PennOS

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 dir_entry_t Struct Reference

Directory entry structure for files in the filesystem.

```
#include <fat_routines.h>
```

Public Attributes

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

3.1.1 Detailed Description

Directory entry structure for files in the filesystem.

Definition at line 46 of file fat_routines.h.

3.1.2 Member Data Documentation

3.1.2.1 firstBlock

```
uint16_t dir_entry_t::firstBlock
```

Definition at line 49 of file fat_routines.h.

3.1.2.2 mtime

```
time_t dir_entry_t::mtime
```

Definition at line 52 of file fat_routines.h.

3.1.2.3 name

```
char dir_entry_t::name[32]
```

Definition at line 47 of file fat_routines.h.

3.1.2.4 perm

```
uint8_t dir_entry_t::perm
```

Definition at line 51 of file fat_routines.h.

3.1.2.5 reserved

```
char dir_entry_t::reserved[16]
```

Definition at line 53 of file fat_routines.h.

3.1.2.6 size

```
uint32_t dir_entry_t::size
```

Definition at line 48 of file fat_routines.h.

3.1.2.7 type

```
uint8_t dir_entry_t::type
```

Definition at line 50 of file fat_routines.h.

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

- SRC/fs/fat_routines.h

3.2 fd_entry_t Struct Reference

File descriptor entry structure for open files.

```
#include <fat_routines.h>
```

Public Attributes

- int in_use
- int ref_count
- char filename [32]
- uint32_t size
- uint16_t first_block
- uint32_t position
- uint8_t mode

3.2.1 Detailed Description

File descriptor entry structure for open files.

Definition at line 59 of file fat_routines.h.

3.2.2 Member Data Documentation

3.2.2.1 filename

```
char fd_entry_t::filename[32]
```

Definition at line 62 of file fat_routines.h.

3.2.2.2 first_block

```
uint16_t fd_entry_t::first_block
```

Definition at line 64 of file fat_routines.h.

3.2.2.3 in_use

```
int fd_entry_t::in_use
```

Definition at line 60 of file fat_routines.h.

3.2.2.4 mode

`uint8_t fd_entry_t::mode`

Definition at line 66 of file fat_routines.h.

3.2.2.5 position

`uint32_t fd_entry_t::position`

Definition at line 65 of file fat_routines.h.

3.2.2.6 ref_count

`int fd_entry_t::ref_count`

Definition at line 61 of file fat_routines.h.

3.2.2.7 size

`uint32_t fd_entry_t::size`

Definition at line 63 of file fat_routines.h.

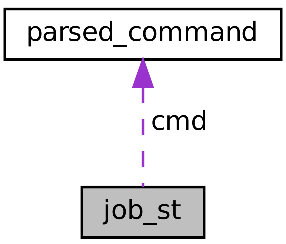
The documentation for this struct was generated from the following file:

- SRC/fs/fat_routines.h

3.3 job_st Struct Reference

`#include <Job.h>`

Collaboration diagram for job_st:



Public Attributes

- jid_t id
- struct parsed_command * cmd
- pid_t * pids
- job_state_t state
- size_t num_pids
- pid_t pgid
- size_t finished_count

3.3.1 Detailed Description

Definition at line 16 of file Job.h.

3.3.2 Member Data Documentation

3.3.2.1 cmd

```
struct parsed_command* job_st::cmd
```

Definition at line 18 of file Job.h.

3.3.2.2 finished_count

```
size_t job_st::finished_count
```

Definition at line 23 of file Job.h.

3.3.2.3 id

```
jid_t job_st::id
```

Definition at line 17 of file Job.h.

3.3.2.4 num_pids

```
size_t job_st::num_pids
```

Definition at line 21 of file Job.h.

3.3.2.5 pgid

```
pid_t job_st::pgid
```

Definition at line 22 of file Job.h.

3.3.2.6 pids

```
pid_t* job_st::pids
```

Definition at line 19 of file Job.h.

3.3.2.7 state

```
job_state_t job_st::state
```

Definition at line 20 of file Job.h.

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

- SRC/shell/Job.h

3.4 parsed_command Struct Reference

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

Public Attributes

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

3.4.1 Detailed Description

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

Definition at line 36 of file parser.h.

3.4.2 Member Data Documentation

3.4.2.1 commands

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

Definition at line 56 of file parser.h.

3.4.2.2 is_background

```
bool parsed_command::is_background
```

Definition at line 39 of file parser.h.

3.4.2.3 is_file_append

```
bool parsed_command::is_file_append
```

Definition at line 43 of file parser.h.

3.4.2.4 num_commands

```
size_t parsed_command::num_commands
```

Definition at line 52 of file parser.h.

3.4.2.5 stdin_file

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

Definition at line 46 of file parser.h.

3.4.2.6 stdout_file

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

Definition at line 49 of file parser.h.

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

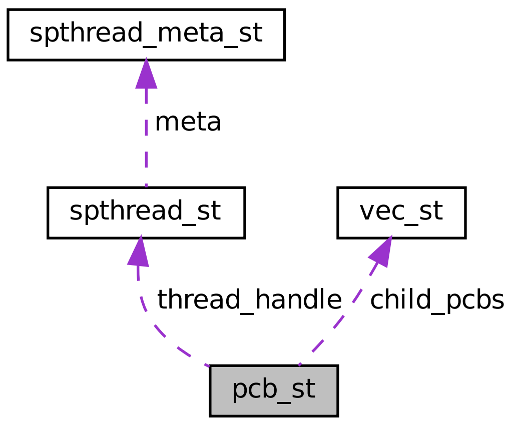
- SRC/shell/parser.h

3.5 pcb_st Struct Reference

The PCB structure, which contains all the information about a process. Notably, it contains the thread handle, pid, parent pid, child pcbs, priority level, process state, command string, signals to be sent, input and output file descriptors, process status, sleeping status, and time to wake.

```
#include <kern_pcb.h>
```

Collaboration diagram for pcb_st:



Public Attributes

- spthread_t thread_handle
- pid_t pid
- pid_t par_pid
- Vec child_pcbs
- int priority
- char process_state
- char * cmd_str
- bool signals [3]
- int input_fd
- int output_fd
- int process_status
- bool is_sleeping
- int time_to_wake
- int fd_table [FILE_DESCRIPTOR_TABLE_SIZE]

3.5.1 Detailed Description

The PCB structure, which contains all the information about a process. Notably, it contains the thread handle, pid, parent pid, child pcbs, priority level, process state, command string, signals to be sent, input and output file descriptors, process status, sleeping status, and time to wake.

Definition at line 30 of file kern_pcb.h.

3.5.2 Member Data Documentation

3.5.2.1 child_pcbs

```
Vec pcb_st::child_pcbs
```

Definition at line 36 of file kern_pcb.h.

3.5.2.2 cmd_str

```
char* pcb_st::cmd_str
```

Definition at line 42 of file kern_pcb.h.

3.5.2.3 fd_table

```
int pcb_st::fd_table[FILE_DESCRIPTOR_TABLE_SIZE]
```

Definition at line 61 of file kern_pcb.h.

3.5.2.4 input_fd

```
int pcb_st::input_fd
```

Definition at line 48 of file kern_pcb.h.

3.5.2.5 is_sleeping

`bool pcb_st::is_sleeping`

Definition at line 58 of file kern_pcb.h.

