4 fd_bitmap_test()

```
bool fd_bitmap_test (
            const FD_Bitmap * bitmap,
            uint32_t fd )
```

Tests whether the bit for a given file descriptor is set, indicating whether it is in use.

Parameters

bitmap	A pointer to the file descriptor bitmap.
fd	The file descriptor to check.

Returns

bool True if the file descriptor is in use, false otherwise.

10.20 src/util/bitmap.h File Reference

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

Data Structures

- struct FD_Bitmap

  Structure for managing open file descriptors using a bitmap.

Macros

- #define FD_BITMAP_SIZE 1024
- #define FD_BITMAP_BYTES (FD_BITMAP_SIZE / 8)

Functions

- void fd_bitmap_initialize (FD_Bitmap *bitmap)

  Initializes the bitmap to all zeros, indicating that no file descriptors are in use.
- bool fd_bitmap_set (FD_Bitmap *bitmap, uint32_t fd)

  Sets the bit for a given file descriptor, indicating it is now in use.
- bool fd_bitmap_clear (FD_Bitmap *bitmap, uint32_t fd)

  Clears the bit for a given file descriptor, indicating it is no longer in use.
- bool fd_bitmap_test (const FD_Bitmap *bitmap, uint32_t fd)

  Tests whether the bit for a given file descriptor is set, indicating whether it is in use.

10.20.1 Macro Definition Documentation

10.20.1.1 FD_BITMAP_BYTES

```
#define FD_BITMAP_BYTES (FD_BITMAP_SIZE / 8)
```

10.20.1.2 FD_BITMAP_SIZE

```
#define FD_BITMAP_SIZE 1024
```

10.20.2 Function Documentation

10.20.2.1 fd_bitmap_clear()

```
bool fd_bitmap_clear (
            FD_Bitmap * bitmap,
            uint32_t fd )
```

Clears the bit for a given file descriptor, indicating it is no longer in use.

Parameters

bitmap	A pointer to the file descriptor bitmap.
fd	The file descriptor to mark as not in use.

Returns

bool True if the operation was successful, false if the fd is out of range.

10.20.2.2 fd_bitmap_initialize()

```
void fd_bitmap_initialize (
            FD_Bitmap * bitmap )
```

Initializes the bitmap to all zeros, indicating that no file descriptors are in use.

Parameters

bitmap	A pointer to the file descriptor bitmap to initialize.

10.20.2.3 fd_bitmap_set()

```
bool fd_bitmap_set (
            FD_Bitmap * bitmap,
            uint32_t fd )
```

Sets the bit for a given file descriptor, indicating it is now in use.

Parameters

bitmap	A pointer to the file descriptor bitmap.
fd	The file descriptor to mark as in use.

Returns

bool True if the operation was successful, false if the fd is out of range.

10.20.2.4 fd_bitmap_test()

```
bool fd_bitmap_test (
            const FD_Bitmap * bitmap,
            uint32_t fd )
```

Tests whether the bit for a given file descriptor is set, indicating whether it is in use.

Parameters

bitmap	A pointer to the file descriptor bitmap.
fd	The file descriptor to check.

Returns

bool True if the file descriptor is in use, false otherwise.

10.21 bitmap.h

Go to the documentation of this file.

```
00001 #ifndef OPENFD_BITMAP_H
00002 #define OPENFD_BITMAP_H
00003
00004 #include <stdbool.h>  // For bool type
00005 #include <stdint.h>   // For uint8_t, uint32_t types
00006
00007 #define FD_BITMAP_SIZE 1024  // Maximum number of file descriptors
00008 #define FD_BITMAP_BYTES (FD_BITMAP_SIZE / 8)  // Number of bytes needed
```

```
00009
00013 typedef struct {
00014   uint8_t bits[FD_BITMAP_BYTES];
00015 } FD_Bitmap;
00016
00023 void fd_bitmap_initialize(FD_Bitmap* bitmap);
00024
00033 bool fd_bitmap_set(FD_Bitmap* bitmap, uint32_t fd);
00034
00044 bool fd_bitmap_clear(FD_Bitmap* bitmap, uint32_t fd);
00045
00054 bool fd_bitmap_test(const FD_Bitmap* bitmap, uint32_t fd);
00055
00056 #endif
```

10.22 src/util/clinkedlist.c File Reference

```
#include "clinkedlist.h"
#include <stdlib.h>
#include "kernel.h"
```

Functions

- CircularList * init_list (void)
- int add_process (CircularList *list, pcb_t *process)
- void add_process_front (CircularList *list, pcb_t *process)
- bool remove_process (CircularList *list, pid_t pid)
- pcb_t * find_process (CircularList *list, pid_t pid)
- pcb_t * find_process_job_id (CircularList *list, int index)
- void free_list (CircularList *list)

10.22.1 Function Documentation

10.22.1.1 add_process()

```
int add_process (
            CircularList * list,
            pcb_t * process )
```

Adds a new process to the circular linked list.

Parameters

list	Pointer to the circular linked list.
process	Pointer to the process control block (pcb_t) to add.

10.22.1.2 add_process_front()

```
void add_process_front (
            CircularList * list,
            pcb_t * process )
```

Adds a new process to the front of circular linked list.

Parameters

list	Pointer to the circular linked list.
process	Pointer to the process control block (pcb_t) to add.

10.22.1.3 find_process()

```
pcb_t * find_process (
            CircularList * list,
            pid_t pid )
```

Finds a process in the circular linked list by its PID.

Parameters

list	Pointer to the circular linked list.
pid	PID of the process to find.

Returns

pcb_t* Pointer to the found process control block, or NULL if not found.

10.22.1.4 find_process_job_id()

```
pcb_t * find_process_job_id (
            CircularList * list,
            int index )
```

Finds a process in the circular linked list by its Job ID.

Parameters

list	Pointer to the circular linked list.
index	Job Id specified by user.

Returns

pcb_t* Pointer to the found process control block, or NULL if not found.

10.22.1.5 free_list()

```
void free_list (
            CircularList * list )
```

Frees all nodes and their associated processes in a circular linked list, then frees the list itself.

Parameters

list	Pointer to the circular linked list to free.

10.22.1.6 init_list()

```
CircularList * init_list (
            void  )
```

Initializes a circular linked list.

Returns

CircularList* Pointer to the newly initialized list.

10.22.1.7 remove_process()

```
bool remove_process (
            CircularList * list,
            pid_t pid )
```

Removes a process from the circular linked list by its PID.

Parameters

list	Pointer to the circular linked list.
pid	PID of the process to remove.

Returns

bool true if the process was successfully removed, false otherwise.

10.23 src/util/clinkedlist.h File Reference

```
#include <stdbool.h>
#include <sys/types.h>
#include <stdlib.h>
```

Data Structures

- struct Node

  This structure represents a node in the circular linked list.
- struct CircularList

  This structure represents a circular linked list for managing processes.

Typedefs

- typedef struct pcb_t pcb_t
- typedef struct Node Node

Functions

- CircularList * init_list (void)
- int add_process (CircularList *list, pcb_t *process)
- void add_process_front (CircularList *list, pcb_t *process)
- bool remove_process (CircularList *list, pid_t pid)
- pcb_t * find_process (CircularList *list, pid_t pid)
- pcb_t * find_process_job_id (CircularList *list, int index)
- void free_list (CircularList *list)

10.23.1 Typedef Documentation

10.23.1.1 Node

```
typedef struct Node Node
```

10.23.1.2 pcb_t

```
typedef struct pcb_t pcb_t
```

10.23.2 Function Documentation

10.23.2.1 add_process()

```
int add_process (
            CircularList * list,
            pcb_t * process )
```

Adds a new process to the circular linked list.

Parameters

list	Pointer to the circular linked list.
process	Pointer to the process control block (pcb_t) to add.

10.23.2.2 add_process_front()

```
void add_process_front (
            CircularList * list,
            pcb_t * process )
```

Adds a new process to the front of circular linked list.

Parameters

list	Pointer to the circular linked list.
process	Pointer to the process control block (pcb_t) to add.

10.23.2.3 find_process()

```
pcb_t * find_process (
            CircularList * list,
            pid_t pid )
```

Finds a process in the circular linked list by its PID.

Parameters

list	Pointer to the circular linked list.
pid	PID of the process to find.

Returns

pcb_t* Pointer to the found process control block, or NULL if not found.

10.23.2.4 find_process_job_id()

```
pcb_t * find_process_job_id (
            CircularList * list,
            int index )
```

Finds a process in the circular linked list by its Job ID.

Parameters

list	Pointer to the circular linked list.
index	Job Id specified by user.

Returns

pcb_t* Pointer to the found process control block, or NULL if not found.

10.23.2.5 free_list()

```
void free_list (
            CircularList * list )
```

Frees all nodes and their associated processes in a circular linked list, then frees the list itself.

Parameters

list	Pointer to the circular linked list to free.

10.23.2.6 init_list()

```
CircularList * init_list (
            void  )
```

Initializes a circular linked list.

Returns

CircularList* Pointer to the newly initialized list.

10.23.2.7 remove_process()

```
bool remove_process (
            CircularList * list,
            pid_t pid )
```

Removes a process from the circular linked list by its PID.

Parameters

list	Pointer to the circular linked list.
pid	PID of the process to remove.

Returns

bool true if the process was successfully removed, false otherwise.

10.24 clinkedlist.h

Go to the documentation of this file.

```
00001 #ifndef SCHEDULER_LIST_H
00002 #define SCHEDULER_LIST_H
00003
00004 #include <stdbool.h>
00005 #include <sys/types.h>
00006 #include <stdlib.h>
00007
00008 typedef struct pcb_t pcb_t;
00009
00014 typedef struct Node {
00015   pcb_t* process;
00016   struct Node* next;
00017 } Node;
00018
00024 typedef struct {
00025   Node* head;
00026   Node* tail;
00027   unsigned int size;
00028 } CircularList;
00029
```

```
00034 CircularList* init_list(void);
00035
00041 int add_process(CircularList* list, pcb_t* process);
00042
00048 void add_process_front(CircularList* list, pcb_t* process);
00049
00056 bool remove_process(CircularList* list, pid_t pid);
00057
00065 pcb_t* find_process(CircularList* list, pid_t pid);
00066
00074 pcb_t* find_process_job_id(CircularList* list, int index);
00075
00082 void free_list(CircularList* list);
00083
00084 #endif  // SCHEDULER_LIST_H
```

10.25 src/util/error.c File Reference

```
#include "error.h"
```

Functions

- void u_perror (char *message)

  This is an analogue of the perror(3) function. This allows the shell or other function to pass in a message describing what error occured, which the function then concatenates to the default error message based on your errno value. The filesystem and kernel will both set errno and return -1 on system calls if they fail, after which u_perror() can be called.

10.25.1 Function Documentation

10.25.1.1 u_perror()

```
void u_perror (
            char * message )
```

This is an analogue of the perror(3) function. This allows the shell or other function to pass in a message describing what error occured, which the function then concatenates to the default error message based on your errno value. The filesystem and kernel will both set errno and return -1 on system calls if they fail, after which u_perror() can be called.

Parameters

message	This is the message that will be concatenated to the default error message.

10.26 src/util/error.h File Reference

```
#include <string.h>
#include "errno.h"
#include "stdio.h"
#include "unistd.h"
```

Macros

- #define EPCBCREATE 0
- #define ENOARGS 1
- #define EADDPROC 2
- #define ETHREADCREATE 3
- #define EBITMAP 4
- #define EINVARG 5
- #define EBADFILENAME 400
- #define EMULTWRITE 401
- #define EWRONGPERM 402
- #define ESYSERR 403
- #define ENOFILE 404
- #define EFILEDEL 405
- #define EINVALIDFD 406
- #define EINVALIDPARAMETER 407
- #define EUSEDFILE 408
- #define EINVALIDCHMOD 500
- #define EREADERROR 501
- #define ENOREADPERM 502
- #define ENOWRITEPERM 503
- #define ENOPROC 410

  feel free to fix the number
- #define EPCBSTATE 411
- #define EREMOVEPROC 412
- #define EINVALIDCMD 413
- #define ELISTNULL 414
- #define ENOJOB 415
- #define ENOPIDJOB 416
- #define ENOTSTOP 417
- #define EINVALIDSIG 418
- #define EINVALIDLOG 419
- #define EINVALIDLOGWRITE 420
- #define EINVALIDSTDOUT 421

Functions

- void u_perror (char *message)

  This is an analogue of the perror(3) function. This allows the shell or other function to pass in a message describing what error occured, which the function then concatenates to the default error message based on your errno value. The filesystem and kernel will both set errno and return -1 on system calls if they fail, after which u_perror() can be called.

Variables

- int errno

10.26.1 Macro Definition Documentation

10.26.1.1 EADDPROC

```
#define EADDPROC 2
```

10.26.1.2 EBADFILENAME

```
#define EBADFILENAME 400
```

10.26.1.3 EBITMAP

```
#define EBITMAP 4
```

10.26.1.4 EFILEDEL

```
#define EFILEDEL 405
```

10.26.1.5 EINVALIDCHMOD

```
#define EINVALIDCHMOD 500
```

10.26.1.6 EINVALIDCMD

```
#define EINVALIDCMD 413
```

10.26.1.7 EINVALIDFD

```
#define EINVALIDFD 406
```

10.26.1.8 EINVALIDLOG

```
#define EINVALIDLOG 419
```

10.26.1.9 EINVALIDLOGWRITE

```
#define EINVALIDLOGWRITE 420
```

10.26.1.10 EINVALIDPARAMETER

```
#define EINVALIDPARAMETER 407
```

10.26.1.11 EINVALIDSIG

```
#define EINVALIDSIG 418
```

10.26.1.12 EINVALIDSTDOUT

```
#define EINVALIDSTDOUT 421
```

10.26.1.13 EINVARG

```
#define EINVARG 5
```

10.26.1.14 ELISTNULL

```
#define ELISTNULL 414
```

10.26.1.15 EMULTWRITE

```
#define EMULTWRITE 401
```

10.26.1.16 ENOARGS

```
#define ENOARGS 1
```

10.26.1.17 ENOFILE

```
#define ENOFILE 404
```

10.26.1.18 ENOJOB

```
#define ENOJOB 415
```

10.26.1.19 ENOPIDJOB

```
#define ENOPIDJOB 416
```

10.26.1.20 ENOPROC

```
#define ENOPROC 410
```

feel free to fix the number

10.26.1.21 ENOREADPERM

```
#define ENOREADPERM 502
```

10.26.1.22 ENOTSTOP

```
#define ENOTSTOP 417
```

10.26.1.23 ENOWRITEPERM

```
#define ENOWRITEPERM 503
```

10.26.1.24 EPCBCREATE

```
#define EPCBCREATE 0
```

10.26.1.25 EPCBSTATE

```
#define EPCBSTATE 411
```

10.26.1.26 EREADERROR

```
#define EREADERROR 501
```

10.26.1.27 EREMOVEPROC

```
#define EREMOVEPROC 412
```

10.26.1.28 ESYSERR

```
#define ESYSERR 403
```

10.26.1.29 ETHREADCREATE

```
#define ETHREADCREATE 3
```

10.26.1.30 EUSEDFILE

```
#define EUSEDFILE 408
```

10.26.1.31 EWRONGPERM

```
#define EWRONGPERM 402
```

10.26.2 Function Documentation

10.26.2.1 u_perror()

```
void u_perror (
            char * message )
```

This is an analogue of the perror(3) function. This allows the shell or other function to pass in a message describing what error occured, which the function then concatenates to the default error message based on your errno value. The filesystem and kernel will both set errno and return -1 on system calls if they fail, after which u_perror() can be called.

Parameters

message	This is the message that will be concatenated to the default error message.

10.26.3 Variable Documentation

10.26.3.1 errno

```
int errno  [extern]
```

This is an integer that can be set by various functions to denote the kind of error that occurred. This is similar to that defined by errno.h, except that it is a custom definition.