3.5.2.6 output_fd

`int pcb_st::output_fd`

Definition at line 49 of file kern_pcb.h.

3.5.2.7 par_pid

`pid_t pcb_st::par_pid`

Definition at line 34 of file kern_pcb.h.

3.5.2.8 pid

`pid_t pcb_st::pid`

Definition at line 33 of file kern_pcb.h.

3.5.2.9 priority

`int pcb_st::priority`

Definition at line 38 of file kern_pcb.h.

3.5.2.10 process_state

`char pcb_st::process_state`

Definition at line 39 of file kern_pcb.h.

3.5.2.11 process_status

`int pcb_st::process_status`

Definition at line 51 of file kern_pcb.h.

3.5.2.12 signals

`bool pcb_st::signals[3]`

Definition at line 44 of file kern_pcb.h.

3.5.2.13 thread_handle

`spthread_t pcb_st::thread_handle`

Definition at line 31 of file kern_pcb.h.

3.5.2.14 time_to_wake

`int pcb_st::time_to_wake`

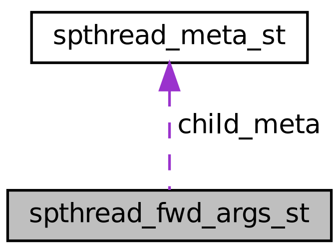
Definition at line 59 of file kern_pcb.h.

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

- SRC/kernel/kern_pcb.h

3.6 spthread_fwd_args_st Struct Reference

Collaboration diagram for spthread_fwd_args_st:



Public Attributes

- 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

3.6.1 Detailed Description

Definition at line 22 of file spthread.c.

3.6.2 Member Data Documentation

3.6.2.1 actual_arg

`void* spthread_fwd_args_st::actual_arg`

Definition at line 27 of file spthread.c.

3.6.2.2 actual_routine

`pthread_fn spthread_fwd_args_st::actual_routine`

Definition at line 26 of file spthread.c.

3.6.2.3 child_meta

`spthread_meta_t* spthread_fwd_args_st::child_meta`

Definition at line 40 of file spthread.c.

3.6.2.4 setup_cond

`pthread_cond_t spthread_fwd_args_st::setup_cond`

Definition at line 37 of file spthread.c.

3.6.2.5 setup_done

```
bool spthread_fwd_args_st::setup_done
```

Definition at line 35 of file spthread.c.

3.6.2.6 setup_mutex

```
pthread_mutex_t spthread_fwd_args_st::setup_mutex
```

Definition at line 36 of file spthread.c.

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

- SRC/lib/spthread.c

3.7 spthread_meta_st Struct Reference

Public Attributes

- sigset_t suspend_set
- volatile sig_atomic_t state
- pthread_mutex_t meta_mutex

3.7.1 Detailed Description

Definition at line 57 of file spthread.c.

3.7.2 Member Data Documentation

3.7.2.1 meta_mutex

```
pthread_mutex_t spthread_meta_st::meta_mutex
```

Definition at line 71 of file spthread.c.

3.7.2.2 state

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

Definition at line 68 of file spthread.c.

3.7.2.3 suspend_set

```
sigset_t spthread_meta_st::suspend_set
```

Definition at line 59 of file spthread.c.

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

- SRC/lib/spthread.c

3.8 spthread_signal_args_st Struct Reference

Public Attributes

- const int signal
- volatile sig_atomic_t ack
- pthread_mutex_t shutup_mutex

3.8.1 Detailed Description

Definition at line 46 of file spthread.c.

3.8.2 Member Data Documentation

3.8.2.1 ack

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

Definition at line 48 of file spthread.c.

3.8.2.2 shutup_mutex

```
pthread_mutex_t spthread_signal_args_st::shutup_mutex
```

Definition at line 49 of file spthread.c.

3.8.2.3 signal

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

Definition at line 47 of file spthread.c.

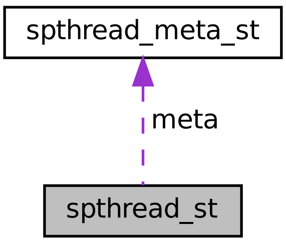
The documentation for this struct was generated from the following file:

- SRC/lib/spthread.c

3.9 spthread_st Struct Reference

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

Collaboration diagram for spthread_st:



Public Attributes

- pthread_t thread
- spthread_meta_t * meta

3.9.1 Detailed Description

Definition at line 28 of file spthread.h.

3.9.2 Member Data Documentation

3.9.2.1 meta

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

Definition at line 30 of file spthread.h.

3.9.2.2 thread

```
pthread_t spthread_st::thread
```

Definition at line 29 of file spthread.h.

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

- SRC/lib/spthread.h

3.10 vec_st Struct Reference

```
#include <Vec.h>
```

Public Attributes

- ptr_t * data
- size_t length
- size_t capacity
- ptr_dtor_fn ele_dtor_fn

3.10.1 Detailed Description

Definition at line 10 of file Vec.h.

3.10.2 Member Data Documentation

3.10.2.1 capacity

```
size_t vec_st::capacity
```

Definition at line 13 of file Vec.h.

3.10.2.2 data

```
ptr_t* vec_st::data
```

Definition at line 11 of file Vec.h.

3.10.2.3 ele_dtor_fn

```
ptr_dtor_fn vec_st::ele_dtor_fn
```

Definition at line 14 of file Vec.h.

3.10.2.4 length

```
size_t vec_st::length
```

Definition at line 12 of file Vec.h.

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

- SRC/lib/Vec.h

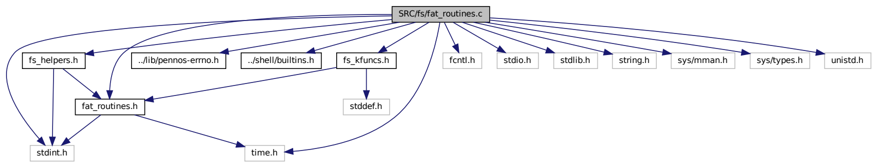
Chapter 4

File Documentation

4.1 SRC/fs/fat_routines.c File Reference

```
#include "fat_routines.h"
#include "../lib/pennos-errno.h"
#include "../shell/builtins.h"
#include "fs_helpers.h"
#include "fs_kfuncs.h"
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
```

Include dependency graph for fat_routines.c:



Functions

- int mkfs (const char ∗fs_name, int num_blocks, int blk_size)

  Creates a PennFAT filesystem in the file named fs_name at the OS-level.
- int mount (const char ∗fs_name)

  Mounts a filesystem with name fs_name by loading its FAT into memory.
- int unmount ()

  Unmounts the current filesystem and reset variables.
- void ∗ cat (void ∗arg)

  Concatenates and displays files.

- void * ls (void *arg)

  Searches root directory and lists all files in the directory.
- void * touch (void *arg)

  Creates files or updates timestamps.
- void * mv (void *arg)

  Renames files.
- void * cp (void *arg)

  Copies the source file to the destination.
- void * rm (void *arg)

  Removes files.
- void * chmod (void *arg)

  Changes the permissions of a file.
- void * cmpctdir (void *arg)

  Implements compaction of root directory.

4.1.1 Function Documentation

4.1.1.1 cat()

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

Concatenates and displays files.