10.27 error.h

Go to the documentation of this file.

```
00001 #include <string.h>
00002 #include "errno.h"
00003 #include "stdio.h"
00004 #include "unistd.h"
00005
00006 #ifndef ERROR
00007 #define ERROR
00014 extern int errno;
00015
00016 // KERNEL LEVEL ERRORS
00017
00018 // S_SPAWN ERRORS
00019 #define EPCBCREATE 0
00020 #define ENOARGS 1
00021 #define EADDPROC 2
00022 #define ETHREADCREATE 3
00023 #define EBITMAP 4
00024
00025 // S_SLEEP ERRORS
00026 #define EINVARG 5
00027
00028 // S_OPEN, S_READ, S_WRITE, S_CLOSE, S_UNLINK, S_LSEEK, S_LS, S_CHMOD ERRORS
00029 #define EBADFILENAME 400
00030 #define EMULTWRITE 401
00031 #define EWRONGPERM 402
00032 #define ESYSERR 403
00033 #define ENOFILE 404
00034 #define EFILEDEL 405
00035 #define EINVALIDFD 406
00036 #define EINVALIDPARAMETER 407
00037 #define EUSEDFILE 408
00038 #define EINVALIDCHMOD 500
00039 #define EREADERROR 501
00040 #define ENOREADPERM 502
00041 #define ENOWRITEPERM 503
00042
00044 #define ENOPROC 410
00045 #define EPCBSTATE 411
00046 #define EREMOVEPROC 412
00047 #define EINVALIDCMD 413
00048 #define ELISTNULL 414
00049 #define ENOJOB 415
00050 #define ENOPIDJOB 416
00051 #define ENOTSTOP 417
00052 #define EINVALIDSIG 418
00053 #define EINVALIDLOG 419
00054 #define EINVALIDLOGWRITE 420
00055 #define EINVALIDSTDOUT 421
00056
00057 // FAT LEVEL ERRORS
00068 void u_perror(char* message);
00069
00070 #endif
```

10.28 src/util/globals.c File Reference

```
#include "globals.h"
```

Variables

- CircularList * processes [3]
- CircularList * blocked
- CircularList * stopped
- CircularList * zombied
- CircularList * bg_list
- pcb_t * fg_proc = NULL
- pcb_t * current = NULL
- pid_t next_pid = 1
- uint64_t job_id = 1
- int logfiledescriptor = 0
- unsigned int tick = 0

10.28.1 Variable Documentation

10.28.1.1 bg_list

```
CircularList* bg_list
```

A global pointer to the process list of background processes. The processes enter this list when

10.28.1.2 blocked

```
CircularList* blocked
```

A global pointer to the process list of blocked processes. Processes enter this list via s_waitpid() or s_sleep().

10.28.1.3 current

```
pcb_t* current = NULL
```

This is the currently scheduled process. It can be accessed by any method to easily access the current method.

10.28.1.4 fg_proc

```
pcb_t* fg_proc = NULL
```

This is the foreground processor and if it exists, the processor takes control of the terminal

10.28.1.5 job_id

```
uint64_t job_id = 1
```

This is to keep track of the job numbers from processes that are in the background or stopped.

10.28.1.6 logfiledescriptor

```
int logfiledescriptor = 0
```

This is the int representing the file descriptor of the log file, to be used for writing purposes for the logging of events.

10.28.1.7 next_pid

```
pid_t next_pid = 1
```

This the next pid to be used, by k_proc_create(), to ensure that PIDs are not duplicated. This may be rewritten later to reuse/reallocate old processes that have been exited/terminated. I have no strong desire to do so, but do so if you wish.

10.28.1.8 processes

```
CircularList* processes[3]
```

A global array of pointers to the process lists. Each priority level can be accessed via processes[priority]. Processes enter this list after creation or via s_kill after receiving P_SIGCONT when stopped.

10.28.1.9 stopped

```
CircularList* stopped
```

A global pointer to the process list of stopped processes. Processes enter this list via s_kill() after receiving a P_SIGTERM signal.

10.28.1.10 tick

```
unsigned int tick = 0
```

This is an int representing the current tick of pennos, to be used for logging purposes.

10.28.1.11 zombied

```
CircularList* zombied
```

A global pointer to the process list of zombied/terminated processes. These processes enter this list via s_exit() or s_kill(), with the P_SIGTERM signal.

10.29 src/util/globals.h File Reference

```
#include "clinkedlist.h"
#include "kernel.h"
#include "prioritylist.h"
```

Variables

- CircularList * processes [3]
- CircularList * blocked
- CircularList * stopped
- CircularList * zombied
- CircularList * bg_list
- pcb_t * current
- pcb_t * fg_proc
- pid_t next_pid
- uint64_t job_id
- int logfiledescriptor
- unsigned int tick

10.29.1 Variable Documentation

10.29.1.1 bg_list

```
CircularList* bg_list  [extern]
```

A global pointer to the process list of background processes. The processes enter this list when

10.29.1.2 blocked

```
CircularList* blocked  [extern]
```

A global pointer to the process list of blocked processes. Processes enter this list via s_waitpid() or s_sleep().

10.29.1.3 current

```
pcb_t* current  [extern]
```

This is the currently scheduled process. It can be accessed by any method to easily access the current method.

10.29.1.4 fg_proc

```
pcb_t* fg_proc  [extern]
```

This is the foreground processor and if it exists, the processor takes control of the terminal

10.29.1.5 job_id

```
uint64_t job_id  [extern]
```

This is to keep track of the job numbers from processes that are in the background or stopped.

10.29.1.6 logfiledescriptor

```
int logfiledescriptor  [extern]
```

This is the int representing the file descriptor of the log file, to be used for writing purposes for the logging of events.

10.29.1.7 next_pid

```
pid_t next_pid  [extern]
```

This the next pid to be used, by k_proc_create(), to ensure that PIDs are not duplicated. This may be rewritten later to reuse/reallocate old processes that have been exited/terminated. I have no strong desire to do so, but do so if you wish.

10.29.1.8 processes

```
CircularList* processes[3]  [extern]
```

A global array of pointers to the process lists. Each priority level can be accessed via processes[priority]. Processes enter this list after creation or via s_kill after receiving P_SIGCONT when stopped.

10.29.1.9 stopped

```
CircularList* stopped  [extern]
```

A global pointer to the process list of stopped processes. Processes enter this list via s_kill() after receiving a P_SIGTERM signal.

10.29.1.10 tick

```
unsigned int tick  [extern]
```

This is an int representing the current tick of pennos, to be used for logging purposes.

10.29.1.11 zombied

```
CircularList* zombied  [extern]
```

A global pointer to the process list of zombied/terminated processes. These processes enter this list via s_exit() or s_kill(), with the P_SIGTERM signal.

10.30 globals.h

Go to the documentation of this file.

```
00001 #ifndef GLOBALS_H
00002 #define GLOBALS_H
00003
00004 #include "clinkedlist.h"
00005 #include "kernel.h"
00006 #include "prioritylist.h"
00007
00014 extern CircularList* processes[3];
00015
00020 extern CircularList* blocked;
00021
00026 extern CircularList* stopped;
00027
00033 extern CircularList* zombied;
00034
00039 extern CircularList* bg_list;
00040
00046 extern pcb_t* current;
00047
00053 extern pcb_t* fg_proc;
00054
00062 extern pid_t next_pid;
00063
00069 extern uint64_t job_id;
00070
00076 extern int logfiledescriptor;
00077
00083 extern unsigned int tick;
00084
00085 #endif
```

10.31 src/util/kernel.c File Reference

```
#include "kernel.h"
#include "stdio.h"
```

Functions

- pcb_t * k_proc_create (pcb_t *parent)

  Create a new child process, inheriting applicable properties from the parent.
- void k_proc_cleanup (pcb_t *proc)

  Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc.

10.31.1 Function Documentation

10.31.1.1 k_proc_cleanup()

```
void k_proc_cleanup (
            pcb_t * proc )
```

Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc.

Parameters

proc	This is a pointer to the process control block of a process that has terminated.

need more

10.31.1.2 k_proc_create()

```
pcb_t * k_proc_create (
            pcb_t * parent )
```

Create a new child process, inheriting applicable properties from the parent.

Parameters

parent	This is a pointer to the process control block of the parent, from which it inherits.

Returns

Reference to the child PCB.

10.32 src/util/kernel.h File Reference

```
#include <sys/types.h>
#include "array.h"
#include "bitmap.h"
#include "clinkedlist.h"
#include "globals.h"
#include "spthread.h"
#include "stdlib.h"
#include "sys_call.h"
```

Data Structures

- struct pcb_t

  This structure stores all required information about a running process.

Macros

- #define P_SIGSTOP 0

  This is the STOP signal definition to be used by s_kill(). Running processes (ONLY) that receive the P_SIGSTOP signal will become stopped and have their state and process list adjusted accordingly. Note that statechanged will NOT be changed, as this state transition does NOT cause s_waitpid() to return/unblock.
- #define P_SIGCONT 1

  This is the CONTINUE signal definition to be used by s_kill(). Stopped processes (ONLY) that receive the P_SIGCONT signal will become running and have their state and process list adjusted accordingly. Note that statechanged will NOT be changed, as this state transition does NOT cause s_waitpid() to return/unblock.
- #define P_SIGTER 2

  This is the TERMINATE signal definition to be used by s_kill(). Any process that receives the P_SIGTER signal will become zombied and have their state and process list adjusted accordingly. Note that statechanged WILL be changed, as this state transition DOES cause s_waitpid() to return/unblock.

Typedefs

- typedef struct pcb_t pcb_t

Enumerations

- enum process_state_t { RUNNING , STOPPED , BLOCKED , ZOMBIED }

  Defines the possible states of a process in the system.

Functions

- pcb_t * k_proc_create (pcb_t *parent)

  Create a new child process, inheriting applicable properties from the parent.
- void k_proc_cleanup (pcb_t *proc)

  Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc.

10.32.1 Macro Definition Documentation

10.32.1.1 P_SIGCONT

```
#define P_SIGCONT 1
```

This is the CONTINUE signal definition to be used by s_kill(). Stopped processes (ONLY) that receive the P_↩ SIGCONT signal will become running and have their state and process list adjusted accordingly. Note that state-changed will NOT be changed, as this state transition does NOT cause s_waitpid() to return/unblock.

10.32.1.2 P_SIGSTOP

```
#define P_SIGSTOP 0
```

This is the STOP signal definition to be used by s_kill(). Running processes (ONLY) that receive the P_SIGSTOP signal will become stopped and have their state and process list adjusted accordingly. Note that statechanged will NOT be changed, as this state transition does NOT cause s_waitpid() to return/unblock.

10.32.1.3 P_SIGTER

```
#define P_SIGTER 2
```

This is the TERMINATE signal definition to be used by s_kill(). Any process that receives the P_SIGTER signal will become zombied and have their state and process list adjusted accordingly. Note that statechanged WILL be changed, as this state transition DOES cause s_waitpid() to return/unblock.

10.32.2 Typedef Documentation

10.32.2.1 pcb_t

```
typedef struct pcb_t pcb_t
```

10.32.3 Enumeration Type Documentation

10.32.3.1 process_state_t

```
enum process_state_t
```

Defines the possible states of a process in the system.

This enumeration lists all the possible states that a process could be in at any given time. It is used within the pcb_t structure to track the current state of each process.

Enumerator

RUNNING	Process is currently executing. A process enters the RUNNING state when the scheduler selects it for execution, typically from the READY state.
STOPPED	Process is not executing, but can be resumed. A process should only become STOPPED if signaled by s_kill, recieving the P_SIGSTOP signal.
BLOCKED	Process is not executing, waiting for an event to occur. A process should only be blocked if it made a call to either s_waitpid or s_sleep.
ZOMBIED	Process has finished execution but awaits resource cleanup. A process enters the ZOMBIED state after it has finished its execution and is waiting for the parent process to read its exit status. If the parent process ever exits prior to reading exit status, this process should immediately cleaned up.

10.32.4 Function Documentation

10.32.4.1 k_proc_cleanup()

```
void k_proc_cleanup (
            pcb_t * proc )
```

Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc.

Parameters

proc	This is a pointer to the process control block of a process that has terminated.

need more

10.32.4.2 k_proc_create()

```
pcb_t * k_proc_create (
            pcb_t * parent )
```

Create a new child process, inheriting applicable properties from the parent.

Parameters

parent	This is a pointer to the process control block of the parent, from which it inherits.

Returns

Reference to the child PCB.

10.33 kernel.h

Go to the documentation of this file.

```
00001 #ifndef KERNEL_H
00002 #define KERNEL_H
00003
00004 #include <sys/types.h>  //needed for ssize_t, if we use ints, can remove
00005 #include "array.h"
00006 #include "bitmap.h"
00007 #include "clinkedlist.h"
00008 #include "globals.h"
00009 #include "spthread.h"
00010 #include "stdlib.h"
00011 #include "sys_call.h"
00012
00021 typedef enum {
00022   RUNNING,
00026   STOPPED,
00031   BLOCKED,
00036   ZOMBIED
00043 } process_state_t;
00044
00053 #define P_SIGSTOP 0
00054
00062 #define P_SIGCONT 1
00063
00071 #define P_SIGTER 2
00072
00078 typedef struct pcb_t {
00079   spthread_t handle;
00080   pid_t pid;
00081   pid_t ppid;
00082   DynamicPIDArray* child_pids;
00084   unsigned int priority : 2;
00086   process_state_t
00087       state;
00088   process_state_t initial_state;
00090   FD_Bitmap* open_fds;
00093   int input_fd;
00095   int output_fd;
00097   bool statechanged;
00100   int exit_status;
00102   int term_signal;
00104   bool waiting_for_change;
00106   pid_t waiting_on_pid;
00108   unsigned int ticks_to_wait;
00111   char* processname;
00113   bool is_bg;
00116   int job_num;
00119   char* cmd_name;
00121 } pcb_t;
00122
00130 pcb_t* k_proc_create(pcb_t* parent);
00131
00138 void k_proc_cleanup(pcb_t* proc);
00139
00140 #endif
```

10.34 src/util/pennfat_kernel.c File Reference

```
#include "pennfat_kernel.h"
#include "unistd.h"
```

Functions

- void zero_out_helper (int curr)
- int k_open (const char *fname, int mode)

  Open file name `fname` with the mode `mode`, and return a file descriptor to that file.
- bool is_file_name_valid (char *name)

  Checks whether the filename follows the POSIX standard.
- struct directory_entries * does_file_exist (const char *fname)

  helper that traverses root directory block by block to check if fname file exists return: the directory entry struct with name fname (NULL if not found) also moves fs_fd to the end of the root directory

- void move_to_open_de (bool found)

  Change the offset to the fs_fd to the first open directory entry.
- off_t does_file_exist2 (const char *fname)

  Helper function that given a files name, it outputs the offset to the directory entry or negative number if the file isn't found.
- int get_first_empty_fat_index ()

  Finds and returns the first empty fat index marked as 0x0000.
- void lseek_to_root_directory ()

  lseek the file system's offset to the start of the root directory.
- struct file_descriptor_st * get_file_descriptor (int fd)

  Return the file descriptor struct for the given file descriptor number.
- struct file_descriptor_st * create_file_descriptor (int fd, char *fname, int mode, int offset)

  Creates a new file_descriptor_st struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct file_descriptor_st.
- struct directory_entries * create_directory_entry (const char *name, uint32_t size, uint16_t firstBlock, uint8_t type, uint8_t perm, time_t mtime)

  Creates a new directory_entries struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct directory_entries.
- ssize_t k_read (int fd, int n, char *buf)

  Read n bytes from the file referenced by fd. On return, k_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error.
- void extend_fat (int start_index, int empty_fat_index)

  Extend the fat region of the given file (marked by the `start_index`) by one block.
- void write_one_byte_in_while (int bytes_left, int size, int true_offset, int *size_increment, int *bytes_written, int *current_offset, const char *str, uint16_t firstBlock)
- void update_directory_entry_after_write (struct directory_entries *curr_de, char *fname, int bytes_written)
- ssize_t k_write (int fd, const char *str, int n)

  Write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, k_write returns the number of bytes written, or a negative value on error.
- int k_count_fd_num (const char *name)

  Returns the number of currently open in the global_fd_table with the `name` as the fname.
- int k_close (int fd)

  Close the file fd and return 0 on success, or a negative value on failure.
- int k_unlink (const char *fname)

  Remove the file by freeing the FAT table and zeroing out previously existing data.
- off_t k_lseek (int fd, int offset, int whence)

  Reposition the file pointer for fd to the offset relative to whence. Refer to lseek(2) for how whence interacts with the file offset. If the newly calculated offset is creater than the current size of the file, the file expands to match that offset with the newly allocated space filled with 0s.
- void generate_permission (uint8_t perm, char **permissions)
- char * formatTime (time_t t)
- int k_ls (const char *filename, int fd)

  List the file filename in the current directory. If filename is NULL, list all files in the current directory.
- int k_update_timestamp (const char *source)

  Change the timestamp of the file to the current time.
- int k_rename (const char *source, const char *dest)

  Rename `source` to `dest`.
- int k_change_mode (const char *change, const char *filename)

  Change file mode bits.
- char * k_read_all (const char *filename, int *read_num)

  Reads all contents from the file with the file name `filename`. Outputs the contents as well as update `read_num` to the number of bytes read.
- char * k_get_fname_from_fd (int fd)

Returns the filename for the given file descriptor number.

- int k_cp_within_fat (char *source, char *dest)

  Copies the contents from `source` to `dest`. Both `source` and `dest` must be files within the PENNFAT system.

- int k_cp_to_host (char *source, char *host_dest)

  Copies the contents from `source` to `host_dest` `source` must be a file within the PENNFAT system. `host_dest` is a host system file.

- int k_cp_from_host (char *host_source, char *dest)

  Copies the contents from `host_source` to `dest`. `dest` must be a file within the PENNFAT system. `host_source` is a host system file.

Variables

- uint16_t * fat = NULL

  PennFAT filesystem that has been mounted to memory using the mmap(2).

- struct file_descriptor_st ** global_fd_table = NULL

  Kernel level global file descriptor table that stores all file descriptor that has been created through out the program's runtime.

- int fs_fd = -1

  File descriptor number (host system level) for the filesystem that has been mounted to the program.

- int block_size = 0

  Block size of the currently mounted filesystem that is defined during the mkfs process.