This function reads the content of files and writes it to stdout or to another file. It supports reading from stdin when no input files are specified.

Usage formats:

- cat FILE ... (displays content to stdout)
- cat FILE ... -w OUTPUT_FILE (writes content to OUTPUT_FILE, overwriting)
- cat FILE ... -a OUTPUT_FILE (appends content to OUTPUT_FILE)
- cat -w OUTPUT_FILE (reads from stdin, writes to OUTPUT_FILE)
- cat -a OUTPUT_FILE (reads from stdin, appends to OUTPUT_FILE)

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 209 of file fat_routines.c.

```
                        {
  char** args = (char**)arg;

  // verify that the file system is mounted
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    u_perror("cat");
    return NULL;
  }

  // early return if there is nothing after cat
  if (args[1] == NULL) {
    P_ERRNO = P_EINVAL;
    u_perror("cat");
    return NULL;
  }

  // check for output file with -w or -a flag
  int out_fd = -1;
  int out_mode = 0;

  // scan arguments and determine output fd and output mode
  int i;
  for (i = 1; args[i] != NULL; i++) {
    if (strcmp(args[i], "-w") == 0 && args[i + 1] != NULL) {
      out_mode = F_WRITE;
      out_fd = k_open(args[i + 1], F_WRITE);
      if (out_fd < 0) {
        u_perror("cat");
        return NULL;
      }
      break;
    } else if (strcmp(args[i], "-a") == 0 && args[i + 1] != NULL) {
      out_mode = F_APPEND;
      out_fd = k_open(args[i + 1], F_APPEND);
      if (out_fd < 0) {
        u_perror("cat");
        return NULL;
      }
      break;
    }
  }

  // if no output redirection found, use STDOUT
  if (out_fd < 0) {
    out_fd = STDOUT_FILENO;
  }

  // handle small case: cat -w OUTPUT_FILE or cat -a OUTPUT_FILE (read from stdin)
  if ((strcmp(args[1], "-w") == 0 || strcmp(args[1], "-a") == 0) &&
      args[2] != NULL && args[3] == NULL) {
    char buffer[1024];

    while (1) {
      ssize_t bytes_read = k_read(STDIN_FILENO, buffer, sizeof(buffer));

      if (bytes_read < 0) {
        u_perror("cat");
        if (out_fd != STDOUT_FILENO) {
          k_close(out_fd);
        }
        return NULL;
      }

      if (bytes_read == 0) {
        break;
      }

      if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
        u_perror("cat");
        if (out_fd != STDOUT_FILENO) {
          k_close(out_fd);
        }
        return NULL;
      }
    }

    if (out_fd != STDOUT_FILENO) {
      k_close(out_fd);
    }
    return NULL;
  }

  // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
  int start = 1;
  int end = i - 1;

```

```
  if (out_mode != 0) {
    end = i - 1;  // skip the output redirection arguments
  }

  // process each input file
  for (i = start; i <= end; i++) {
    // skip the redirection flags and their arguments
    if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
      i++;
      continue;
    }

    // open the current input file
    int in_fd = k_open(args[i], F_READ);
    if (in_fd < 0) {
      u_perror("cat");
      continue;
    }

    // use lseek to get the size of in_fd
    off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
    if (in_fd_size == -1) {
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    // use lseek to reset position to 0 for reading
    if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    // copy file content to output
    char* buffer = (char*)malloc(block_size);
    if (buffer == NULL) {
      P_ERRNO = P_EMALLOC;
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    int bytes_read;
    ssize_t bytes_remaining = in_fd_size;

    while (bytes_remaining > 0) {
      ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
      bytes_read = k_read(in_fd, buffer, bytes_to_read);

      if (bytes_read <= 0) {
        break;
      }

      if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
        free(buffer);
        k_close(in_fd);
        u_perror("cat");
        break;
      }

      bytes_remaining -= bytes_read;
    }

    // read error
    if (bytes_read < 0) {
      free(buffer);
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    k_close(in_fd);
    free(buffer);
  }

  // close output file if not stdout
  if (out_fd != STDOUT_FILENO) {
    k_close(out_fd);
  }

  return NULL;
}
```

4.1.1.2 chmod()

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

Changes the permissions of a file.

Changes file permissions.

- chmod +x FILE (adds executable permission)
- chmod +rw FILE (adds read and write permissions)
- chmod -wx FILE (removes write and executable permissions)

Definition at line 751 of file fat_routines.c.

```
751                         {
752   char** args = (char**)arg;
753   if (!args || !args[0] || !args[1] || !args[2]) {
754     P_ERRNO = P_EINVAL;
755     return NULL;
756   }
757
758   // Parse permission string
759   const char* perm_str = args[1];
760   if (perm_str[0] != '+' && perm_str[0] != '-') {
761     P_ERRNO = P_EINVAL;
762     return NULL;
763   }
764
765   // Find the file and get its current directory entry
766   dir_entry_t dir_entry;
767   int entry_offset = find_file(args[2], &dir_entry);
768   if (entry_offset < 0) {
769     P_ERRNO = P_ENOENT;
770     return NULL;
771   }
772
773   // Calculate new permissions
774   uint8_t new_perm = dir_entry.perm;
775   int i = 1;  // Start after + or -
776   while (perm_str[i] != '\0') {
777     switch (perm_str[i]) {
778       case 'r':
779         if (perm_str[0] == '+') {
780           new_perm |= PERM_READ;
781         } else {
782           new_perm &= ~PERM_READ;
783         }
784         break;
785       case 'w':
786         if (perm_str[0] == '+') {
787           new_perm |= PERM_WRITE;
788         } else {
789           new_perm &= ~PERM_WRITE;
790         }
791         break;
792       case 'x':
793         if (perm_str[0] == '+') {
794           new_perm |= PERM_EXEC;
795         } else {
796           new_perm &= ~PERM_EXEC;
797         }
798         break;
799       default:
800         P_ERRNO = P_EINVAL;
801         return NULL;
802     }
803     i++;
804   }
805
806   // Update the directory entry
807   dir_entry.perm = new_perm;
808   dir_entry.mtime = time(NULL);
809
810   // Seek to the entry's position
811   if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
812     P_ERRNO = P_ELSEEK;
```

```
813     return NULL;
814   }
815
816   // Write the updated entry back
817   if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
818     P_ERRNO = P_EWRITE;
819     return NULL;
820   }
821
822   return NULL;
823 }
```

4.1.1.3 cmpctdir()

```
void* cmpctdir (
            void * arg )
```

Implements compaction of root directory.

Compacts the root directory by removing all deleted entries.

Definition at line 832 of file fat_routines.c.

```
832                         {
833   if (!is_mounted) {
834     P_ERRNO = P_EFS_NOT_MOUNTED;
835     u_perror("cmpctdir");
836     return NULL;
837   }
838
839   if (compact_directory() != 0) {
840     u_perror("cmpctdir");
841   }
842
843   return NULL;
844 }
```

4.1.1.4 cp()

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

Copies the source file to the destination.

Copies files.