- int fat_size = 0

  FAT region size of the currently mounted filesystem.

- int num_fat_entries = 0

  Calculated value of the total number of FAT entries within the currently mounted filesystem.

- int data_size = 0

  Calculated data region size of the currently mounted filesystem.

- int fd_counter = 3

10.34.1 Function Documentation

10.34.1.1 create_directory_entry()

```
struct directory_entries * create_directory_entry (
            const char * name,
            uint32_t size,
            uint16_t firstBlock,
            uint8_t type,
            uint8_t perm,
            time_t mtime )
```

Creates a new directory_entries struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct directory_entries.

Parameters

name	Name of the file.
size	Size of the current file.
firstBlock	First FAT block number.
type	Type of the file.
perm	Permission of the file.
mtime	Last modified time.

Returns

A newly created directory_entries struct. NULL on memory allocation error.

10.34.1.2 create_file_descriptor()

```
struct file_descriptor_st * create_file_descriptor (
            int fd,
            char * fname,
            int mode,
            int offset )
```

Creates a new file_descriptor_st struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct file_descriptor_st.

Parameters

fd	File descriptor number.
fname	Name of the file.
mode	Either F_WRITE, F_READ, F_APPEND.
offset	Offset to the start of the file.

Returns

A newly created file_descriptor_st struct. NULL on memory allocation error.

10.34.1.3 does_file_exist()

```
struct directory_entries * does_file_exist (
            const char * fname )
```

helper that traverses root directory block by block to check if fname file exists return: the directory entry struct with name fname (NULL if not found) also moves fs_fd to the end of the root directory

Parameters

fname	Name of the file that we want to check.

10.34.1.4 does_file_exist2()

```
off_t does_file_exist2 (
            const char * fname )
```

Helper function that given a files name, it outputs the offset to the directory entry or negative number if the file isn't found.

Parameters

fname	Name of the file that we want to check.

10.34.1.5 extend_fat()

```
void extend_fat (
            int start_index,
            int empty_fat_index )
```

Extend the fat region of the given file (marked by the `start_index`) by one block.

Parameters

start_index	Start fat index for the given file.
empty_fat_index	The first empty index of the current FAT table. Should be calculated using get_first_empty_fat_index().

10.34.1.6 formatTime()

```
char * formatTime (
            time_t t )
```

10.34.1.7 generate_permission()

```
void generate_permission (
            uint8_t perm,
            char ** permissions )
```

10.34.1.8 get_file_descriptor()

```
struct file_descriptor_st * get_file_descriptor (
            int fd )
```

Return the file descriptor struct for the given file descriptor number.

Parameters

fd	File descriptor number.

10.34.1.9 get_first_empty_fat_index()

```
int get_first_empty_fat_index ( )
```

Finds and returns the first empty fat index marked as 0x0000.

Returns

first empty fat index.

10.34.1.10 is_file_name_valid()

```
bool is_file_name_valid (
            char * name )
```

Checks whether the filename follows the POSIX standard.

Parameters

name	Filename.

Returns

True if valid. False otherwise.

10.34.1.11 k_change_mode()

```
int k_change_mode (
            const char * change,
            const char * filename )
```

Change file mode bits.

The operator + causes the selected file mode bits to be added to the existing file mode bits of each file; - causes them to be removed.

Parameters

change	String that determines how the bits are modified.
filename	Name of the file.

Returns

1 on success. Negative number on failure.

10.34.1.12 k_close()

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

Close the file fd and return 0 on success, or a negative value on failure.

Parameters

fd	File descriptor number that needs to be closed

Returns

0 on success, or a negative value on failure.

10.34.1.13 k_count_fd_num()

```
int k_count_fd_num (
            const char * name )
```

Returns the number of currently open in the global_fd_table with the `name` as the fname.

Parameters

name	Name of the file that we want to check.

10.34.1.14 k_cp_from_host()

```
int k_cp_from_host (
            char * host_source,
            char * dest )
```

Copies the contents from `host_source` to `dest`. `dest` must be a file within the PENNFAT system. `host_source` is a host system file.

`host_source` must exist. If `dest` does not exist, it will be newly created.

Parameters

host_source	File name of source. Must be a host system file.
dest	File name of dest. Must be a PennFAT file.

Returns

1 on success. Negative number on failure.

10.34.1.15 k_cp_to_host()

```
int k_cp_to_host (
            char * source,
            char * host_dest )
```

Copies the contents from `source` to `host_dest` `source` must be a file within the PENNFAT system. `host_dest` is a host system file.

`source` must exist. If `host_dest` does not exist, it will be newly created.

Parameters

source	File name of source. Must be a PennFAT file.
host_dest	File name of dest. Must be a host system file.

Returns

1 on success. Negative number on failure.

10.34.1.16 k_cp_within_fat()

```
int k_cp_within_fat (
            char * source,
            char * dest )
```

Copies the contents from `source` to `dest`. Both `source` and `dest` must be files within the PENNFAT system.

`source` must exist. If `dest` does not exist, it will be newly created.

Parameters

source	File name of source. Must be a PennFAT file.
dest	File name of dest. Must be a PennFAT file.

Returns

1 on success. Negative number on failure.

10.34.1.17 k_get_fname_from_fd()

```
char * k_get_fname_from_fd (
            int fd )
```

Returns the filename for the given file descriptor number.

Parameters

fd	The file descriptor number.

Returns

The file name of the `fd` . NULL is `fd` is invalid.

10.34.1.18 k_ls()

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

List the file filename in the current directory. If filename is NULL, list all files in the current directory.

Similar to posix ls.

Parameters

filename	Optional parameter. If specified, ls data for the specified file is printed
fd	The file descriptor you want to write the result of k_ls to

Returns

1 on success, negative value on failure

10.34.1.19 k_lseek()

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

Reposition the file pointer for fd to the offset relative to whence. Refer to lseek(2) for how whence interacts with the file offset. If the newly calculated offset is creater than the current size of the file, the file expands to match that offset with the newly allocated space filled with 0s.

Parameters

fd	File descriptor number
offset	Offset value
whence	F_SEEK_SET, F_SEEK_CUR, and F_SEEK_END. Follows the lseek(2) whence mode.

Returns

off_t Newly calculated offset for `fd`

10.34.1.20 k_open()

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

Open file name `fname` with the mode `mode`, and return a file descriptor to that file.

This function opens a file specified by the file name `fname` in the mode specified by `mode` and returns a file descriptor associated with the open file that can be used for subsequent file operations.

Parameters

fname	The name of the file to open. See POSIX standard for allowed names.
mode	The mode with which to open the file. This should specify the access mode (e.g., read, write) and other flags as defined by the operating system. Allowed modes are: write (`F_WRITE`), read (`F_READ`), and append (`F_APPEND`).

Returns

int A non-negative file descriptor on success, or -1 on error and `errno` set.

Note

The `mode` parameter may only be `F_WRITE`, `F_READ`, or `F_APPEND`. Note that despite their names, write and append support both reading and writing. `F_APPEND`'s file pointer will point to the end of the file rather than the beginning. Both `F_WRITE` and `F_APPEND` will create the named file if it does not already exist.

See also

https://www.ibm.com/docs/en/zos/3.1.0?topic=locales-posix-portable-file-name-charac

Possible values of `errno` are:

- `EACCES` : // need to fill these in, will expand as further progress
- `ENAMETOOLONG` :

10.34.1.21 k_read()

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

Read n bytes from the file referenced by fd. On return, k_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error.

Parameters

fd	File descriptor number we are reading from
n	Number of bytes we are reading from `fd`
buf	Buffer where we store the read value

Returns

ssize_t the number of bytes read, 0 if EOF is reached, or a negative number on error

10.34.1.22 k_read_all()

```
char * k_read_all (
            const char * filename,
            int * read_num )
```

Reads all contents from the file with the file name `filename`. Outputs the contents as well as update `read_num` to the number of bytes read.

Parameters

filename	Name of the file we want to read from.
read_num	Pointer to an integer variable that will store the number of bytes read.

Returns

All contents of `filename` in char* format.

10.34.1.23 k_rename()

```
int k_rename (
            const char * source,
            const char * dest )
```

Rename `source` to `dest`.

Parameters

source	Source file name.
dest	Destination file name.

Returns

1 on success. Negative number on failure.

10.34.1.24 k_unlink()

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

Remove the file by freeing the FAT table and zeroing out previously existing data.

Parameters

fname	Name of the file we want to remove.

Returns

1 on success. Negative value of failure.

10.34.1.25 k_update_timestamp()

```
int k_update_timestamp (
            const char * source )
```

Change the timestamp of the file to the current time.

Parameters

source	Source file name.

Returns

1 on success. Negative number on failure.

10.34.1.26 k_write()

```
ssize_t k_write (
            int fd,
            const char * str,
            int n )
```

Write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, k_write returns the number of bytes written, or a negative value on error.

Parameters

fd	File descriptor number we are reading to
str	Provided string we want to write to `fd`
n	Number of bytes we are writing

Returns

ssize_t number of bytes written, or a negative value on error.

10.34.1.27 lseek_to_root_directory()

```
void lseek_to_root_directory ( )
```

lseek the file system's offset to the start of the root directory.

10.34.1.28 move_to_open_de()

```
void move_to_open_de (
            bool found )
```

Change the offset to the fs_fd to the first open directory entry.

Parameters

found	

10.34.1.29 update_directory_entry_after_write()

```
void update_directory_entry_after_write (
            struct directory_entries * curr_de,
            char * fname,
            int bytes_written )
```

10.34.1.30 write_one_byte_in_while()

```
void write_one_byte_in_while (
            int bytes_left,
            int size,
            int true_offset,
            int * size_increment,
            int * bytes_written,
            int * current_offset,
            const char * str,
            uint16_t firstBlock )
```

10.34.1.31 zero_out_helper()

```
void zero_out_helper (
            int curr )
```

10.34.2 Variable Documentation

10.34.2.1 block_size

```
int block_size = 0
```

Block size of the currently mounted filesystem that is defined during the mkfs process.

10.34.2.2 data_size

```
int data_size = 0
```

Calculated data region size of the currently mounted filesystem.

10.34.2.3 fat

```
uint16_t* fat = NULL
```

PennFAT filesystem that has been mounted to memory using the mmap(2).

10.34.2.4 fat_size

```
int fat_size = 0
```

FAT region size of the currently mounted filesystem.

10.34.2.5 fd_counter

```
int fd_counter = 3
```

10.34.2.6 fs_fd

```
int fs_fd = -1
```

File descriptor number (host system level) for the filesystem that has been mounted to the program.

10.34.2.7 global_fd_table

```
struct file_descriptor_st** global_fd_table = NULL
```

Kernel level global file descriptor table that stores all file descriptor that has been created through out the program's runtime.

10.34.2.8 num_fat_entries

```
int num_fat_entries = 0
```

Calculated value of the total number of FAT entries within the currently mounted filesystem.

10.35 src/util/pennfat_kernel.h File Reference

```
#include <stdint.h>
#include <sys/types.h>
#include "../pennfat.h"
#include "spthread.h"
```

Data Structures

- struct directory_entries

  This structure stores all required information about the directory entries that are stored in the root directory.
- struct file_descriptor_st

  This structure stores all required information about the file descriptor.

Macros

- #define F_READ 0

  open the file for reading only
- #define F_WRITE 1

  writing and reading, truncates if the file exists, or creates it if it does not exist. Only one instance of a file can be opened in F_WRITE mode; error if attempted to open a file in F_WRITE mode more than once
- #define F_APPEND 2

  open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file.
- #define MAX_FD_NUM 1024

  Size of the global_fd_table.
- #define FILE_NOT_FOUND -1

  Error number for when there does not exist a file with the given file name.
- #define INVALID_FILE_NAME -2

  Error number for when the file name doesn't follow the POSIX standard.
- #define MULTIPLE_F_WRITE -3

  Error number for when trying to open more than one file descriptor in F_WRITE / F_APPEND mode.
- #define WRONG_PERMISSION -4

  Error number for when trying to use the file descriptor in an invalid way such as writing to F_READ file descriptor.
- #define SYSTEM_ERROR -5

  Error number for when C level system function fails.
- #define FILE_DELETED -6

  Error number for when trying to access or use a deleted file.
- #define INVALID_FILE_DESCRIPTOR -7

  Error number for when trying to access or use a invalid file descriptor.
- #define FILE_IN_USE -8

  Error number for when delete a file that is used by some other processes.
- #define INVALID_PARAMETERS -9

  Error number for when the parameter given to the function is invalid.
- #define FS_NOT_MOUNTED -10

  Error number for when the filesystem is not mounted but tries to access or use the file system.
- #define INVALID_CHMOD -11

  Error number for when the resulting file mode/permission is invalid.
- #define SOURCE_FILE_NO_READ_PERM -12

  Error number for cp when source file doesn't have read permission.
- #define DEST_FILE_NO_WRITE_PERM -13

  Error number for cp when dest file doesn't have write permission.

Enumerations

- enum Whence { F_SEEK_SET , F_SEEK_CUR , F_SEEK_END }

  Defines how the offset will be calculated when using the k_lseek method. For more detail, refer to lseek(2).

Functions

- struct file_descriptor_st * create_file_descriptor (int fd, char *fname, int mode, int offset)

  Creates a new file_descriptor_st struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct file_descriptor_st.
- struct directory_entries * create_directory_entry (const char *name, uint32_t size, uint16_t firstBlock, uint8_t type, uint8_t perm, time_t mtime)

  Creates a new directory_entries struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct directory_entries.
- void lseek_to_root_directory ()

  lseek the file system's offset to the start of the root directory.
- void extend_fat (int start_index, int empty_fat_index)

  Extend the fat region of the given file (marked by the `start_index`) by one block.
- int get_first_empty_fat_index ()

  Finds and returns the first empty fat index marked as 0x0000.
- void move_to_open_de (bool found)

  Change the offset to the fs_fd to the first open directory entry.
- struct directory_entries * does_file_exist (const char *fname)

  helper that traverses root directory block by block to check if fname file exists return: the directory entry struct with name fname (NULL if not found) also moves fs_fd to the end of the root directory
- off_t does_file_exist2 (const char *fname)

  Helper function that given a files name, it outputs the offset to the directory entry or negative number if the file isn't found.
- int k_count_fd_num (const char *name)

  Returns the number of currently open in the global_fd_table with the `name` as the fname.
- struct file_descriptor_st * get_file_descriptor (int fd)

  Return the file descriptor struct for the given file descriptor number.
- int k_open (const char *fname, int mode)

  Open file name `fname` with the mode `mode`, and return a file descriptor to that file.
- ssize_t k_read (int fd, int n, char *buf)

  Read n bytes from the file referenced by fd. On return, k_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error.
- ssize_t k_write (int fd, const char *str, int n)

  Write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, k_write returns the number of bytes written, or a negative value on error.
- int k_close (int fd)

  Close the file fd and return 0 on success, or a negative value on failure.
- int k_unlink (const char *fname)

  Remove the file by freeing the FAT table and zeroing out previously existing data.
- off_t k_lseek (int fd, int offset, int whence)

  Reposition the file pointer for fd to the offset relative to whence. Refer to lseek(2) for how whence interacts with the file offset. If the newly calculated offset is creater than the current size of the file, the file expands to match that offset with the newly allocated space filled with 0s.
- int k_ls (const char *filename, int fd)

  List the file filename in the current directory. If filename is NULL, list all files in the current directory.
- int k_rename (const char *source, const char *dest)

  Rename `source` to `dest`.
- int k_update_timestamp (const char *source)

  Change the timestamp of the file to the current time.
- int k_change_mode (const char *change, const char *filename)

  Change file mode bits.
- char * k_read_all (const char *filename, int *read_num)

Reads all contents from the file with the file name `filename`. Outputs the contents as well as update `read_num` to the number of bytes read.

- bool is_file_name_valid (char *name)

  Checks whether the filename follows the POSIX standard.
- char * k_get_fname_from_fd (int fd)

  Returns the filename for the given file descriptor number.
- int k_cp_within_fat (char *source, char *dest)

  Copies the contents from `source` to `dest`. Both `source` and `dest` must be files within the PENNFAT system.
- int k_cp_to_host (char *source, char *host_dest)

  Copies the contents from `source` to `host_dest` `source` must be a file within the PENNFAT system. `host_dest` is a host system file.
- int k_cp_from_host (char *host_source, char *dest)

  Copies the contents from `host_source` to `dest`. `dest` must be a file within the PENNFAT system. `host_source` is a host system file.

Variables

- uint16_t * fat

  PennFAT filesystem that has been mounted to memory using the mmap(2).
- struct file_descriptor_st ** global_fd_table

  Kernel level global file descriptor table that stores all file descriptor that has been created through out the program's runtime.
- int fs_fd

  File descriptor number (host system level) for the filesystem that has been mounted to the program.
- int block_size

  Block size of the currently mounted filesystem that is defined during the mkfs process.
- int fat_size

  FAT region size of the currently mounted filesystem.
- int num_fat_entries

  Calculated value of the total number of FAT entries within the currently mounted filesystem.
- int data_size

  Calculated data region size of the currently mounted filesystem.

10.35.1 Macro Definition Documentation

10.35.1.1 DEST_FILE_NO_WRITE_PERM

```
#define DEST_FILE_NO_WRITE_PERM -13
```

Error number for cp when dest file doesn't have write permission.

10.35.1.2 F_APPEND

```
#define F_APPEND 2
```

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

10.35.1.3 F_READ

```
#define F_READ 0
```

open the file for reading only

10.35.1.4 F_WRITE

```
#define F_WRITE 1
```

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

10.35.1.5 FILE_DELETED

```
#define FILE_DELETED -6
```

Error number for when trying to access or use a deleted file.

10.35.1.6 FILE_IN_USE

```
#define FILE_IN_USE -8
```

Error number for when delete a file that is used by some other processes.

10.35.1.7 FILE_NOT_FOUND

```
#define FILE_NOT_FOUND -1
```

Error number for when there does not exist a file with the given file name.

10.35.1.8 FS_NOT_MOUNTED

```
#define FS_NOT_MOUNTED -10
```

Error number for when the filesystem is not mounted but tries to access or use the file system.

10.35.1.9 INVALID_CHMOD

```
#define INVALID_CHMOD -11
```

Error number for when the resulting file mode/permission is invalid.

10.35.1.10 INVALID_FILE_DESCRIPTOR

```
#define INVALID_FILE_DESCRIPTOR -7
```

Error number for when trying to access or use a invalid file descriptor.

10.35.1.11 INVALID_FILE_NAME

```
#define INVALID_FILE_NAME -2
```

Error number for when the file name doesn't follow the POSIX standard.

10.35.1.12 INVALID_PARAMETERS

```
#define INVALID_PARAMETERS -9
```

Error number for when the parameter given to the function is invalid.

10.35.1.13 MAX_FD_NUM

```
#define MAX_FD_NUM 1024
```

Size of the global_fd_table.

10.35.1.14 MULTIPLE_F_WRITE

```
#define MULTIPLE_F_WRITE -3
```

Error number for when trying to open more than one file descriptor in F_WRITE / F_APPEND mode.

10.35.1.15 SOURCE_FILE_NO_READ_PERM

```
#define SOURCE_FILE_NO_READ_PERM -12
```

Error number for cp when source file doesn't have read permission.

10.35.1.16 SYSTEM_ERROR

```
#define SYSTEM_ERROR -5
```

Error number for when C level system function fails.

10.35.1.17 WRONG_PERMISSION

```
#define WRONG_PERMISSION -4
```

Error number for when trying to use the file descriptor in an invalid way such as writing to F_READ file descriptor.

10.35.2 Enumeration Type Documentation

10.35.2.1 Whence

```
enum Whence
```

Defines how the offset will be calculated when using the k_lseek method. For more detail, refer to lseek(2).

Enumerator

F_SEEK_SET	
F_SEEK_CUR	
F_SEEK_END	

10.35.3 Function Documentation

10.35.3.1 create_directory_entry()

```
struct directory_entries * create_directory_entry (
            const char * name,
            uint32_t size,
            uint16_t firstBlock,
            uint8_t type,
            uint8_t perm,
            time_t mtime )
```

Creates a new directory_entries struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct directory_entries.

Parameters

name	Name of the file.
size	Size of the current file.
firstBlock	First FAT block number.
type	Type of the file.
perm	Permission of the file.
mtime	Last modified time.

Returns

A newly created directory_entries struct. NULL on memory allocation error.

10.35.3.2 create_file_descriptor()

```
struct file_descriptor_st * create_file_descriptor (
            int fd,
            char * fname,
            int mode,
            int offset )
```

Creates a new file_descriptor_st struct, initialized with the values provided by the parameters. For more information of the parameters, refer to struct file_descriptor_st.

Parameters

fd	File descriptor number.
fname	Name of the file.
mode	Either F_WRITE, F_READ, F_APPEND.
offset	Offset to the start of the file.

Returns

A newly created file_descriptor_st struct. NULL on memory allocation error.

10.35.3.3 does_file_exist()

```
struct directory_entries * does_file_exist (
            const char * fname )
```

helper that traverses root directory block by block to check if fname file exists return: the directory entry struct with name fname (NULL if not found) also moves fs_fd to the end of the root directory

Parameters

fname	Name of the file that we want to check.

10.35.3.4 does_file_exist2()

```
off_t does_file_exist2 (
            const char * fname )
```

Helper function that given a files name, it outputs the offset to the directory entry or negative number if the file isn't found.

Parameters

fname	Name of the file that we want to check.

10.35.3.5 extend_fat()

```
void extend_fat (
            int start_index,
            int empty_fat_index )
```

Extend the fat region of the given file (marked by the `start_index`) by one block.

Parameters

start_index	Start fat index for the given file.
empty_fat_index	The first empty index of the current FAT table. Should be calculated using get_first_empty_fat_index().

10.35.3.6 get_file_descriptor()

```
struct file_descriptor_st * get_file_descriptor (
            int fd )
```

Return the file descriptor struct for the given file descriptor number.

Parameters

fd	File descriptor number.

10.35.3.7 get_first_empty_fat_index()

```
int get_first_empty_fat_index ( )
```

Finds and returns the first empty fat index marked as 0x0000.

Returns

first empty fat index.

10.35.3.8 is_file_name_valid()

```
bool is_file_name_valid (
            char * name )
```

Checks whether the filename follows the POSIX standard.

Parameters

name	Filename.

Returns

True if valid. False otherwise.

10.35.3.9 k_change_mode()

```
int k_change_mode (
            const char * change,
            const char * filename )
```

Change file mode bits.

The operator + causes the selected file mode bits to be added to the existing file mode bits of each file; - causes them to be removed.

Parameters

change	String that determines how the bits are modified.
filename	Name of the file.

Returns

1 on success. Negative number on failure.

10.35.3.10 k_close()

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

Close the file fd and return 0 on success, or a negative value on failure.

Parameters

fd	File descriptor number that needs to be closed

Returns

0 on success, or a negative value on failure.

10.35.3.11 k_count_fd_num()

```
int k_count_fd_num (
             const char * name )
```

Returns the number of currently open in the global_fd_table with the `name` as the fname.

Parameters

name	Name of the file that we want to check.

10.35.3.12 k_cp_from_host()

```
int k_cp_from_host (
             char * host_source,
             char * dest )
```

Copies the contents from `host_source` to `dest`. `dest` must be a file within the PENNFAT system. `host_source` is a host system file.

`host_source` must exist. If `dest` does not exist, it will be newly created.

Parameters

host_source	File name of source. Must be a host system file.
dest	File name of dest. Must be a PennFAT file.

Returns

1 on success. Negative number on failure.

10.35.3.13 k_cp_to_host()

```
int k_cp_to_host (
            char * source,
            char * host_dest )
```

Copies the contents from `source` to `host_dest` `source` must be a file within the PENNFAT system. `host_dest` is a host system file.

`source` must exist. If `host_dest` does not exist, it will be newly created.

Parameters

source	File name of source. Must be a PennFAT file.
host_dest	File name of dest. Must be a host system file.

Returns

1 on success. Negative number on failure.

10.35.3.14 k_cp_within_fat()

```
int k_cp_within_fat (
            char * source,
            char * dest )
```

Copies the contents from `source` to `dest`. Both `source` and `dest` must be files within the PENNFAT system.

`source` must exist. If `dest` does not exist, it will be newly created.

Parameters

source	File name of source. Must be a PennFAT file.
dest	File name of dest. Must be a PennFAT file.

Returns

1 on success. Negative number on failure.

10.35.3.15 k_get_fname_from_fd()

```
char * k_get_fname_from_fd (
            int fd )
```

Returns the filename for the given file descriptor number.

Parameters

fd	The file descriptor number.

Returns

The file name of the `fd` . NULL is `fd` is invalid.

10.35.3.16 k_ls()

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

List the file filename in the current directory. If filename is NULL, list all files in the current directory.

Similar to posix ls.

Parameters

filename	Optional parameter. If specified, ls data for the specified file is printed
fd	The file descriptor you want to write the result of k_ls to

Returns

1 on success, negative value on failure

10.35.3.17 k_lseek()

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

Reposition the file pointer for fd to the offset relative to whence. Refer to lseek(2) for how whence interacts with the file offset. If the newly calculated offset is creater than the current size of the file, the file expands to match that offset with the newly allocated space filled with 0s.

Parameters

fd	File descriptor number
offset	Offset value
whence	F_SEEK_SET, F_SEEK_CUR, and F_SEEK_END. Follows the lseek(2) whence mode.

Returns

off_t Newly calculated offset for `fd`

10.35.3.18 k_open()

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

Open file name `fname` with the mode `mode`, and return a file descriptor to that file.

This function opens a file specified by the file name `fname` in the mode specified by `mode` and returns a file descriptor associated with the open file that can be used for subsequent file operations.

Parameters

fname	The name of the file to open. See POSIX standard for allowed names.
mode	The mode with which to open the file. This should specify the access mode (e.g., read, write) and other flags as defined by the operating system. Allowed modes are: write (`F_WRITE`), read (`F_READ`), and append (`F_APPEND`).

Returns

int A non-negative file descriptor on success, or -1 on error and `errno` set.

Note

The `mode` parameter may only be `F_WRITE`, `F_READ`, or `F_APPEND`. Note that despite their names, write and append support both reading and writing. `F_APPEND`'s file pointer will point to the end of the file rather than the beginning. Both `F_WRITE` and `F_APPEND` will create the named file if it does not already exist.

See also

https://www.ibm.com/docs/en/zos/3.1.0?topic=locales-posix-portable-file-name-charac

Possible values of `errno` are:

- `EACCES` : // need to fill these in, will expand as further progress
- `ENAMETOOLONG` :

10.35.3.19 k_read()

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

Read n bytes from the file referenced by fd. On return, k_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error.

Parameters

fd	File descriptor number we are reading from
n	Number of bytes we are reading from `fd`
buf	Buffer where we store the read value

Returns

ssize_t the number of bytes read, 0 if EOF is reached, or a negative number on error

10.35.3.20 k_read_all()

```
char * k_read_all (
            const char * filename,
            int * read_num )
```

Reads all contents from the file with the file name `filename`. Outputs the contents as well as update `read_num` to the number of bytes read.

Parameters

filename	Name of the file we want to read from.
read_num	Pointer to an integer variable that will store the number of bytes read.

Returns

All contents of `filename` in char* format.

10.35.3.21 k_rename()

```
int k_rename (
            const char * source,
            const char * dest )
```

Rename `source` to `dest`.

Parameters

source	Source file name.
dest	Destination file name.

Returns

1 on success. Negative number on failure.

10.35.3.22 k_unlink()

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

Remove the file by freeing the FAT table and zeroing out previously existing data.

Parameters

fname	Name of the file we want to remove.

Returns

1 on success. Negative value of failure.

10.35.3.23 k_update_timestamp()

```
int k_update_timestamp (
            const char * source )
```

Change the timestamp of the file to the current time.

Parameters

source	Source file name.

Returns

1 on success. Negative number on failure.

10.35.3.24 k_write()

```
ssize_t k_write (
            int fd,
            const char * str,
            int n )
```

Write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, k_write returns the number of bytes written, or a negative value on error.

Parameters

fd	File descriptor number we are reading to
str	Provided string we want to write to `fd`
n	Number of bytes we are writing

Returns

ssize_t number of bytes written, or a negative value on error.

10.35.3.25 lseek_to_root_directory()

```
void lseek_to_root_directory ( )
```

lseek the file system's offset to the start of the root directory.

10.35.3.26 move_to_open_de()

```
void move_to_open_de (
            bool found )
```

Change the offset to the fs_fd to the first open directory entry.

Parameters

found	

10.35.4 Variable Documentation

10.35.4.1 block_size

```
int block_size  [extern]
```

Block size of the currently mounted filesystem that is defined during the mkfs process.

10.35.4.2 data_size

```
int data_size  [extern]
```

Calculated data region size of the currently mounted filesystem.

10.35.4.3 fat

```
uint16_t* fat  [extern]
```

PennFAT filesystem that has been mounted to memory using the mmap(2).

10.35.4.4 fat_size

```
int fat_size  [extern]
```

FAT region size of the currently mounted filesystem.

10.35.4.5 fs_fd

```
int fs_fd  [extern]
```

File descriptor number (host system level) for the filesystem that has been mounted to the program.

10.35.4.6 global_fd_table

```
struct file_descriptor_st** global_fd_table  [extern]
```

Kernel level global file descriptor table that stores all file descriptor that has been created through out the program's runtime.

10.35.4.7 num_fat_entries

```
int num_fat_entries  [extern]
```

Calculated value of the total number of FAT entries within the currently mounted filesystem.

10.36 pennfat_kernel.h

Go to the documentation of this file.

```
00001 #ifndef PENNFAT_KERNEL_H
00002 #define PENNFAT_KERNEL_H
00003
00004 #include <stdint.h>
00005 #include <sys/types.h>  //needed for ssize_t, if we use ints, can remove
00006 #include "../pennfat.h"
00007 #include "spthread.h"
00008
00009 /*********************************************
00010  *  PENNFAT MACRO DEFINITION
00011  *********************************************/
00012
00017 #define F_READ 0
00018
00024 #define F_WRITE 1
00025
00030 #define F_APPEND 2
00031
00035 #define MAX_FD_NUM 1024
00036
00041 #define FILE_NOT_FOUND -1
00042
00046 #define INVALID_FILE_NAME -2
00047
00052 #define MULTIPLE_F_WRITE -3
00053
00058 #define WRONG_PERMISSION -4
00059
00063 #define SYSTEM_ERROR -5
00064
00068 #define FILE_DELETED -6
00069
00074 #define INVALID_FILE_DESCRIPTOR -7
00075
00080 #define FILE_IN_USE -8
00081
00085 #define INVALID_PARAMETERS -9
00086
00091 #define FS_NOT_MOUNTED -10
00092
00096 #define INVALID_CHMOD -11
00097
00101 #define SOURCE_FILE_NO_READ_PERM -12
00102
00106 #define DEST_FILE_NO_WRITE_PERM -13
00107
00113 enum Whence { F_SEEK_SET, F_SEEK_CUR, F_SEEK_END };
00114
00117 extern uint16_t* fat;
00118
00121 extern struct file_descriptor_st** global_fd_table;
00122
00125 extern int fs_fd;
00126
00129 extern int block_size;
00130
00132 extern int fat_size;
00133
00136 extern int num_fat_entries;
00137
00139 extern int data_size;
00140
00146 struct directory_entries {
00147   char name[32];
00151   uint32_t size;
00153   uint16_t firstBlock;
00156   uint8_t type;
00159   uint8_t perm;
00163   time_t mtime;
```

```
00166   uint8_t reserved[16];
00167 };
00168
00174 struct file_descriptor_st {
00175   int fd;
00177   char* fname;
00178   int mode;
00180   int offset;
00181   int ref_cnt;
00183 };
00184
00198 struct file_descriptor_st* create_file_descriptor(int fd,
00199                                                   char* fname,
00200                                                   int mode,
00201                                                   int offset);
00202
00218 struct directory_entries* create_directory_entry(const char* name,
00219                                                  uint32_t size,
00220                                                  uint16_t firstBlock,
00221                                                  uint8_t type,
00222                                                  uint8_t perm,
00223                                                  time_t mtime);
00224
00229 void lseek_to_root_directory();
00230
00240 void extend_fat(int start_index, int empty_fat_index);
00241
00247 int get_first_empty_fat_index();
00248
00254 void move_to_open_de(bool found);
00255
00263 struct directory_entries* does_file_exist(const char* fname);
00264
00271 off_t does_file_exist2(const char* fname);
00272
00279 int k_count_fd_num(const char* name);
00280
00287 struct file_descriptor_st* get_file_descriptor(int fd);
00288
00289 /*********************************************
00290  *  PENNFAT KERNEL LEVEL FUNCTIONS
00291  *********************************************/
00292
00328 int k_open(const char* fname, int mode);
00329
00341 ssize_t k_read(int fd, int n, char* buf);
00342
00354 ssize_t k_write(int fd, const char* str, int n);
00355
00363 int k_close(int fd);
00364
00373 int k_unlink(const char* fname);
00374
00388 off_t k_lseek(int fd, int offset, int whence);
00389
00402 int k_ls(const char* filename, int fd);
00403
00412 int k_rename(const char* source, const char* dest);
00413
00421 int k_update_timestamp(const char* source);
00422
00434 int k_change_mode(const char* change, const char* filename);
00435
00447 char* k_read_all(const char* filename, int* read_num);
00448
00456 bool is_file_name_valid(char* name);
00457
00465 char* k_get_fname_from_fd(int fd);
00466
00478 int k_cp_within_fat(char* source, char* dest);
00479
00492 int k_cp_to_host(char* source, char* host_dest);
00493
00506 int k_cp_from_host(char* host_source, char* dest);
00507
00508 #endif
```

10.37 src/util/prioritylist.c File Reference

```
#include "prioritylist.h"
#include <stdlib.h>
```

Functions

- PList * init_priority (void)
- void add_priority (PList *list, unsigned int priority)
- bool remove_priority (PList *list, unsigned int priority)
- void free_plist (PList *list)

10.37.1 Function Documentation

10.37.1.1 add_priority()

```
void add_priority (
            PList * list,
            unsigned int priority )
```

Adds a new process to the circular linked list.