Definition at line 621 of file fat_routines.c.

```
621                   {
622   char** args = (char**)arg;
623
624   // check that we have enough arguments
625   if (args[1] == NULL || args[2] == NULL) {
626     P_ERRNO = P_EINVAL;
627     u_perror("cp");
628     return NULL;
629   }
630
631   // cp -h SOURCE DEST
632   if (strcmp(args[1], "-h") == 0) {
633     if (args[2] == NULL || args[3] == NULL) {
634       P_ERRNO = P_EINVAL;
635       u_perror("cp");
636       return NULL;
637     }
638
639     if (copy_host_to_pennfat(args[2], args[3]) != 0) {
```

```
      u_perror("cp");
      return NULL;
    }
    return NULL;
  }

  // cp SOURCE -h DEST
  if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
    if (args[3] == NULL) {
      P_ERRNO = P_EINVAL;
      u_perror("cp");
      return NULL;
    }

    if (copy_pennfat_to_host(args[1], args[3]) != 0) {
      u_perror("cp");
      return NULL;
    }
    return NULL;
  }

  // cp SOURCE DEST
  if ((args[1] != NULL && strcmp(args[1], "-h") != 0) &&
      (args[2] != NULL && strcmp(args[2], "-h") != 0) && args[3] == NULL) {
    if (copy_source_to_dest(args[1], args[2]) != 0) {
      u_perror("cp");
      return NULL;
    }
    return NULL;
  }

  P_ERRNO = P_EUNKNOWN;
  u_perror("cp");
  return NULL;
}
```

4.1.1.5 ls()

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

Searches root directory and lists all files in the directory.

Lists files in the current directory.

Definition at line 383 of file fat_routines.c.

```
383                     {
384   if (!is_mounted) {
385     P_ERRNO = P_EFS_NOT_MOUNTED;
386     u_perror("ls");
387     return NULL;
388   }
389
390   // start at root directory block
391   uint16_t current_block = 1;
392   int offset = 0;
393   dir_entry_t dir_entry;
394
395   while (1) {
396     // adjust pointer to beginning of current block
397     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
398       P_ERRNO = P_ELSEEK;
399       u_perror("ls");
400       return NULL;
401     }
402
403     offset = 0;
404
405     // search current block
406     while (offset < block_size) {
407       if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
408         P_ERRNO = P_EREAD;
409         u_perror("ls");
410         return NULL;
411       }
412
```

```
      // check if we've reached the end of directory
      if (dir_entry.name[0] == 0) {
        break;
      }

      // skip deleted entries
      if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
        offset += sizeof(dir_entry);
        continue;
      }

      // format permission string
      char perm_str[4] = "---";
      if (dir_entry.perm & PERM_READ)
        perm_str[0] = 'r';
      if (dir_entry.perm & PERM_WRITE)
        perm_str[1] = 'w';
      if (dir_entry.perm & PERM_EXEC)
        perm_str[2] = 'x';

      // format time
      struct tm* tm_info = localtime(&dir_entry.mtime);
      char time_str[50];
      strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);

      // print entry details
      char buffer[128];
      int len;
      if (dir_entry.firstBlock == 0) {
        len = snprintf(buffer, sizeof(buffer), "   -%s- %6d %s %s\n",
                perm_str, dir_entry.size, time_str, dir_entry.name);
      } else {
        len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
                dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
      }

      if (len < 0 || len >= (int)sizeof(buffer)) {
        P_ERRNO = P_EUNKNOWN;
        u_perror("ls");
        return NULL;
      }

      if (k_write(STDOUT_FILENO, buffer, len) != len) {
        P_ERRNO = P_EWRITE;
        u_perror("ls");
        return NULL;
      }

      offset += sizeof(dir_entry);
    }

    // move to the next block if there is one
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
      continue;
    }

    // no more blocks to search
    break;
  }

  return NULL;
}
```

4.1.1.6 mkfs()

```
int mkfs (
        const char * fs_name,
        int num_blocks,
        int blk_size )
```

Creates a PennFAT filesystem in the file named fs_name at the OS-level.

Creates a PennFAT filesystem in the file named fs_name.

Definition at line 29 of file fat_routines.c.

```
                                                             {
  // validate arguments
  if (num_blocks < 1 || num_blocks > 32) {
    P_ERRNO = P_EINVAL;
    return -1;
  }
  if (blk_size < 0 || blk_size > 4) {
    P_ERRNO = P_EINVAL;
    return -1;
  }

  // determine the file system size
  int block_sizes[] = {256, 512, 1024, 2048, 4096};
  int actual_block_size = block_sizes[blk_size];
  int fat_size = num_blocks * actual_block_size;
  int fat_entries = fat_size / 2;
  int num_data_blocks =
      (num_blocks == 32)
          ? fat_entries - 2
          : fat_entries - 1;  // note: first entry is reserved for metadata!
  size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);

  // create the file for the filesystem
  int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
  if (fd == -1) {
    P_ERRNO = P_EOPEN;
    return -1;
  }

  // extend the file to the required size
  if (ftruncate(fd, filesystem_size) == -1) {
    P_ERRNO = P_EFUNC;
    close(fd);
    return -1;
  }

  // allocate the FAT
  uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
  if (!temp_fat) {
    P_ERRNO = P_EMALLOC;
    close(fd);
    return -1;
  }

  // initialize FAT entries to their correct values
  temp_fat[0] = (num_blocks « 8) | blk_size;
  temp_fat[1] = FAT_EOF;
  for (int i = 2; i < fat_entries; i++) {
    temp_fat[i] = FAT_FREE;
  }

  // write the FAT to the file
  if (write(fd, temp_fat, fat_size) != fat_size) {
    P_ERRNO = P_EWRITE;
    free(temp_fat);
    close(fd);
    return -1;
  }

  // initialize the root directory + write to memory
  uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
  if (lseek(fd, fat_size, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    free(temp_fat);
    free(root_dir);
    close(fd);
    return -1;
  }
  if (write(fd, root_dir, actual_block_size) != actual_block_size) {
    P_ERRNO = P_EWRITE;
    free(temp_fat);
    free(root_dir);
    close(fd);
    return -1;
  }

  // clean up
  free(temp_fat);
  free(root_dir);
  close(fd);
  return 0;
}
```

4.1.1.7 mount()

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

Mounts a filesystem with name fs_name by loading its FAT into memory.

Mounts the filesystem named fs_name by loading its FAT into memory.

Definition at line 115 of file fat_routines.c.

```
115                         {
116   // check if a filesystem is already mounted
117   if (is_mounted) {
118     P_ERRNO = P_EBUSY;
119     return -1;
120   }
121
122   // open the file with fs_name + set the global fs_fd
123   fs_fd = open(fs_name, O_RDWR);
124   if (fs_fd == -1) {
125     P_ERRNO = P_ENOENT;
126     return -1;
127   }
128
129   // read the first two bytes to get size configuration
130   uint16_t config;
131   if (read(fs_fd, &config, sizeof(config)) != sizeof(config)) {
132     P_ERRNO = P_EREAD;
133     close(fs_fd);
134     fs_fd = -1;
135     return -1;
136   }
137
138   // extract FAT region size information
139   num_fat_blocks = (config » 8) & 0xFF;  // MSB
140   int block_size_config = config & 0xFF;  // LSB
141   int block_sizes[] = {256, 512, 1024, 2048, 4096};
142   block_size = block_sizes[block_size_config];
143   fat_size = num_fat_blocks * block_size;
144
145   // map the FAT region into memory
146   if (lseek(fs_fd, 0, SEEK_SET) == -1) {
147     P_ERRNO = P_ELSEEK;
148     close(fs_fd);
149     fs_fd = -1;
150     return -1;
151   }
152
153   fat = mmap(NULL, fat_size, PROT_READ | PROT_WRITE, MAP_SHARED, fs_fd, 0);
154   if (fat == MAP_FAILED) {
155     P_ERRNO = P_EMAP;
156     close(fs_fd);
157     fs_fd = -1;
158     return -1;
159   }
160
161   init_fd_table(fd_table);  // initialize the file descriptor table
162   is_mounted = true;
163   return 0;
164 }
```

4.1.1.8 mv()

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

Renames files.

Renames the source file to the destination name. If the destination file already exists, it will be overwritten.

Usage: mv SOURCE DEST

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 543 of file fat_routines.c.

```
543                         {
544   char** args = (char**)arg;
545 
546   // verify that the file system is mounted
547   if (!is_mounted) {
548     P_ERRNO = P_EFS_NOT_MOUNTED;
549     u_perror("mv");
550     return NULL;
551   }
552 
553   // check if we have both source and destination arguments
554   if (args[1] == NULL || args[2] == NULL) {
555     P_ERRNO = P_EINVAL;
556     u_perror("mv");
557     return NULL;
558   }
559 
560   char* source = args[1];
561   char* dest = args[2];
562 
563   // check if they're trying to rename to the same name
564   if (strcmp(source, dest) == 0) {
565     return NULL;
566   }
567 
568   // check if source file exists
569   dir_entry_t source_entry;
570   int source_offset = find_file(source, &source_entry);
571   if (source_offset < 0) {
572     u_perror("mv");
573     return NULL;
574   }
575 
576   // check if the destination file already exists
577   dir_entry_t dest_entry;
578   int dest_offset = find_file(dest, &dest_entry);
579 
580   // destination file exists
581   if (dest_offset >= 0) {
582     // check if the destination file is currently open by any process
583     for (int i = 0; i < MAX_FDS; i++) {
584       if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {
585         P_ERRNO = P_EBUSY;
586         u_perror("mv");
587         return NULL;
588       }
589     }
590 
591     // if destination file exists, delete it
592     if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
593       u_perror("mv");
594       return NULL;
595     }
596   }
597 
598   // rename file
599   strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
600   source_entry.name[sizeof(source_entry.name) - 1] ='\0';
601 
602   // write the updated entry back to disk
603   if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
604     P_ERRNO = P_ELSEEK;
605     u_perror("mv");
606     return NULL;
607   }
608 
609   if (write(fs_fd, &source_entry, sizeof(source_entry)) != sizeof(source_entry)) {
610     P_ERRNO = P_EWRITE;
611     u_perror("mv");
612     return NULL;
613   }
```

```c

  return NULL;
}
```

4.1.1.9 rm()

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

Removes files.

Deletes one or more files from the filesystem. Each file is processed as a separate transaction.

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 679 of file fat_routines.c.

```c
679                    {
680   char** args = (char**)arg;
681 
682   // verify that the file system is mounted
683   if (!is_mounted) {
684     P_ERRNO = P_EFS_NOT_MOUNTED;
685     u_perror("rm");
686     return NULL;
687   }
688 
689   // check if we have any arguments
690   if (args[1] == NULL) {
691     P_ERRNO = P_EINVAL;
692     u_perror("rm");
693     return NULL;
694   }
695 
696   // process each file argument
697   for (int i = 1; args[i] != NULL; i++) {
698     // find the file in the directory
699     dir_entry_t entry;
700     int entry_offset = find_file(args[i], &entry);
701 
702     if (entry_offset < 0) {
703       // file doesn't exist
704       P_ERRNO = P_ENOENT;
705       u_perror("rm");
706       continue;
707     }
708 
709     // check if file is currently open
710     for (int j = 0; j < MAX_FDS; j++) {
711       if (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
712         P_ERRNO = P_EBUSY;
713         u_perror("rm");
714         continue;
715       }
716     }
717 
718     // mark the directory entry as deleted
719     if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
720       P_ERRNO = P_ELSEEK;
721       u_perror("rm");
722       continue;
723     }
724 
```

```
    char deleted = 1;  // mark as deleted
    if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
      P_ERRNO = P_EWRITE;
      u_perror("rm");
      continue;
    }

    // free the FAT chain for this file
    uint16_t block = entry.firstBlock;
    while (block != FAT_FREE && block != FAT_EOF) {
      uint16_t next_block = fat[block];
      fat[block] = FAT_FREE;
      block = next_block;
    }
  }

  return NULL;
}
```

4.1.1.10 touch()

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

Creates files or updates timestamps.

Creates empty files or updates timestamps.

For each file argument, creates the file if it doesn't exist, or updates its timestamp if it already exists.

Definition at line 483 of file fat_routines.c.

```
483                         {
484   char** args = (char**)arg;
485
486   // verify that the file system is mounted
487   if (!is_mounted) {
488     P_ERRNO = P_EFS_NOT_MOUNTED;
489     u_perror("touch");
490     return NULL;
491   }
492
493   // check if we have any arguments
494   if (args[1] == NULL) {
495     P_ERRNO = P_EINVAL;
496     u_perror("touch");
497     return NULL;
498   }
499
500   // process each file argument
501   for (int i = 1; args[i] != NULL; i++) {
502     dir_entry_t entry;
503     int entry_offset = find_file(args[i], &entry);
504
505     // file exists
506     if (entry_offset >= 0) {
507       entry.mtime = time(NULL);
508
509       // write the updated entry back to the directory
510       if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
511         P_ERRNO = P_ELSEEK;
512         u_perror("touch");
513         continue;
514       }
515       if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
516         P_ERRNO = P_EWRITE;
517         u_perror("touch");
518         continue;
519       }
520     } else {
521       // file doesn't exist, create a new empty file
522
523       // check if the fat is full
524       if (P_ERRNO == P_EFULL) {
525         u_perror("touch");
526         return NULL;
```

```
      }

      // add the file entry to root directory
      if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
        u_perror("touch");
        continue;
      }
    }
  }

  return NULL;
}
```

4.1.1.11 unmount()

```
int unmount ( )
```

Unmounts the current filesystem and reset variables.

Unmounts the currently mounted filesystem.

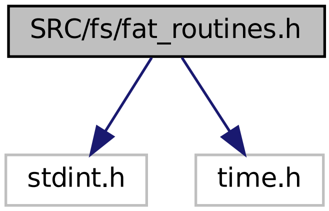
Definition at line 169 of file fat_routines.c.

```
169             {
170   // first check that a file system is actually mounted
171   if (!is_mounted) {
172     P_ERRNO = P_EFS_NOT_MOUNTED;
173     return -1;
174   }
175 
176   // unmap the FAT
177   if (fat != NULL) {
178     if (munmap(fat, fat_size) == -1) {
179       P_ERRNO = P_EMAP;
180       return -1;
181     }
182     fat = NULL;
183   }
184 
185   // close fs_fd
186   if (fs_fd != -1) {
187     if (close(fs_fd) == -1) {
188       P_ERRNO = P_ECLOSE;
189       return -1;
190     }
191     fs_fd = -1;
192   }
193 
194   // reset the other globals
195   num_fat_blocks = 0;
196   block_size = 0;
197   fat_size = 0;
198   is_mounted = false;
199   return 0;
200 }
```

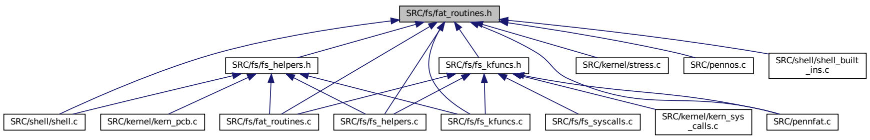
4.2 SRC/fs/fat_routines.h File Reference

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

Include dependency graph for fat_routines.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct dir_entry_t

  Directory entry structure for files in the filesystem.
- struct fd_entry_t

  File descriptor entry structure for open files.