Parameters

list	Pointer to the circular linked list.
process	Pointer to the process control block (pcb_t) to add.

10.37.1.2 free_plist()

```
void free_plist (
            PList * list )
```

Frees all nodes and their associated processes in a circular linked list, then frees the list itself.

Parameters

list	Pointer to the circular linked list to free.

10.37.1.3 init_priority()

```
PList * init_priority (
            void  )
```

Initializes a circular linked list.

Returns

CircularList* Pointer to the newly initialized list.

10.37.1.4 remove_priority()

```
bool remove_priority (
            PList * list,
            unsigned int priority )
```

Removes a process from the circular linked list by its PID.

Parameters

list	Pointer to the circular linked list.
pid	PID of the process to remove.

Returns

bool true if the process was successfully removed, false otherwise.

10.38 src/util/prioritylist.h File Reference

```
#include <stdbool.h>
```

Data Structures

- struct PNode
- struct PList

Typedefs

- typedef struct PNode PNode

Functions

- PList * init_priority (void)
- void add_priority (PList *list, unsigned int priority)
- bool remove_priority (PList *list, unsigned int priority)
- void free_plist (PList *list)

10.38.1 Typedef Documentation

10.38.1.1 PNode

```
typedef struct PNode PNode
```

10.38.2 Function Documentation

10.38.2.1 add_priority()

```
void add_priority (
            PList * list,
            unsigned int priority )
```

Adds a new process to the circular linked list.

Parameters

list	Pointer to the circular linked list.
process	Pointer to the process control block (pcb_t) to add.

10.38.2.2 free_plist()

```
void free_plist (
            PList * list )
```

Frees all nodes and their associated processes in a circular linked list, then frees the list itself.

Parameters

list	Pointer to the circular linked list to free.

10.38.2.3 init_priority()

```
PList * init_priority (
            void  )
```

Initializes a circular linked list.

Returns

CircularList* Pointer to the newly initialized list.

10.38.2.4 remove_priority()

```
bool remove_priority (
            PList * list,
            unsigned int priority )
```

Removes a process from the circular linked list by its PID.

Parameters

list	Pointer to the circular linked list.
pid	PID of the process to remove.

Returns

bool true if the process was successfully removed, false otherwise.

10.39 prioritylist.h

Go to the documentation of this file.

```
00001 #ifndef PLIST_H
00002 #define PLIST_H
00003
00004 #include <stdbool.h>
00005
00010 typedef struct PNode {
00011   unsigned int
00012       priority : 2;
00013   struct PNode* next;
00014 } PNode;
00015
00021 typedef struct {
00022   PNode* head;
00023   unsigned int size;
00024 } PList;
00025
00030 PList* init_priority(void);
00031
00037 void add_priority(PList* list, unsigned int priority);
00038
00045 bool remove_priority(PList* list, unsigned int priority);
00046
00053 void free_plist(PList* list);
00054
00055 #endif  // SCHEDULER_LIST_H
```

10.40 src/util/shellbuiltins.c File Reference

```
#include "shellbuiltins.h"
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include "errno.h"
#include "pennos.h"
#include "sys_call.h"
#include "unistd.h"
```

Functions

- void * b_background_poll (void *arg)

  For each shell run, the background processors are checked and signaled.
- void * b_sleep (void *arg)

  Sleep for n seconds.
- void * b_busy (void *arg)

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

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

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

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

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

  Lists all available commands.
- void * b_nice (void *arg)

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

- void * b_nice_pid (void *arg)

  Adjust the priority level of an existing process.
- void * b_orphan_child (void *arg)

  Helper for orphanify.
- void * b_orphanify (void *arg)

  Used to test orphanifying functionality of your kernel.
- void * b_zombie_child (void *arg)

  Helper for zombify.
- void * b_zombify (void *arg)

  Used to test zombifying functionality of your kernel.
- void * b_logout (void *arg)

  Exits the shell and shutsdown PennOS.
- void * b_clear (void *arg)

  Clears the terminal.
- void * b_ls (void *arg)

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

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

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

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

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

10.40.1 Function Documentation

10.40.1.1 b_background_poll()

```
void * b_background_poll (
            void * arg )
```

For each shell run, the background processors are checked and signaled.

10.40.1.2 b_bg()

```
void * b_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)

10.40.1.3 b_busy()

```
void * b_busy (
            void * arg )
```

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

Example Usage: busy

10.40.1.4 b_cat()

```
void * b_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)

10.40.1.5 b_chmod()

```
void * b_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)

10.40.1.6 b_clear()

```
void * b_clear (
            void * arg )
```

Clears the terminal.

Example Usage: clear

10.40.1.7 b_cp()

```
void * b_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

10.40.1.8 b_echo()

```
void * b_echo (
            void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

10.40.1.9 b_fg()

```
void * b_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)

10.40.1.10 b_jobs()

```
void * b_jobs (
            void * arg )
```

Lists all jobs.

Example Usage: jobs

10.40.1.11 b_kill()

```
void * b_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)

10.40.1.12 b_logout()

```
void * b_logout (
            void * arg )
```

Exits the shell and shutsdown PennOS.

Example Usage: logout

10.40.1.13 b_ls()

```
void * b_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)

10.40.1.14 b_man()

```
void * b_man (
            void * arg )
```

Lists all available commands.

Example Usage: man

10.40.1.15 b_mv()

```
void * b_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

10.40.1.16 b_nice()

```
void * b_nice (
            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)

10.40.1.17 b_nice_pid()

```
void * b_nice_pid (
            void * arg )
```

Adjust the priority level of an existing process.

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

10.40.1.18 b_orphan_child()

```
void * b_orphan_child (
            void * arg )
```

Helper for orphanify.

10.40.1.19 b_orphanify()

```
void * b_orphanify (
            void * arg )
```

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

10.40.1.20 b_ps()

```
void * b_ps (
            void * arg )
```

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

Example Usage: ps

10.40.1.21 b_rm()

```
void * b_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

10.40.1.22 b_sleep()

```
void * b_sleep (
            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

10.40.1.23 b_touch()

```
void * b_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

10.40.1.24 b_zombie_child()

```
void * b_zombie_child (
            void * arg )
```

Helper for zombify.

10.40.1.25 b_zombify()

```
void * b_zombify (
            void * arg )
```

Used to test zombifying functionality of your kernel.

Example Usage: zombify

10.41 src/util/shellbuiltins.h File Reference

```
#include "error.h"
```

Functions

- void * b_background_poll (void *arg)

  For each shell run, the background processors are checked and signaled.
- void * b_cat (void *arg)

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

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

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

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

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

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

  Rename a file. If the `dst_file` file already exists, overwrite it.
- void * b_cp (void *arg)
- void * b_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 * b_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 * b_ps (void *arg)

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

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

  Adjust the priority level of an existing process.
- void * b_man (void *arg)

  Lists all available commands.
- void * b_bg (void *arg)

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

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

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

  Exits the shell and shutsdown PennOS.
- void * b_clear (void *arg)

  Clears the terminal.
- void * b_zombify (void *arg)

  Used to test zombifying functionality of your kernel.
- void * b_zombie_child (void *arg)

  Helper for zombify.
- void * b_orphanify (void *arg)

  Used to test orphanifying functionality of your kernel.
- void * b_orphan_child (void *arg)

  Helper for orphanify.

10.41.1 Function Documentation

10.41.1.1 b_background_poll()

```
void * b_background_poll (
            void * arg )
```

For each shell run, the background processors are checked and signaled.

10.41.1.2 b_bg()

```
void * b_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)

10.41.1.3 b_busy()

```
void * b_busy (
            void * arg )
```

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

Example Usage: busy

10.41.1.4 b_cat()

```
void * b_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)

10.41.1.5 b_chmod()

```
void * b_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)

10.41.1.6 b_clear()

```
void * b_clear (
            void * arg )
```

Clears the terminal.

Example Usage: clear

10.41.1.7 b_cp()

```
void * b_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

10.41.1.8 b_echo()

```
void * b_echo (
            void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

10.41.1.9 b_fg()

```
void * b_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)

10.41.1.10 b_jobs()

```
void * b_jobs (
            void * arg )
```

Lists all jobs.

Example Usage: jobs

10.41.1.11 b_kill()

```
void * b_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)

10.41.1.12 b_logout()

```
void * b_logout (
            void * arg )
```

Exits the shell and shutsdown PennOS.

Example Usage: logout

10.41.1.13 b_ls()

```
void * b_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)

10.41.1.14 b_man()

```
void * b_man (
            void * arg )
```

Lists all available commands.

Example Usage: man

10.41.1.15 b_mv()

```
void * b_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

10.41.1.16 b_nice()

```
void * b_nice (
            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)

10.41.1.17 b_nice_pid()

```
void * b_nice_pid (
            void * arg )
```

Adjust the priority level of an existing process.

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

10.41.1.18 b_orphan_child()

```
void * b_orphan_child (
            void * arg )
```

Helper for orphanify.

10.41.1.19 b_orphanify()

```
void * b_orphanify (
            void * arg )
```

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

10.41.1.20 b_ps()

```
void * b_ps (
            void * arg )
```

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

Example Usage: ps

10.41.1.21 b_rm()

```
void * b_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

10.41.1.22 b_sleep()

```
void * b_sleep (
            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

10.41.1.23 b_touch()

```
void * b_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

10.41.1.24 b_zombie_child()

```
void * b_zombie_child (
            void * arg )
```

Helper for zombify.

10.41.1.25 b_zombify()

```
void * b_zombify (
            void * arg )
```

Used to test zombifying functionality of your kernel.

Example Usage: zombify

10.42 shellbuiltins.h

Go to the documentation of this file.

```
00001 #ifndef SHELL_BUILTINS
00002 #define SHELL_BUILTINS
00003 #include "error.h"
00004 // SHELL BUILTINS: Implemented using user and system level functions only!
00008 void* b_background_poll(void* arg);
00009
00020 void* b_cat(void* arg);
00021
00030 void* b_sleep(void* arg);
00031
00038 void* b_busy(void* arg);
00039
00045 void* b_echo(void* arg);
00046
00054 void* b_ls(void* arg);
00055
00062 void* b_touch(void* arg);
00063
00074 void* b_mv(void* arg);
00075
00086 void* b_cp(void* arg);
00087
00098 void* b_rm(void* arg);
00099
00114 void* b_chmod(void* arg);
00115
00122 void* b_ps(void* arg);
00123
00134 void* b_kill(void* arg);
00135
00136 // SHELL BUILTINS THAT DON'T SPAWN PROCESSES
00137
00145 void* b_nice(void* arg);
00146
00152 void* b_nice_pid(void* arg);
00153
00159 void* b_man(void* arg);
00160
00168 void* b_bg(void* arg);
00169
00177 void* b_fg(void* arg);
00178
00184 void* b_jobs(void* arg);
00185
00191 void* b_logout(void* arg);
00192
00198 void* b_clear(void* arg);
00199
00200 // SHELL BUILTINS TO TEST ZOMBIE + ORPHANS
00206 void* b_zombify(void* arg);
00207
00211 void* b_zombie_child(void* arg);
00212
00218 void* b_orphanify(void* arg);
00219
00223 void* b_orphan_child(void* arg);
00224
00225 #endif
```

10.43 src/util/spthread.c File Reference

```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdbool.h>
#include <stdlib.h>
#include "./spthread.h"
#include <stdio.h>
#include <string.h>
```

Data Structures

- struct spthread_fwd_args_st
- struct spthread_signal_args_st
- struct spthread_meta_st

Macros

- #define _GNU_SOURCE
- #define _XOPEN_SOURCE 700
- #define MILISEC_IN_NANO 100000
- #define SPTHREAD_RUNNING_STATE 0
- #define SPTHREAD_SUSPENDED_STATE 1
- #define SPTHREAD_TERMINATED_STATE 2
- #define SPTHREAD_SIG_SUSPEND -1
- #define SPTHREAD_SIG_CONTINUE -2

Typedefs

- typedef void *(* pthread_fn) (void *)
- typedef struct spthread_fwd_args_st spthread_fwd_args
- typedef struct spthread_signal_args_st spthread_signal_args
- typedef struct spthread_meta_st spthread_meta_t

Functions

- int spthread_create (spthread_t *thread, const pthread_attr_t *attr, pthread_fn start_routine, void *arg)
- int spthread_suspend (spthread_t thread)
- int spthread_suspend_self ()
- int spthread_continue (spthread_t thread)
- int spthread_cancel (spthread_t thread)
- bool spthread_self (spthread_t *thread)
- int spthread_join (spthread_t thread, void **retval)
- void spthread_exit (void *status)

10.43.1 Macro Definition Documentation

10.43.1.1 _GNU_SOURCE

```
#define _GNU_SOURCE
```

10.43.1.2 _XOPEN_SOURCE

```
#define _XOPEN_SOURCE 700
```

10.43.1.3 MILISEC_IN_NANO

```
#define MILISEC_IN_NANO 100000
```

10.43.1.4 SPTHREAD_RUNNING_STATE

```
#define SPTHREAD_RUNNING_STATE 0
```

10.43.1.5 SPTHREAD_SIG_CONTINUE

```
#define SPTHREAD_SIG_CONTINUE -2
```

10.43.1.6 SPTHREAD_SIG_SUSPEND

```
#define SPTHREAD_SIG_SUSPEND -1
```

10.43.1.7 SPTHREAD_SUSPENDED_STATE

```
#define SPTHREAD_SUSPENDED_STATE 1
```

10.43.1.8 SPTHREAD_TERMINATED_STATE

```
#define SPTHREAD_TERMINATED_STATE 2
```

10.43.2 Typedef Documentation

10.43.2.1 pthread_fn

```
typedef void *(* pthread_fn) (void *)
```

10.43.2.2 spthread_fwd_args

```
typedef struct spthread_fwd_args_st spthread_fwd_args
```

10.43.2.3 spthread_meta_t

```
typedef struct spthread_meta_st spthread_meta_t
```

10.43.2.4 spthread_signal_args

```
typedef struct spthread_signal_args_st spthread_signal_args
```

10.43.3 Function Documentation

10.43.3.1 spthread_cancel()

```
int spthread_cancel (
            spthread_t thread )
```

10.43.3.2 spthread_continue()

```
int spthread_continue (
            spthread_t thread )
```

10.43.3.3 spthread_create()

```
int spthread_create (
            spthread_t * thread,
            const pthread_attr_t * attr,
            pthread_fn start_routine,
            void * arg )
```

10.43.3.4 spthread_exit()

```
void spthread_exit (
            void * status )
```

10.43.3.5 spthread_join()

```
int spthread_join (
            spthread_t thread,
            void ** retval )
```

10.43.3.6 spthread_self()

```
bool spthread_self (
            spthread_t * thread )
```

10.43.3.7 spthread_suspend()

```
int spthread_suspend (
            spthread_t thread )
```

10.43.3.8 spthread_suspend_self()

```
int spthread_suspend_self ( )
```

10.44 src/util/spthread.h File Reference

```
#include <pthread.h>
#include <stdbool.h>
```

Data Structures

- struct spthread_st

Macros

- #define SIGPTHD SIGUSR1

Typedefs

- typedef struct spthread_meta_st spthread_meta_t
- typedef struct spthread_st spthread_t

Functions

- int spthread_create (spthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
- int spthread_suspend (spthread_t thread)
- int spthread_suspend_self ()
- int spthread_continue (spthread_t thread)
- int spthread_cancel (spthread_t thread)
- bool spthread_self (spthread_t *thread)
- int spthread_join (spthread_t thread, void **retval)
- void spthread_exit (void *status)

10.44.1 Macro Definition Documentation

10.44.1.1 SIGPTHD

```
#define SIGPTHD SIGUSR1
```

10.44.2 Typedef Documentation

10.44.2.1 spthread_meta_t

```
typedef struct spthread_meta_st spthread_meta_t
```

10.44.2.2 spthread_t

```
typedef struct spthread_st spthread_t
```

10.44.3 Function Documentation

10.44.3.1 spthread_cancel()

```
int spthread_cancel (
            spthread_t thread )
```

10.44.3.2 spthread_continue()

```
int spthread_continue (
            spthread_t thread )
```

10.44.3.3 spthread_create()

```
int spthread_create (
            spthread_t * thread,
            const pthread_attr_t * attr,
            void *(*)(void *) start_routine,
            void * arg )
```

10.44.3.4 spthread_exit()

```
void spthread_exit (
            void * status )
```

10.44.3.5 spthread_join()

```
int spthread_join (
            spthread_t thread,
            void ** retval )
```

10.44.3.6 spthread_self()

```
bool spthread_self (
            spthread_t * thread )
```

10.44.3.7 spthread_suspend()

```
int spthread_suspend (
            spthread_t thread )
```