Macros

- #define FAT_EOF 0xFFFF
- #define FAT_FREE 0x0000
- #define TYPE_UNKNOWN 0
- #define TYPE_REGULAR 1
- #define TYPE_DIRECTORY 2
- #define TYPE_SYMLINK 4
- #define PERM_NONE 0
- #define PERM_WRITE 1
- #define PERM_READ 2
- #define PERM_EXEC 4
- #define PERM_READ_WRITE (PERM_READ | PERM_WRITE)
- #define PERM_READ_EXEC (PERM_READ | PERM_EXEC)
- #define PERM_READ_WRITE_EXEC (PERM_READ | PERM_WRITE | PERM_EXEC)
- #define F_READ 0x01
- #define F_WRITE 0x02
- #define F_APPEND 0x04

Functions

- int mkfs (const char *fs_name, int num_blocks, int block_size)

  Creates a PennFAT filesystem in the file named fs_name.
- int mount (const char *fs_name)

  Mounts the filesystem named fs_name by loading its FAT into memory.
- int unmount ()

  Unmounts the currently mounted filesystem.
- void * cat (void *arg)

  Concatenates and displays files.
- void * ls (void *arg)

  Lists files in the current directory.
- void * touch (void *arg)

  Creates empty files or updates timestamps.
- void * mv (void *arg)

  Renames files.
- void * cp (void *arg)

  Copies files.
- void * rm (void *arg)

  Removes files.
- void * chmod (void *arg)

  Changes file permissions.
- void * cmpctdir (void *arg)

  Compacts the root directory by removing all deleted entries.

4.2.1 Macro Definition Documentation

4.2.1.1 F_APPEND

```
#define F_APPEND 0x04
```

Definition at line 37 of file fat_routines.h.

4.2.1.2 F_READ

```
#define F_READ 0x01
```

Definition at line 35 of file fat_routines.h.

4.2.1.3 F_WRITE

```
#define F_WRITE 0x02
```

Definition at line 36 of file fat_routines.h.

4.2.1.4 FAT_EOF

```
#define FAT_EOF 0xFFFF
```

Definition at line 16 of file fat_routines.h.

4.2.1.5 FAT_FREE

```
#define FAT_FREE 0x0000
```

Definition at line 17 of file fat_routines.h.

4.2.1.6 PERM_EXEC

```
#define PERM_EXEC 4
```

Definition at line 29 of file fat_routines.h.

4.2.1.7 PERM_NONE

```
#define PERM_NONE 0
```

Definition at line 26 of file fat_routines.h.

4.2.1.8 PERM_READ

```
#define PERM_READ 2
```

Definition at line 28 of file fat_routines.h.

4.2.1.9 PERM_READ_EXEC

```
#define PERM_READ_EXEC (PERM_READ | PERM_EXEC)
```

Definition at line 31 of file fat_routines.h.

4.2.1.10 PERM_READ_WRITE

```
#define PERM_READ_WRITE (PERM_READ | PERM_WRITE)
```

Definition at line 30 of file fat_routines.h.

4.2.1.11 PERM_READ_WRITE_EXEC

```
#define PERM_READ_WRITE_EXEC (PERM_READ | PERM_WRITE | PERM_EXEC)
```

Definition at line 32 of file fat_routines.h.

4.2.1.12 PERM_WRITE

```
#define PERM_WRITE 1
```

Definition at line 27 of file fat_routines.h.

4.2.1.13 TYPE_DIRECTORY

```
#define TYPE_DIRECTORY 2
```

Definition at line 22 of file fat_routines.h.

4.2.1.14 TYPE_REGULAR

```
#define TYPE_REGULAR 1
```

Definition at line 21 of file fat_routines.h.

4.2.1.15 TYPE_SYMLINK

```
#define TYPE_SYMLINK 4
```

Definition at line 23 of file fat_routines.h.

4.2.1.16 TYPE_UNKNOWN

```
#define TYPE_UNKNOWN 0
```

Definition at line 20 of file fat_routines.h.

4.2.2 Function Documentation

4.2.2.1 cat()

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

Concatenates and displays files.

This function reads the content of files and writes it to stdout or to another file. It supports reading from stdin when no input files are specified.

Usage formats:

- cat FILE ... (displays content to stdout)
- cat FILE ... -w OUTPUT_FILE (writes content to OUTPUT_FILE, overwriting)
- cat FILE ... -a OUTPUT_FILE (appends content to OUTPUT_FILE)
- cat -w OUTPUT_FILE (reads from stdin, writes to OUTPUT_FILE)
- cat -a OUTPUT_FILE (reads from stdin, appends to OUTPUT_FILE)

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 209 of file fat_routines.c.

```
                    {
char** args = (char**)arg;

// verify that the file system is mounted
if (!is_mounted) {
  P_ERRNO = P_EFS_NOT_MOUNTED;
  u_perror("cat");
  return NULL;
}

// early return if there is nothing after cat
if (args[1] == NULL) {
  P_ERRNO = P_EINVAL;
  u_perror("cat");
  return NULL;
}

// check for output file with -w or -a flag
int out_fd = -1;
int out_mode = 0;

// scan arguments and determine output fd and output mode
int i;
for (i = 1; args[i] != NULL; i++) {
  if (strcmp(args[i], "-w") == 0 && args[i + 1] != NULL) {
    out_mode = F_WRITE;
    out_fd = k_open(args[i + 1], F_WRITE);
    if (out_fd < 0) {
      u_perror("cat");
      return NULL;
    }
    break;
  } else if (strcmp(args[i], "-a") == 0 && args[i + 1] != NULL) {
    out_mode = F_APPEND;
    out_fd = k_open(args[i + 1], F_APPEND);
    if (out_fd < 0) {
      u_perror("cat");
      return NULL;
    }
    break;
  }
}

// if no output redirection found, use STDOUT
if (out_fd < 0) {
  out_fd = STDOUT_FILENO;
}

// handle small case: cat -w OUTPUT_FILE or cat -a OUTPUT_FILE (read from stdin)
if ((strcmp(args[1], "-w") == 0 || strcmp(args[1], "-a") == 0) &&
    args[2] != NULL && args[3] == NULL) {
  char buffer[1024];

  while (1) {
    ssize_t bytes_read = k_read(STDIN_FILENO, buffer, sizeof(buffer));

    if (bytes_read < 0) {
      u_perror("cat");
      if (out_fd != STDOUT_FILENO) {
        k_close(out_fd);
      }
      return NULL;
    }

    if (bytes_read == 0) {
      break;
    }

    if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
      u_perror("cat");
      if (out_fd != STDOUT_FILENO) {
        k_close(out_fd);
      }
      return NULL;
    }
  }

  if (out_fd != STDOUT_FILENO) {
    k_close(out_fd);
  }
  return NULL;
}

// handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
int start = 1;
int end = i - 1;

```

```
  if (out_mode != 0) {
    end = i - 1;  // skip the output redirection arguments
  }

  // process each input file
  for (i = start; i <= end; i++) {
    // skip the redirection flags and their arguments
    if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
      i++;
      continue;
    }

    // open the current input file
    int in_fd = k_open(args[i], F_READ);
    if (in_fd < 0) {
      u_perror("cat");
      continue;
    }

    // use lseek to get the size of in_fd
    off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
    if (in_fd_size == -1) {
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    // use lseek to reset position to 0 for reading
    if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    // copy file content to output
    char* buffer = (char*)malloc(block_size);
    if (buffer == NULL) {
      P_ERRNO = P_EMALLOC;
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    int bytes_read;
    ssize_t bytes_remaining = in_fd_size;

    while (bytes_remaining > 0) {
      ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
      bytes_read = k_read(in_fd, buffer, bytes_to_read);

      if (bytes_read <= 0) {
        break;
      }

      if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
        free(buffer);
        k_close(in_fd);
        u_perror("cat");
        break;
      }

      bytes_remaining -= bytes_read;
    }

    // read error
    if (bytes_read < 0) {
      free(buffer);
      k_close(in_fd);
      u_perror("cat");
      continue;
    }

    k_close(in_fd);
    free(buffer);
  }

  // close output file if not stdout
  if (out_fd != STDOUT_FILENO) {
    k_close(out_fd);
  }

  return NULL;
}
```

4.2.2.2 chmod()

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

Changes file permissions.

Modifies the permissions of the specified file.

Usage formats:

- chmod +x FILE (adds executable permission)
- chmod +rw FILE (adds read and write permissions)
- chmod -wx FILE (removes write and executable permissions)

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Changes file permissions.

- chmod +x FILE (adds executable permission)
- chmod +rw FILE (adds read and write permissions)
- chmod -wx FILE (removes write and executable permissions)

Definition at line 751 of file fat_routines.c.

```
751                         {
752   char** args = (char**)arg;
753   if (!args || !args[0] || !args[1] || !args[2]) {
754     P_ERRNO = P_EINVAL;
755     return NULL;
756   }
757
758   // Parse permission string
759   const char* perm_str = args[1];
760   if (perm_str[0] != '+' && perm_str[0] != '-') {
761     P_ERRNO = P_EINVAL;
762     return NULL;
763   }
764
765   // Find the file and get its current directory entry
766   dir_entry_t dir_entry;
767   int entry_offset = find_file(args[2], &dir_entry);
768   if (entry_offset < 0) {
769     P_ERRNO = P_ENOENT;
770     return NULL;
771   }
772
773   // Calculate new permissions
774   uint8_t new_perm = dir_entry.perm;
775   int i = 1;  // Start after + or -
776   while (perm_str[i] != '\0') {
777     switch (perm_str[i]) {
778       case 'r':
779         if (perm_str[0] == '+') {
780           new_perm |= PERM_READ;
```

```
        } else {
          new_perm &= ~PERM_READ;
        }
        break;
      case 'w':
        if (perm_str[0] == '+') {
          new_perm |= PERM_WRITE;
        } else {
          new_perm &= ~PERM_WRITE;
        }
        break;
      case 'x':
        if (perm_str[0] == '+') {
          new_perm |= PERM_EXEC;
        } else {
          new_perm &= ~PERM_EXEC;
        }
        break;
      default:
        P_ERRNO = P_EINVAL;
        return NULL;
    }
    i++;
  }

  // Update the directory entry
  dir_entry.perm = new_perm;
  dir_entry.mtime = time(NULL);

  // Seek to the entry's position
  if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    return NULL;
  }

  // Write the updated entry back
  if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
    P_ERRNO = P_EWRITE;
    return NULL;
  }

  return NULL;
}
```

4.2.2.3 cmpctdir()

```
void* cmpctdir (
              void * arg )
```

Compacts the root directory by removing all deleted entries.

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Compacts the root directory by removing all deleted entries.

Definition at line 832 of file fat_routines.c.

```
832                          {
833   if (!is_mounted) {
834     P_ERRNO = P_EFS_NOT_MOUNTED;
835     u_perror("cmpctdir");
836     return NULL;
837   }
838
```

```
  if (compact_directory() != 0) {
    u_perror("cmpctdir");
  }

  return NULL;
}
```

4.2.2.4 cp()

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

Copies files.

Copies the source file to the destination. If the destination file already exists, it will be overwritten.

Usage formats:

- cp SOURCE DEST (copies within PennFAT)
- cp -h SOURCE DEST (copies from host OS to PennFAT)
- cp SOURCE -h DEST (copies from PennFAT to host OS)

Parameters

arg	Arguments array (command line arguments)

Returns

return 0 on success, -1 on error

Copies files.

Definition at line 621 of file fat_routines.c.

```
621                     {
622   char** args = (char**)arg;
623
624   // check that we have enough arguments
625   if (args[1] == NULL || args[2] == NULL) {
626     P_ERRNO = P_EINVAL;
627     u_perror("cp");
628     return NULL;
629   }
630
631   // cp -h SOURCE DEST
632   if (strcmp(args[1], "-h") == 0) {
633     if (args[2] == NULL || args[3] == NULL) {
634       P_ERRNO = P_EINVAL;
635       u_perror("cp");
636       return NULL;
637     }
638
639     if (copy_host_to_pennfat(args[2], args[3]) != 0) {
640       u_perror("cp");
641       return NULL;
642     }
643     return NULL;
644   }
645
646   // cp SOURCE -h DEST
647   if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
```

```
    if (args[3] == NULL) {
      P_ERRNO = P_EINVAL;
      u_perror("cp");
      return NULL;
    }

    if (copy_pennfat_to_host(args[1], args[3]) != 0) {
      u_perror("cp");
      return NULL;
    }
    return NULL;
  }

  // cp SOURCE DEST
  if ((args[1] != NULL && strcmp(args[1], "-h") != 0) &&
      (args[2] != NULL && strcmp(args[2], "-h") != 0) && args[3] == NULL) {
    if (copy_source_to_dest(args[1], args[2]) != 0) {
      u_perror("cp");
      return NULL;
    }
    return NULL;
  }

  P_ERRNO = P_EUNKNOWN;
  u_perror("cp");
  return NULL;
}
```

4.2.2.5 ls()

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

Lists files in the current directory.

This function displays information about files in the current directory, including block number, permissions, size, and name.

Parameters

arg	Arguments array (command line arguments)

Returns

0 on success, -1 on error

Lists files in the current directory.