10.44.3.8 spthread_suspend_self()

```
int spthread_suspend_self ( )
```

10.45 spthread.h

Go to the documentation of this file.

```
00001 #ifndef SPTHREAD_H_
00002 #define SPTHREAD_H_
00003
00004 #include <pthread.h>
00005 #include <stdbool.h>
00006
00007 // CAUTION: according to `man 7 pthread`:
00008 //
00009 //   On older Linux kernels, SIGUSR1 and SIGUSR2
00010 //   are used.  Applications must avoid the use of whichever set of
00011 //   signals is employed by the implementation.
00012 //
00013 // This may not work on other linux versions
00014
00015 // SIGNAL PTHREAD
00016 // NOTE: if within a created spthread you change
00017 // the behaviour of SIGUSR1, then you will not be able
00018 // to suspend and continue a spthread
00019 #define SIGPTHD SIGUSR1
00020
00021 // declares a struct, but the internals of the
00022 // struct cannot be seen by functions outside of spthread.c
00023 typedef struct spthread_meta_st spthread_meta_t;
00024
00025 // The spthread wrapper struct.
00026 // Sometimes you may have to access the inner pthread member
00027 // but you shouldn't need to do that
00028 typedef struct spthread_st {
00029   pthread_t thread;
00030   spthread_meta_t* meta;
00031 } spthread_t;
00032
00033 // NOTE:
00034 // None of these are signal safe
00035 // Also note that most of these functions are not safe to suspension,
00036 // meaning that if the thread calling these is an spthread and is suspended
00037 // in the middle of spthread_continue or spthread_suspend, then it may not work.
00038 //
00039 // Make sure that the calling thread cannot be suspended before calling these
00040 // functions. Exceptions to this are spthread_exit(), spthread_self() and if a
00041 // thread is continuing or suspending itself.
00042 // spthread_create:
00043 // this function works similar to pthread_create, except for two differences.
00044 // 1) the created pthread is able to be asychronously suspended, and continued
00045 //    using the functions:
00046 //      - spthread_suspend
00047 //      - spthread_continue
00048 // 2) The created pthread will be suspended before it executes the specified
00049 //    routine. It must first be continued with `spthread_continue` before
00050 //    it will start executing.
00051 //
00052 // It is worth noting that this function is not signal safe.
00053 // In other words, it should not be called from a signal handler.
00054 //
00055 // to avoid repetition, see pthread_create(3) for details
00056 // on arguments and return values as they are the same here.
00057 int spthread_create(spthread_t* thread,
00058                     const pthread_attr_t* attr,
00059                     void* (*start_routine)(void*),
00060                     void* arg);
00061
00062 // The spthread_suspend function will signal to the
00063 // specified thread to suspend execution.
00064 //
00065 // Calling spthread_suspend on an already suspended
00066 // thread does not do anything.
00067 //
00068 // It is worth noting that this function is not signal safe.
00069 // In other words, it should not be called from a signal handler.
00070 //
00071 // args:
00072 // - pthread_t thread: the thread we want to suspend
00073 //   This thread must be created using the spthread_create() function,
00074 //   if created by some other function, the behaviour is undefined.
00075 //
00076
00077 // returns:
00078 // - 0 on success
00079 // - EAGAIN if the thread could not be signaled
00080 // - ENOSYS if not supported on this system
00081 // - ESRCH if the thread specified is not a valid pthread
00082 int spthread_suspend(spthread_t thread);
```

```
00083
00084 // The spthread_suspend_self function will cause the calling
00085 // thread (which should be created by spthread_create) to suspend
00086 // itself.
00087 //
00088 // returns:
00089 // - 0 on success
00090 // - EAGAIN if the thread could not be signaled
00091 // - ENOSYS if not supported on this system
00092 // - ESRCH if the calling thread is not an spthread
00093 int spthread_suspend_self();
00094
00095 // The spthread_continue function will signal to the
00096 // specified thread to resume execution if suspended.
00097 //
00098 // Calling spthread_continue on an already non-suspended
00099 // thread does not do anything.
00100 //
00101 // It is worth noting that this function is not signal safe.
00102 // In other words, it should not be called from a signal handler.
00103 //
00104 // args:
00105 // - spthread_t thread: the thread we want to continue
00106 //   This thread must be created using the spthread_create() function,
00107 //   if created by some other function, the behaviour is undefined.
00108 //
00109 // returns:
00110 // - 0 on success
00111 // - EAGAIN if the thread could not be signaled
00112 // - ENOSYS if not supported on this system
00113 // - ESRCH if the thread specified is not a valid pthread
00114 int spthread_continue(spthread_t thread);
00115
00116 // The spthread_cancel function will send a
00117 // cancellation request to the specified thread.
00118 //
00119 // as of now, this function is identical to pthread_cancel(3)
00120 // so to avoid repitition, you should look there.
00121 //
00122 // Here are a few things that are worth highlighting:
00123 // - it is worth noting that it is a cancellation __request__
00124 //   the thread may not terminate immediately, instead the
00125 //   thread is checked whenever it calls a function that is
00126 //   marked as a cancellation point. At those points, it will
00127 //   start the cancellation procedure
00128 // - to make sure all things are de-allocated properly on
00129 //   normal exiting of the thread and when it is cancelled,
00130 //   you should mark a defered de-allocation with
00131 //   pthread_cleanup_push(3).
00132 //   consider the following example:
00133 //
00134 //     void* thread_routine(void* arg) {
00135 //        int* num = malloc(sizeof(int));
00136 //        pthread_cleanup_push(&free, num);
00137 //        return NULL;
00138 //     }
00139 //
00140 //    this program will allocate an integer on the heap
00141 //    and mark that data to be de-allocated on cleanup.
00142 //    This means that when the thread returns from the
00143 //    routine specified in spthread_create, free will
00144 //    be called on num. This will also happen if the thread
00145 //    is cancelled and not able to be exited normally.
00146 //
00147 //    Another function that should be used in conjunction
00148 //    is pthread_cleanup_pop(3). I will leave that
00149 //    to you to read more on.
00150 //
00151 // It is worth noting that this function is not signal safe.
00152 // In other words, it should not be called from a signal handler.
00153 //
00154 // args:
00155 // - spthread_t thread: the thread we want to cancel.
00156 //   This thread must be created using the spthread_create() function,
00157 //   if created by some other function, the behaviour is undefined.
00158 //
00159 // returns:
00160 // - 0 on success
00161 // - ESRCH if the thread specified is not a valid pthread
00162 int spthread_cancel(spthread_t thread);
00163
00164 // Can be called by a thread to get two peices of information:
00165 // 1. Whether or not the calling thread is an spthread (true or false)
00166 // 2. The spthread_t of the calling thread, if it is an spthread_t
00167 //
00168 // almost always the function will be called like this:
00169 // spthread_t self;
```

```
00170 // bool i_am_spthread = spthread_self(&self);
00171 //
00172 // args:
00173 // - spthread_t* thread: the output parameter to get the spthread_t
00174 //   representing the calling thread, if it is an spthread
00175 //
00176 // returns:
00177 // - true if the calling thread is an spthread_t
00178 // - false otherwise.
00179 bool spthread_self(spthread_t* thread);
00180
00181 // The equivalent of pthread_join but for spthread
00182 // To make sure all resources are cleaned up appropriately
00183 // spthreads that are created must at some ppoint have spthread_join
00184 // called on them. Do not use pthread_join on an spthread.
00185 //
00186 // to avoid repetition, see pthread_join(3) for details
00187 // on arguments and return values as they are the same as this function.
00188 int spthread_join(spthread_t thread, void** retval);
00189
00190 // The equivalent of pthread_exit but for spthread
00191 // spthread_exit must be used by spthreads instead of pthread_exit.
00192 // Otherwise, calls to spthread_join or other functions (like spthread_suspend)
00193 // may not work as intended.
00194 //
00195 // to avoid repetition, see pthread_exit(3) for details
00196 // on arguments and return values as they are the same as this function.
00197 void spthread_exit(void* status);
00198
00199 #endif  // SPTHREAD_H_
```

10.46 src/util/stress.c File Reference

```
#include "stress.h"
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include "kernel.h"
```

Functions

- void * hang (void *arg)
- void * nohang (void *arg)
- void * recur (void *arg)

10.46.1 Function Documentation

10.46.1.1 hang()

```
void * hang (
            void * arg )
```

10.46.1.2 nohang()

```
void * nohang (
            void * arg )
```

10.46.1.3 recur()

```
void * recur (
            void * arg )
```

10.47 src/util/stress.h File Reference

```
#include "sys_call.h"
#include "globals.h"
```

Functions

- void * hang (void *)
- void * nohang (void *)
- void * recur (void *)

10.47.1 Function Documentation

10.47.1.1 hang()

```
void * hang (
            void * arg )
```

10.47.1.2 nohang()

```
void * nohang (
            void * arg )
```

10.47.1.3 recur()

```
void * recur (
            void * arg )
```

10.48 stress.h

Go to the documentation of this file.

```
00001 #ifndef STRESS_H
00002 #define STRESS_H
00003 #include "sys_call.h"
00004 #include "globals.h"
00005 void* hang(void*);
00006 void* nohang(void*);
00007 void* recur(void*);
00008
00009 #endif
```

10.49 src/util/sys_call.c File Reference

```
#include "sys_call.h"
#include <unistd.h>
#include "stdio.h"
```

Functions

- char ** duplicate_argv (char *argv[])
- void free_argv (char *argv[])
- int get_arg_size (char *argv[])
- pid_t s_spawn (void *(*func)(void *), char *argv[], int fd0, int fd1)

  Create a child process that executes the function `func`. The child will retain some attributes of the parent.
- pid_t s_spawn_nice (void *(*func)(void *), char *argv[], int fd0, int fd1, unsigned int priority)

  Create a child process that executes the function `func`, with a specified priority. This is an exact copy of s_spawn except that the priority of the created process can be specificed at creation.
- pid_t s_waitpid (pid_t pid, int *wstatus, bool nohang)

  Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.
- int s_resume_block (pid_t pid)

  Blocks process with command "sleep" that has been stopped before.
- int s_kill (pid_t pid, int signal)

  Send a signal to a particular process.
- void s_reap_all_child (pcb_t *parent)

  Uses recursion to reap all children of specified parent.
- void s_exit (void)

  Unconditionally exit the calling process.
- int s_zombie (pid_t pid)

  Makes the specified processor a zombie, orpanifies all its child processors, then reaps all children.
- int s_nice (pid_t pid, int priority)

  Set the priority of the specified thread.
- int s_sleep (unsigned int ticks)

  Suspends execution of the calling proces for a specified number of clock ticks.
- int s_busy (void)

  Suspends execution of the calling process for an unspecified amount of time.
- int s_spawn_and_wait (void *(*func)(void *), char *argv[], int fd0, int fd1, bool nohang, unsigned int priority)

  Spawns and waits for a process.
- int s_fg (pcb_t *proc)

  If pid is specified, fg looks for processor with the pid. If no specified pid, fg looks in order of stopped and background, then the processor with the highest job id. Fg then gives the processor terminal control.
- int s_bg_wait (pcb_t *proc)

  Checks status of background processes with waitpid(nohang).
- pcb_t * s_find_process (pid_t pid)

  Finds a process in any state.
- pcb_t * find_jobs_proc (CircularList *list)

  Looks for the processor in stopped or background with the highest job id.
- int s_remove_process (pid_t pid)

  Removes a process in any state.
- void * s_function_from_string (char *program)

  Returns the function that was specified through string.

- int s_write_log (log_message_t logtype, pcb_t *proc, unsigned int old_nice)

  Prints information to log.txt for each movement of each processor/thread.
- int s_move_process (CircularList *destination, pid_t pid)

  Find the list that the processor belongs to, removes it from that list, and adds it to the specified list in argument.
- int s_print_process (CircularList *list)

  Prints all processes in processes, stopped, blocked, zombied. Used for 'ps' command.
- int s_print_jobs (void)

  Prints the list of processes in stopped or background list.
- int file_errno_helper (int ret)
- int s_open (const char *fname, int mode)

  open a file name fname with the mode mode and return a file descriptor. The allowed modes are as follows:
- ssize_t s_read (int fd, int n, char *buf)

  read n bytes from the file referenced by fd. On return, s_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error. A kernel level read should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.
- ssize_t s_write (int fd, const char *str, int n)

  write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, s_write returns the number of bytes written, or a negative value on error. Note that this writes bytes not chars, these can be anything, even '\0' A kernel level write should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.
- int s_close (int fd)

  close the file fd and return 0 on success, or a negative value on failure. A kernel level close should occur, and on success the local process' file descriptor table should be cleaned up appropriately.
- int s_unlink (const char *fname)

  removes the file specified by `fname` , `fname` must exist, returns -1 on error if it doesn't
- off_t s_lseek (int fd, int offset, int whence)

  reposition the file pointer for fd to the offset relative to whence. You must also implement the constants F_SEEK_↩ SET, F_SEEK_CUR, and F_SEEK_END, which reference similar file whences as their similarly named counterparts in lseek(2). A kernel level lseek should occur, and necessary changes to the calling process' file descriptor table will be necessary.
- int s_ls (const char *filename, int fd)

  List the file filename in the current directory. If filename is NULL, list all files in the current directory. Before EC implementations, this should be very simple and could literally be a call a similar k_function.
- char * s_read_all (const char *filename, int *read_num)

  Wrapper function around k_read_all.
- char * s_get_fname_from_fd (int fd)

  Wrapper function around k_get_fname_from_fd.
- int s_update_timestamp (const char *source)

  Wrapper function around k_update_timestamp. Returns the filename for the given file descriptor number.
- off_t s_does_file_exist2 (const char *fname)

  Wrapper function around k_does_file_exst2. Change the timestamp of the file to the current time.
- int s_rename (const char *source, const char *dest)

  s-function wrapper around k_rename, which renames `source` to `dest`. `source` must exist, if `dest` already exists, then it is deleted
- int s_change_mode (const char *change, const char *filename)

  s-function wrapper around k_change_mode, which changes the mode (permission) of the file in the directory entry, specificed by `filename` with change `change`. Errors if the resulting permission is invalid or if `filename` doesn't exist
- int s_cp_within_fat (char *source, char *dest)

  s-function wrapper around k_cp_within_fat. Copies contents of `source` to `dest`. `source` must exist, if `dest` doesn't exist then it is created.
- int s_cp_to_host (char *source, char *host_dest)

  s-function wrapper around k_cp_to_host
- int s_cp_from_host (char *host_source, char *dest)

  s-function wrapper around k_cp_from_host

10.49.1 Function Documentation

10.49.1.1 duplicate_argv()

```
char ** duplicate_argv (
            char * argv[] )
```

10.49.1.2 file_errno_helper()

```
int file_errno_helper (
            int ret )
```

10.49.1.3 find_jobs_proc()

```
pcb_t * find_jobs_proc (
            CircularList * list )
```

Looks for the processor in stopped or background with the highest job id.

Parameters

list	

Returns

pcb_t*

10.49.1.4 free_argv()

```
void free_argv (
            char * argv[] )
```

10.49.1.5 get_arg_size()

```
int get_arg_size (
            char * argv[] )
```

10.49.1.6 s_bg_wait()

```
int s_bg_wait (
            pcb_t * proc )
```

Checks status of background processes with waitpid(nohang).

Parameters

proc	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.7 s_busy()

```
int s_busy (
            void  )
```

Suspends execution of the calling process for an unspecified amount of time.

Parameters

void.	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.8 s_change_mode()

```
int s_change_mode (
            const char * change,
            const char * filename )
```

s-function wrapper around k_change_mode, which changes the mode (permission) of the file in the directory entry, specificed by `filename` with change `change`. Errors if the resulting permission is invalid or if `filename` doesn't exist

Parameters

change	the change to be made (e.g. -w, +w, -rw, etc)
filename	name of the specified file to be changed

Returns

-1 on error, 0 if change mode was successful

10.49.1.9 s_close()

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

close the file fd and return 0 on success, or a negative value on failure. A kernel level close should occur, and on success the local process' file descriptor table should be cleaned up appropriately.

Parameters

fd	file descriptor number to be closed

Returns

int (-1 on error)

10.49.1.10 s_cp_from_host()

```
int s_cp_from_host (
            char * host_source,
            char * dest )
```

s-function wrapper around k_cp_from_host

Parameters

host_source	source filename to copy from (on host device)
dest	destination filename to copy into (in PennOS)

Returns

-1 on error, 0 if cp was successful

10.49.1.11 s_cp_to_host()

```
int s_cp_to_host (
            char * source,
            char * host_dest )
```

s-function wrapper around k_cp_to_host

Parameters

source	source filename to copy from (in PennOS)
host_dest	destination filename to copy into (on host device)

Returns

-1 on error, 0 if cp was successful

10.49.1.12 s_cp_within_fat()

```
int s_cp_within_fat (
            char * source,
            char * dest )
```

s-function wrapper around k_cp_within_fat. Copies contents of `source` to `dest`. `source` must exist, if `dest` doesn't exist then it is created.

Parameters

source	source filename to copy from
dest	destination filename to copy into

Returns

-1 on error, 0 if cp was successful

10.49.1.13 s_does_file_exist2()

```
off_t s_does_file_exist2 (
            const char * fname )
```

Wrapper function around k_does_file_exst2. Change the timestamp of the file to the current time.

Parameters

source	Source file name.

Returns

1 on success. Negative number on failure.

10.49.1.14 s_exit()

```
void s_exit (
            void  )
```

Unconditionally exit the calling process.

This will set the process state to zombied, adjust its state within the scheduler structures, and kill all child proceseses. (not done).

10.49.1.15 s_fg()

```
int s_fg (
            pcb_t * proc )
```

If pid is specified, fg looks for processor with the pid. If no specified pid, fg looks in order of stopped and background, then the processor with the highest job id. Fg then gives the processor terminal control.

Parameters

proc	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.16 s_find_process()

```
pcb_t * s_find_process (
            pid_t pid )
```

Finds a process in any state.

Parameters

pid	

Returns

pcb_t*

10.49.1.17 s_function_from_string()

```
void * s_function_from_string (
            char * program )
```

Returns the function that was specified through string.

Parameters

program	

Returns

the function that was specified.

10.49.1.18 s_get_fname_from_fd()

```
char * s_get_fname_from_fd (
            int fd )
```

Wrapper function around k_get_fname_from_fd.

Parameters

fd	The file descriptor number.

Returns

The file name of the `fd` . NULL is `fd` is invalid.

10.49.1.19 s_kill()

```
int s_kill (
            pid_t pid,
            int signal )
```

Send a signal to a particular process.

Parameters

pid	Process ID of the target proces.
signal	Signal number to be sent.

Returns

0 on success, -1 on error.

10.49.1.20 s_ls()

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

List the file filename in the current directory. If filename is NULL, list all files in the current directory. Before EC implementations, this should be very simple and could literally be a call a similar k_function.

Parameters

filename	Lists information about the specified file

Returns

int (-1 on error)

10.49.1.21 s_lseek()

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

reposition the file pointer for fd to the offset relative to whence. You must also implement the constants F_SEEK_↩ SET, F_SEEK_CUR, and F_SEEK_END, which reference similar file whences as their similarly named counterparts in lseek(2). A kernel level lseek should occur, and necessary changes to the calling process' file descriptor table will be necessary.

Parameters

fd	The filedescriptor whose file pointer is to be moved
offset	The offset at which to move by (in bytes)
whence	Relative byte location to offset from

Returns

off_t (-1 on error)

10.49.1.22 s_move_process()

```
int s_move_process (
            CircularList * destination,
            pid_t pid )
```

Find the list that the processor belongs to, removes it from that list, and adds it to the specified list in argument.

Parameters

destination	
pid	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.23 s_nice()

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

Set the priority of the specified thread.

Parameters

pid	Process ID of the target thread.
priority	The new priorty value of the thread (0, 1, or 2)

Returns

0 on success, -1 on failure.

10.49.1.24 s_open()

```
int s_open (
```

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

open a file name fname with the mode mode and return a file descriptor. The allowed modes are as follows:

F_WRITE - writing and reading, truncates if the file exists, or creates it if it does not exist. Only one instance of a file can be opened in F_WRITE mode; error if PennOS attempts to open a file in F_WRITE mode more than once F_READ - open the file for reading only, return an error if the file does not exist F_APPEND - open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file.

s_open returns a file descriptor on success and a negative value on error. This open will initially be done at the kernel level, using a more intricate and already-implemented kernel level function. If the kernel level function succeeds and returns a fd, the user level function should also somehow keep track that such file descriptor is managed by the calling process. This can be done in multiple ways.

Parameters

fname	opens file with this name
mode	opens file in this mode

Returns

int (-1 on error), on success returns fd number of opened file

10.49.1.25 s_print_jobs()

```
int s_print_jobs ( )
```

Prints the list of processes in stopped or background list.

Parameters

return int Returns 0 on success, -1 on failure and sets errno.

10.49.1.26 s_print_process()

```
int s_print_process (
            CircularList * list )
```

Prints all processes in processes, stopped, blocked, zombied. Used for 'ps' command.

Parameters

list	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.27 s_read()

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

read n bytes from the file referenced by fd. On return, s_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error. A kernel level read should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.

Parameters

fd	file descriptor to read from (from fd's offset)
n	number of bytes to read
buf	buffer to read into

Returns

ssize_t (-1 on error), 0 if EOF reached, otherwise number of bytes read

10.49.1.28 s_read_all()

```
char * s_read_all (
            const char * filename,
            int * read_num )
```

Wrapper function around k_read_all.

Reads all contents from the file with the file name `filename`. Outputs the contents as well as update `read_num` to the number of bytes read.

Parameters

filename	Name of the file we want to read from.
read_num	Pointer to an integer variable that will store the number of bytes read.

Returns

All contents of `filename` in char* format.

10.49.1.29 s_reap_all_child()

```
void s_reap_all_child (
            pcb_t * parent )
```

Uses recursion to reap all children of specified parent.

Parameters

parent	PCB of the parent.

10.49.1.30 s_remove_process()

```
int s_remove_process (
            pid_t pid )
```

Removes a process in any state.

Parameters

pid	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.31 s_rename()

```
int s_rename (
            const char * source,
            const char * dest )
```

s-function wrapper around k_rename, which renames `source` to `dest`. `source` must exist, if `dest` already exists, then it is deleted

Parameters

source	name of the source file to be renamed
dest	new name of file

Returns

-1 on error, 0 if rename was successful

10.49.1.32 s_resume_block()

```
int s_resume_block (
            pid_t pid )
```

Blocks process with command "sleep" that has been stopped before.

Parameters

pid	Process ID of the child to wait for.

Returns

0 on success, -1 on error.

10.49.1.33 s_sleep()

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

Suspends execution of the calling proces for a specified number of clock ticks.

This function is analogous to `sleep(3)` in Linux, with the behavior that the system clock continues to tick even if the call is interrupted. The sleep can be interrupted by a P_SIGTERM signal, after which the function will return prematurely.

Parameters

ticks	Duration of the sleep in system clock ticks. Must be greater than 0.

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.34 s_spawn()

```
pid_t s_spawn (
            void *(*)(void *) func,
            char * argv[],
            int fd0,
            int fd1 )
```

Create a child process that executes the function `func`. The child will retain some attributes of the parent.

Parameters

func	Function to be executed by the child process.
argv	Null-terminated array of args, including the command name as argv[0].
fd0	Input file descriptor.
fd1	Output file descriptor.

Returns

pid_t The process ID of the created child process. // need to define error output?

10.49.1.35 s_spawn_and_wait()

```
int s_spawn_and_wait (
            void *(*)(void *) func,
```

```
            char * argv[],
            int fd0,
            int fd1,
            bool nohang,
            unsigned int priority )
```

Spawns and waits for a process.

Spawns and waits for a process, combining s_spawn and s_waitpid.

Parameters

func	
argv	
fd0	
fd1	
nohang	

Returns

int

10.49.1.36 s_spawn_nice()

```
pid_t s_spawn_nice (
            void *(*)(void *) func,
            char * argv[],
            int fd0,
            int fd1,
            unsigned int priority )
```

Create a child process that executes the function `func`, with a specified priority. This is an exact copy of s_spawn except that the priority of the created process can be specificed at creation.

Parameters

func	Function to be executed by the child process.
argv	Null-terminated array of args, including the command name as argv[0].
fd0	Input file descriptor.
fd1	Output file descriptor.

Returns

pid_t The process ID of the created child process.

10.49.1.37 s_unlink()

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

removes the file specified by `fname` , `fname` must exist, returns -1 on error if it doesn't

Parameters

fname	name of the file to be removed

Returns

int (-1 on error)

10.49.1.38 s_update_timestamp()

```
int s_update_timestamp (
            const char * source )
```

Wrapper function around k_update_timestamp. Returns the filename for the given file descriptor number.

Parameters

source	Soruce file name

Returns

int (-1 on error)

10.49.1.39 s_waitpid()

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

Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.

Parameters

pid	Process ID of the child to wait for.
wstatus	Pointer to an integer variable where the status will be stored.
nohang	If true, return immediately if no child has exited.

Returns

pid_t The process ID of the child which has changed state on success, -1 on error.

10.49.1.40 s_write()

```
ssize_t s_write (
            int fd,
```

```
            const char * str,
            int n )
```

write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, s_write returns the number of bytes written, or a negative value on error. Note that this writes bytes not chars, these can be anything, even '\0' A kernel level write should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.

Parameters

fd	file descriptor to write to (from fd's offset)
str	input string to be written
n	number of bytes to be written

Returns

ssize_t (-1 on error), otherwise number of bytes written

10.49.1.41 s_write_log()

```
int s_write_log (
            log_message_t logtype,
            pcb_t * proc,
            unsigned int old_nice )
```

Prints information to log.txt for each movement of each processor/thread.

Parameters

logtype	
proc	
old_nice	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.49.1.42 s_zombie()

```
int s_zombie (
            pid_t pid )
```

Makes the specified processor a zombie, orpanifies all its child processors, then reaps all children.

Parameters

pid	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50 src/util/sys_call.h File Reference

```
#include <stdbool.h>
#include <string.h>
#include "error.h"
#include "globals.h"
#include "kernel.h"
#include "pennfat_kernel.h"
#include "shellbuiltins.h"
```

Macros

- #define STATUS_EXITED 0x00
- #define STATUS_STOPPED 0x01
- #define STATUS_SIGNALED 0x02
- #define P_WIFEXITED(status) (((status) & 0xFF) == STATUS_EXITED)
- #define P_WIFSTOPPED(status) (((status) & 0xFF) == STATUS_STOPPED)
- #define P_WIFSIGNALED(status) (((status) & 0xFF) == STATUS_SIGNALED)

Enumerations

- enum log_message_t {
  SCHEDULE , CREATE , EXIT , SIGNAL ,
  ZOMBIE , ORPHAN , WAIT , NICE ,
  BLOCK , UNBLOCK , STOP , CONTINUE }

  This is an enum used to specify which log message should be added for s_write_log.

Functions

- pid_t s_spawn (void ∗(∗func)(void ∗), char ∗argv[ ], int fd0, int fd1)

  Create a child process that executes the function `func`. The child will retain some attributes of the parent.
- pid_t s_spawn_nice (void ∗(∗func)(void ∗), char ∗argv[ ], int fd0, int fd1, unsigned int priority)

  Create a child process that executes the function `func`, with a specified priority. This is an exact copy of s_spawn except that the priority of the created process can be specificed at creation.
- pid_t s_waitpid (pid_t pid, int ∗wstatus, bool nohang)

  Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.
- int s_resume_block (pid_t pid)

  Blocks process with command "sleep" that has been stopped before.
- int s_kill (pid_t pid, int signal)

  Send a signal to a particular process.
- void s_reap_all_child (pcb_t ∗parent)

  Uses recursion to reap all children of specified parent.
- void s_exit (void)

  Unconditionally exit the calling process.

- int s_nice (pid_t pid, int priority)

  Set the priority of the specified thread.
- int s_sleep (unsigned int ticks)

  Suspends execution of the calling proces for a specified number of clock ticks.
- int s_busy (void)

  Suspends execution of the calling process for an unspecified amount of time.
- int s_spawn_and_wait (void ∗(∗func)(void ∗), char ∗argv[ ], int fd0, int fd1, bool nohang, unsigned int priority)

  Spawns and waits for a process, combining s_spawn and s_waitpid.
- pcb_t ∗ s_find_process (pid_t pid)

  Finds a process in any state.
- int s_remove_process (pid_t pid)

  Removes a process in any state.
- void ∗ s_function_from_string (char ∗program)

  Returns the function that was specified through string.
- int s_write_log (log_message_t logtype, pcb_t ∗proc, unsigned int old_nice)

  Prints information to log.txt for each movement of each processor/thread.
- int s_move_process (CircularList ∗destination, pid_t pid)

  Find the list that the processor belongs to, removes it from that list, and adds it to the specified list in argument.
- int s_zombie (pid_t pid)

  Makes the specified processor a zombie, orpanifies all its child processors, then reaps all children.
- int s_fg (pcb_t ∗proc)

  If pid is specified, fg looks for processor with the pid. If no specified pid, fg looks in order of stopped and background, then the processor with the highest job id. Fg then gives the processor terminal control.
- int s_bg_wait (pcb_t ∗proc)

  Checks status of background processes with waitpid(nohang).
- pcb_t ∗ find_jobs_proc (CircularList ∗list)

  Looks for the processor in stopped or background with the highest job id.
- int s_print_process (CircularList ∗list)

  Prints all processes in processes, stopped, blocked, zombied. Used for 'ps' command.
- int s_print_jobs ()

  Prints the list of processes in stopped or background list.
- int file_errno_helper (int ret)
- int s_open (const char ∗fname, int mode)

  open a file name fname with the mode mode and return a file descriptor. The allowed modes are as follows:
- ssize_t s_read (int fd, int n, char ∗buf)

  read n bytes from the file referenced by fd. On return, s_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error. A kernel level read should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.
- ssize_t s_write (int fd, const char ∗str, int n)

  write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, s_write returns the number of bytes written, or a negative value on error. Note that this writes bytes not chars, these can be anything, even '\0' A kernel level write should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.
- int s_close (int fd)

  close the file fd and return 0 on success, or a negative value on failure. A kernel level close should occur, and on success the local process' file descriptor table should be cleaned up appropriately.
- int s_unlink (const char ∗fname)

  removes the file specified by `fname` , `fname` must exist, returns -1 on error if it doesn't
- off_t s_lseek (int fd, int offset, int whence)

  reposition the file pointer for fd to the offset relative to whence. You must also implement the constants F_SEEK_↩ SET, F_SEEK_CUR, and F_SEEK_END, which reference similar file whences as their similarly named counterparts in lseek(2). A kernel level lseek should occur, and necessary changes to the calling process' file descriptor table will be necessary.

- int s_ls (const char *filename, int fd)

  List the file filename in the current directory. If filename is NULL, list all files in the current directory. Before EC implementations, this should be very simple and could literally be a call a similar k_function.
- char * s_read_all (const char *filename, int *read_num)

  Wrapper function around k_read_all.
- char * s_get_fname_from_fd (int fd)

  Wrapper function around k_get_fname_from_fd.
- int s_update_timestamp (const char *source)

  Wrapper function around k_update_timestamp. Returns the filename for the given file descriptor number.
- off_t s_does_file_exist2 (const char *fname)

  Wrapper function around k_does_file_exst2. Change the timestamp of the file to the current time.
- int s_rename (const char *source, const char *dest)

  s-function wrapper around k_rename, which renames `source` to `dest`. `source` must exist, if `dest` already exists, then it is deleted
- int s_change_mode (const char *change, const char *filename)

  s-function wrapper around k_change_mode, which changes the mode (permission) of the file in the directory entry, specificed by `filename` with change `change`. Errors if the resulting permission is invalid or if `filename` doesn't exist
- int s_cp_within_fat (char *source, char *dest)

  s-function wrapper around k_cp_within_fat. Copies contents of `source` to `dest`. `source` must exist, if `dest` doesn't exist then it is created.
- int s_cp_to_host (char *source, char *host_dest)

  s-function wrapper around k_cp_to_host
- int s_cp_from_host (char *host_source, char *dest)

  s-function wrapper around k_cp_from_host

10.50.1 Macro Definition Documentation

10.50.1.1 P_WIFEXITED

```
#define P_WIFEXITED(
              status ) (((status) & 0xFF) == STATUS_EXITED)
```

10.50.1.2 P_WIFSIGNALED

```
#define P_WIFSIGNALED(
              status ) (((status) & 0xFF) == STATUS_SIGNALED)
```

10.50.1.3 P_WIFSTOPPED

```
#define P_WIFSTOPPED(
              status ) (((status) & 0xFF) == STATUS_STOPPED)
```

10.50.1.4 STATUS_EXITED

```
#define STATUS_EXITED 0x00
```

10.50.1.5 STATUS_SIGNALED

```
#define STATUS_SIGNALED 0x02
```

10.50.1.6 STATUS_STOPPED

```
#define STATUS_STOPPED 0x01
```

10.50.2 Enumeration Type Documentation

10.50.2.1 log_message_t

```
enum log_message_t
```

This is an enum used to specify which log message should be added for s_write_log.

Enumerator

SCHEDULE	
CREATE	
EXIT	
SIGNAL	
ZOMBIE	
ORPHAN	
WAIT	
NICE	
BLOCK	
UNBLOCK	
STOP	
CONTINUE	

10.50.3 Function Documentation

10.50.3.1 file_errno_helper()

```
int file_errno_helper (
            int ret )
```

10.50.3.2 find_jobs_proc()

```
pcb_t * find_jobs_proc (
            CircularList * list )
```

Looks for the processor in stopped or background with the highest job id.

Parameters

list	

Returns

pcb_t*

10.50.3.3 s_bg_wait()

```
int s_bg_wait (
            pcb_t * proc )
```

Checks status of background processes with waitpid(nohang).

Parameters

proc	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.4 s_busy()

```
int s_busy (
            void  )
```

Suspends execution of the calling process for an unspecified amount of time.

Parameters

void.	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.5 s_change_mode()

```
int s_change_mode (
            const char * change,
            const char * filename )
```

s-function wrapper around k_change_mode, which changes the mode (permission) of the file in the directory entry, specificed by `filename` with change `change`. Errors if the resulting permission is invalid or if `filename` doesn't exist

Parameters

change	the change to be made (e.g. -w, +w, -rw, etc)
filename	name of the specified file to be changed

Returns

-1 on error, 0 if change mode was successful

10.50.3.6 s_close()

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

close the file fd and return 0 on success, or a negative value on failure. A kernel level close should occur, and on success the local process' file descriptor table should be cleaned up appropriately.

Parameters

fd	file descriptor number to be closed

Returns

int (-1 on error)

10.50.3.7 s_cp_from_host()

```
int s_cp_from_host (
            char * host_source,
            char * dest )
```

s-function wrapper around k_cp_from_host

Parameters

host_source	source filename to copy from (on host device)
dest	destination filename to copy into (in PennOS)

Returns

-1 on error, 0 if cp was successful

10.50.3.8 s_cp_to_host()

```
int s_cp_to_host (
            char * source,
            char * host_dest )
```

s-function wrapper around k_cp_to_host

Parameters

source	source filename to copy from (in PennOS)
host_dest	destination filename to copy into (on host device)

Returns

-1 on error, 0 if cp was successful

10.50.3.9 s_cp_within_fat()

```
int s_cp_within_fat (
            char * source,
            char * dest )
```

s-function wrapper around k_cp_within_fat. Copies contents of `source` to `dest`. `source` must exist, if `dest` doesn't exist then it is created.

Parameters

source	source filename to copy from
dest	destination filename to copy into

Returns

-1 on error, 0 if cp was successful

10.50.3.10 s_does_file_exist2()

```
off_t s_does_file_exist2 (
            const char * fname )
```

Wrapper function around k_does_file_exst2. Change the timestamp of the file to the current time.

Parameters

source	Source file name.

Returns

1 on success. Negative number on failure.

10.50.3.11 s_exit()

```
void s_exit (
            void  )
```

Unconditionally exit the calling process.

This will set the process state to zombied, adjust its state within the scheduler structures, and kill all child proceseses. (not done).

10.50.3.12 s_fg()

```
int s_fg (
            pcb_t * proc )
```

If pid is specified, fg looks for processor with the pid. If no specified pid, fg looks in order of stopped and background, then the processor with the highest job id. Fg then gives the processor terminal control.

Parameters

proc	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.13 s_find_process()

```
pcb_t * s_find_process (
            pid_t pid )
```

Finds a process in any state.

Parameters

pid	

Returns

pcb_t*

10.50.3.14 s_function_from_string()

```
void * s_function_from_string (
            char * program )
```

Returns the function that was specified through string.

Parameters

program	

Returns

the function that was specified.

10.50.3.15 s_get_fname_from_fd()

```
char * s_get_fname_from_fd (
            int fd )
```

Wrapper function around k_get_fname_from_fd.

Parameters

fd	The file descriptor number.

Returns

The file name of the `fd` . NULL is `fd` is invalid.

10.50.3.16 s_kill()

```
int s_kill (
            pid_t pid,
            int signal )
```

Send a signal to a particular process.

Parameters

pid	Process ID of the target proces.
signal	Signal number to be sent.

Returns

0 on success, -1 on error.

10.50.3.17 s_ls()

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

List the file filename in the current directory. If filename is NULL, list all files in the current directory. Before EC implementations, this should be very simple and could literally be a call a similar k_function.

Parameters

filename	Lists information about the specified file

Returns

int (-1 on error)

10.50.3.18 s_lseek()

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

reposition the file pointer for fd to the offset relative to whence. You must also implement the constants F_SEEK_↩ SET, F_SEEK_CUR, and F_SEEK_END, which reference similar file whences as their similarly named counterparts in lseek(2). A kernel level lseek should occur, and necessary changes to the calling process' file descriptor table will be necessary.

Parameters

fd	The filedescriptor whose file pointer is to be moved
offset	The offset at which to move by (in bytes)
whence	Relative byte location to offset from

Returns

off_t (-1 on error)

10.50.3.19 s_move_process()

```
int s_move_process (
            CircularList * destination,
            pid_t pid )
```

Find the list that the processor belongs to, removes it from that list, and adds it to the specified list in argument.

Parameters

destination	
pid	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.20 s_nice()

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

Set the priority of the specified thread.

Parameters

pid	Process ID of the target thread.
priority	The new priorty value of the thread (0, 1, or 2)

Returns

0 on success, -1 on failure.

10.50.3.21 s_open()

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

open a file name fname with the mode mode and return a file descriptor. The allowed modes are as follows:

F_WRITE - writing and reading, truncates if the file exists, or creates it if it does not exist. Only one instance of a file can be opened in F_WRITE mode; error if PennOS attempts to open a file in F_WRITE mode more than once F_READ - open the file for reading only, return an error if the file does not exist F_APPEND - open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file.

s_open returns a file descriptor on success and a negative value on error. This open will initially be done at the kernel level, using a more intricate and already-implemented kernel level function. If the kernel level function succeeds and returns a fd, the user level function should also somehow keep track that such file descriptor is managed by the calling process. This can be done in multiple ways.

Parameters

fname	opens file with this name
mode	opens file in this mode

Returns

int (-1 on error), on success returns fd number of opened file

10.50.3.22 s_print_jobs()

```
int s_print_jobs ( )
```

Prints the list of processes in stopped or background list.

Parameters

return int Returns 0 on success, -1 on failure and sets errno.

10.50.3.23 s_print_process()

```
int s_print_process (
            CircularList * list )
```

Prints all processes in processes, stopped, blocked, zombied. Used for 'ps' command.

Parameters

list	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.24 s_read()

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

read n bytes from the file referenced by fd. On return, s_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error. A kernel level read should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.

Parameters

fd	file descriptor to read from (from fd's offset)
n	number of bytes to read
buf	buffer to read into

Returns

ssize_t (-1 on error), 0 if EOF reached, otherwise number of bytes read

10.50.3.25 s_read_all()

```
char * s_read_all (
            const char * filename,
            int * read_num )
```

Wrapper function around k_read_all.

Reads all contents from the file with the file name `filename`. Outputs the contents as well as update `read_num` to the number of bytes read.

Parameters

filename	Name of the file we want to read from.
read_num	Pointer to an integer variable that will store the number of bytes read.

Returns

All contents of `filename` in char* format.

10.50.3.26 s_reap_all_child()

```
void s_reap_all_child (
            pcb_t * parent )
```

Uses recursion to reap all children of specified parent.

Parameters

parent	PCB of the parent.

10.50.3.27 s_remove_process()

```
int s_remove_process (
            pid_t pid )
```

Removes a process in any state.

Parameters

pid	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.28 s_rename()

```
int s_rename (
            const char * source,
            const char * dest )
```

s-function wrapper around k_rename, which renames `source` to `dest`. `source` must exist, if `dest` already exists, then it is deleted

Parameters

source	name of the source file to be renamed
dest	new name of file

Returns

-1 on error, 0 if rename was successful

10.50.3.29 s_resume_block()

```
int s_resume_block (
            pid_t pid )
```

Blocks process with command "sleep" that has been stopped before.

Parameters

pid	Process ID of the child to wait for.

Returns

0 on success, -1 on error.

10.50.3.30 s_sleep()

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

Suspends execution of the calling proces for a specified number of clock ticks.

This function is analogous to `sleep(3)` in Linux, with the behavior that the system clock continues to tick even if the call is interrupted. The sleep can be interrupted by a P_SIGTERM signal, after which the function will return prematurely.

Parameters

ticks	Duration of the sleep in system clock ticks. Must be greater than 0.

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.31 s_spawn()

```
pid_t s_spawn (
            void *(*)(void *) func,
            char * argv[],
            int fd0,
            int fd1 )
```

Create a child process that executes the function `func`. The child will retain some attributes of the parent.

Parameters

func	Function to be executed by the child process.
argv	Null-terminated array of args, including the command name as argv[0].
fd0	Input file descriptor.
fd1	Output file descriptor.

Returns

pid_t The process ID of the created child process. // need to define error output?

10.50.3.32 s_spawn_and_wait()

```
int s_spawn_and_wait (
              void *(*)(void *) func,
              char * argv[],
              int fd0,
              int fd1,
              bool nohang,
              unsigned int priority )
```

Spawns and waits for a process, combining s_spawn and s_waitpid.

Spawns and waits for a process.

This generalizes the control loop of spawning a process, waiting on it via s_waitpid (depending on whether it was a background process) and then cleaning it up, into one function call. Used to call most shell functions (b_functions).

Parameters

func	The function to be executed.
argv	The argument to that function that is passed into the s_spawn call.
fd0	Input file descriptor.
fd1	Output file descriptor.
nohang	Whether or not to wait on the process or immediately proceed.

Returns

int Returns 0 on success, -1 on failure and sets errno.

Parameters

func	
argv	
fd0	
fd1	
nohang	

Returns

int

Spawns and waits for a process, combining s_spawn and s_waitpid.

Parameters

func	
argv	

Parameters

fd0	
fd1	
nohang	

Returns

int

10.50.3.33 s_spawn_nice()

```
pid_t s_spawn_nice (
            void *(*)(void *) func,
            char * argv[],
            int fd0,
            int fd1,
            unsigned int priority )
```

Create a child process that executes the function `func`, with a specified priority. This is an exact copy of s_spawn except that the priority of the created process can be specificed at creation.

Parameters

func	Function to be executed by the child process.
argv	Null-terminated array of args, including the command name as argv[0].
fd0	Input file descriptor.
fd1	Output file descriptor.

Returns

pid_t The process ID of the created child process.

10.50.3.34 s_unlink()

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

removes the file specified by `fname` , `fname` must exist, returns -1 on error if it doesn't

Parameters

fname	name of the file to be removed

Returns

int (-1 on error)

10.50.3.35 s_update_timestamp()

```
int s_update_timestamp (
            const char * source )
```

Wrapper function around k_update_timestamp. Returns the filename for the given file descriptor number.

Parameters

source	Soruce file name

Returns

int (-1 on error)

10.50.3.36 s_waitpid()

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

Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.

Parameters

pid	Process ID of the child to wait for.
wstatus	Pointer to an integer variable where the status will be stored.
nohang	If true, return immediately if no child has exited.

Returns

pid_t The process ID of the child which has changed state on success, -1 on error.

10.50.3.37 s_write()

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

write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, s_write returns the number of bytes written, or a negative value on error. Note that this writes bytes not chars, these can be anything, even '\0' A kernel level write should occur to perform the actual functionality, but its important to remember that the process' file descriptor may need to be updated as the position of the file pointer changes.

Parameters

fd	file descriptor to write to (from fd's offset)
str	input string to be written
n	number of bytes to be written

Returns

ssize_t (-1 on error), otherwise number of bytes written

10.50.3.38 s_write_log()

```
int s_write_log (
            log_message_t logtype,
            pcb_t * proc,
            unsigned int old_nice )
```

Prints information to log.txt for each movement of each processor/thread.

Parameters

logtype	
proc	
old_nice	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.50.3.39 s_zombie()

```
int s_zombie (
            pid_t pid )
```

Makes the specified processor a zombie, orpanifies all its child processors, then reaps all children.

Parameters

pid	

Returns

int Returns 0 on success, -1 on failure and sets errno.

10.51 sys_call.h

Go to the documentation of this file.

```
00001 #ifndef SYSCALL_H
00002 #define SYSCALL_H
00003
00004 #include <stdbool.h>
00005 #include <string.h>
00006 #include "error.h"
00007 #include "globals.h"
00008 #include "kernel.h"
00009 #include "pennfat_kernel.h"
00010 #include "shellbuiltins.h"
00011
00012 #define STATUS_EXITED 0x00
00013 #define STATUS_STOPPED 0x01
00014 #define STATUS_SIGNALED 0x02
00015
00016 #define P_WIFEXITED(status) (((status) & 0xFF) == STATUS_EXITED)
00017 #define P_WIFSTOPPED(status) (((status) & 0xFF) == STATUS_STOPPED)
00018 #define P_WIFSIGNALED(status) (((status) & 0xFF) == STATUS_SIGNALED)
00019
00026 typedef enum {
00027   SCHEDULE,
00028   CREATE,
00029   EXIT,
00030   SIGNAL,
00031   ZOMBIE,
00032   ORPHAN,
00033   WAIT,
00034   NICE,
00035   BLOCK,
00036   UNBLOCK,
00037   STOP,
00038   CONTINUE
00039 } log_message_t;
00040
00041 /*== system call functions for interacting with PennOS process creation==*/
00055 pid_t s_spawn(void* (*func)(void*), char* argv[], int fd0, int fd1);
00056
00069 pid_t s_spawn_nice(void* (*func)(void*),
00070                    char* argv[],
00071                    int fd0,
00072                    int fd1,
00073                    unsigned int priority);
00074
00087 pid_t s_waitpid(pid_t pid, int* wstatus, bool nohang);
00088
00095 int s_resume_block(pid_t pid);
00096
00104 int s_kill(pid_t pid, int signal);
00105
00111 void s_reap_all_child(pcb_t* parent);
00112
00119 void s_exit(void);
00120
00121 /*============================ system calls for interacting with the scheduler
00122  * ==================================*/
00123
00131 int s_nice(pid_t pid, int priority);
00132
00147 int s_sleep(unsigned int ticks);
00148
00157 int s_busy(void);
00158
00159 /***** CUSTOM SYSCALLS FOR SCHEDULER*/
00160
00177 int s_spawn_and_wait(void* (*func)(void*),
00178                      char* argv[],
00179                      int fd0,
00180                      int fd1,
00181                      bool nohang,
00182                      unsigned int priority);
00183
00190 pcb_t* s_find_process(pid_t pid);
00191
00198 int s_remove_process(pid_t pid);
00199
00206 void* s_function_from_string(char* program);
00207
00217 int s_write_log(log_message_t logtype, pcb_t* proc, unsigned int old_nice);
00218
00227 int s_move_process(CircularList* destination, pid_t pid);
00228
00236 int s_zombie(pid_t pid);
00237 /***** CUSTOM SYSCALLS FOR SCHEDULER*/
00238
00249 int s_spawn_and_wait(void* (*func)(void*),
00250                      char* argv[],
00251                      int fd0,
```

```
00252                       int fd1,
00253                       bool nohang,
00254                       unsigned int priority);
00255
00265 int s_fg(pcb_t* proc);
00266
00273 int s_bg_wait(pcb_t* proc);
00274
00281 pcb_t* s_find_process(pid_t pid);
00282
00290 pcb_t* find_jobs_proc(CircularList* list);
00291
00298 int s_remove_process(pid_t pid);
00299
00300 void* s_function_from_string(char* program);
00301
00302 int s_write_log(log_message_t logtype, pcb_t* proc, unsigned int old_nice);
00303
00304 int s_move_process(CircularList* destination, pid_t pid);
00305
00313 int s_print_process(CircularList* list);
00314
00321 int s_print_jobs();
00322
00323 int file_errno_helper(int ret);
00324
00325 /*== system call functions for interacting with PennOS filesystem ==*/
00350 int s_open(const char* fname, int mode);
00351
00365 ssize_t s_read(int fd, int n, char* buf);
00366
00382 ssize_t s_write(int fd, const char* str, int n);
00383
00393 int s_close(int fd);
00394
00403 int s_unlink(const char* fname);
00404
00418 off_t s_lseek(int fd, int offset, int whence);
00419
00429 int s_ls(const char* filename, int fd);
00430
00445 char* s_read_all(const char* filename, int* read_num);
00446
00454 char* s_get_fname_from_fd(int fd);
00455
00464 int s_update_timestamp(const char* source);
00465
00474 off_t s_does_file_exist2(const char* fname);
00475
00486 int s_rename(const char* source, const char* dest);
00487
00499 int s_change_mode(const char* change, const char* filename);
00500
00511 int s_cp_within_fat(char* source, char* dest);
00512
00521 int s_cp_to_host(char* source, char* host_dest);
00522
00531 int s_cp_from_host(char* host_source, char* dest);
00532
00533 #endif
```

10.52 test/sched-demo.c File Reference

```
#include <pthread.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include "../src/util/spthread.h"
```

Macros

- #define _POSIX_C_SOURCE 200809L

- #define _DEFAULT_SOURCE 1
- #define _XOPEN_SOURCE 700
- #define NUM_THREADS 4
- #define BUF_SIZE 4096

Functions

- void cancel_and_join (spthread_t thread)
- int main (void)

10.52.1 Macro Definition Documentation

10.52.1.1 _DEFAULT_SOURCE

```
#define _DEFAULT_SOURCE 1
```

10.52.1.2 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

10.52.1.3 _XOPEN_SOURCE

```
#define _XOPEN_SOURCE 700
```

10.52.1.4 BUF_SIZE

```
#define BUF_SIZE 4096
```

10.52.1.5 NUM_THREADS

```
#define NUM_THREADS 4
```

10.52.2 Function Documentation

10.52.2.1 cancel_and_join()

```
void cancel_and_join (
            spthread_t thread )
```

10.52.2.2 main()

```
int main (
            void  )
```

Index