Definition at line 383 of file fat_routines.c.

```
383                        {
384   if (!is_mounted) {
385     P_ERRNO = P_EFS_NOT_MOUNTED;
386     u_perror("ls");
387     return NULL;
388   }
389
390   // start at root directory block
391   uint16_t current_block = 1;
392   int offset = 0;
393   dir_entry_t dir_entry;
394
395   while (1) {
396     // adjust pointer to beginning of current block
397     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
398       P_ERRNO = P_ELSEEK;
399       u_perror("ls");
400       return NULL;
```

```
    }

    offset = 0;

    // search current block
    while (offset < block_size) {
      if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
        P_ERRNO = P_EREAD;
        u_perror("ls");
        return NULL;
      }

      // check if we've reached the end of directory
      if (dir_entry.name[0] == 0) {
        break;
      }

      // skip deleted entries
      if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
        offset += sizeof(dir_entry);
        continue;
      }

      // format permission string
      char perm_str[4] = "---";
      if (dir_entry.perm & PERM_READ)
        perm_str[0] = 'r';
      if (dir_entry.perm & PERM_WRITE)
        perm_str[1] = 'w';
      if (dir_entry.perm & PERM_EXEC)
        perm_str[2] = 'x';

      // format time
      struct tm* tm_info = localtime(&dir_entry.mtime);
      char time_str[50];
      strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);

      // print entry details
      char buffer[128];
      int len;
      if (dir_entry.firstBlock == 0) {
        len = snprintf(buffer, sizeof(buffer), "   -%s- %6d %s %s\n",
                perm_str, dir_entry.size, time_str, dir_entry.name);
      } else {
        len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
                dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
      }

      if (len < 0 || len >= (int)sizeof(buffer)) {
        P_ERRNO = P_EUNKNOWN;
        u_perror("ls");
        return NULL;
      }

      if (k_write(STDOUT_FILENO, buffer, len) != len) {
        P_ERRNO = P_EWRITE;
        u_perror("ls");
        return NULL;
      }

      offset += sizeof(dir_entry);
    }

    // move to the next block if there is one
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
      continue;
    }

    // no more blocks to search
    break;
  }

  return NULL;
}
```

4.2.2.6 mkfs()

```
int mkfs (
        const char * fs_name,
```

```
int num_blocks,
int blk_size )
```

Creates a PennFAT filesystem in the file named fs_name.

This function initializes a new PennFAT filesystem with the specified parameters. The number of blocks in the FAT ranges from 1 through 32, and the block size is determined by block_size (0=256B, 1=512B, 2=1024B, 3=2048B, 4=4096B).

Parameters

fs_name	The name of the file to create the filesystem in.
num_blocks	The number of blocks in the FAT region (1-32).
block_size	The block size configuration (0-4).

Creates a PennFAT filesystem in the file named fs_name.

Definition at line 29 of file fat_routines.c.

```
29                                                                 {
30   // validate arguments
31   if (num_blocks < 1 || num_blocks > 32) {
32     P_ERRNO = P_EINVAL;
33     return -1;
34   }
35   if (blk_size < 0 || blk_size > 4) {
36     P_ERRNO = P_EINVAL;
37     return -1;
38   }
39 
40   // determine the file system size
41   int block_sizes[] = {256, 512, 1024, 2048, 4096};
42   int actual_block_size = block_sizes[blk_size];
43   int fat_size = num_blocks * actual_block_size;
44   int fat_entries = fat_size / 2;
45   int num_data_blocks =
46       (num_blocks == 32)
47           ? fat_entries - 2
48           : fat_entries - 1;  // note: first entry is reserved for metadata!
49   size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
50 
51   // create the file for the filesystem
52   int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
53   if (fd == -1) {
54     P_ERRNO = P_EOPEN;
55     return -1;
56   }
57 
58   // extend the file to the required size
59   if (ftruncate(fd, filesystem_size) == -1) {
60     P_ERRNO = P_EFUNC;
61     close(fd);
62     return -1;
63   }
64 
65   // allocate the FAT
66   uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
67   if (!temp_fat) {
68     P_ERRNO = P_EMALLOC;
69     close(fd);
70     return -1;
71   }
72 
73   // initialize FAT entries to their correct values
74   temp_fat[0] = (num_blocks « 8) | blk_size;
75   temp_fat[1] = FAT_EOF;
76   for (int i = 2; i < fat_entries; i++) {
77     temp_fat[i] = FAT_FREE;
78   }
79 
80   // write the FAT to the file
81   if (write(fd, temp_fat, fat_size) != fat_size) {
82     P_ERRNO = P_EWRITE;
83     free(temp_fat);
84     close(fd);
85     return -1;
86   }
```

```

  // initialize the root directory + write to memory
  uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
  if (lseek(fd, fat_size, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    free(temp_fat);
    free(root_dir);
    close(fd);
    return -1;
  }
  if (write(fd, root_dir, actual_block_size) != actual_block_size) {
    P_ERRNO = P_EWRITE;
    free(temp_fat);
    free(root_dir);
    close(fd);
    return -1;
  }

  // clean up
  free(temp_fat);
  free(root_dir);
  close(fd);
  return 0;
}
```

4.2.2.7 mount()

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

Mounts the filesystem named fs_name by loading its FAT into memory.

This function loads the filesystem's FAT into memory for subsequent operations. Only one filesystem can be mounted at a time.

Parameters

fs_name	The name of the filesystem file to mount.

Returns

0 on success, -1 on failure with P_ERRNO set.

Mounts the filesystem named fs_name by loading its FAT into memory.

Definition at line 115 of file fat_routines.c.

```
115                            {
116   // check if a filesystem is already mounted
117   if (is_mounted) {
118     P_ERRNO = P_EBUSY;
119     return -1;
120   }
121
122   // open the file with fs_name + set the global fs_fd
123   fs_fd = open(fs_name, O_RDWR);
124   if (fs_fd == -1) {
125     P_ERRNO = P_ENOENT;
126     return -1;
127   }
128
129   // read the first two bytes to get size configuration
130   uint16_t config;
131   if (read(fs_fd, &config, sizeof(config)) != sizeof(config)) {
132     P_ERRNO = P_EREAD;
133     close(fs_fd);
134     fs_fd = -1;
135     return -1;
```

```
  }

  // extract FAT region size information
  num_fat_blocks = (config » 8) & 0xFF;  // MSB
  int block_size_config = config & 0xFF;  // LSB
  int block_sizes[] = {256, 512, 1024, 2048, 4096};
  block_size = block_sizes[block_size_config];
  fat_size = num_fat_blocks * block_size;

  // map the FAT region into memory
  if (lseek(fs_fd, 0, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    close(fs_fd);
    fs_fd = -1;
    return -1;
  }

  fat = mmap(NULL, fat_size, PROT_READ | PROT_WRITE, MAP_SHARED, fs_fd, 0);
  if (fat == MAP_FAILED) {
    P_ERRNO = P_EMAP;
    close(fs_fd);
    fs_fd = -1;
    return -1;
  }

  init_fd_table(fd_table);  // initialize the file descriptor table
  is_mounted = true;
  return 0;
}
```

4.2.2.8 mv()

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

Renames files.

Renames the source file to the destination name. If the destination file already exists, it will be overwritten.

Usage: mv SOURCE DEST

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 543 of file fat_routines.c.

```
543                     {
544   char** args = (char**)arg;
545
546   // verify that the file system is mounted
547   if (!is_mounted) {
548     P_ERRNO = P_EFS_NOT_MOUNTED;
549     u_perror("mv");
550     return NULL;
551   }
552
553   // check if we have both source and destination arguments
554   if (args[1] == NULL || args[2] == NULL) {
555     P_ERRNO = P_EINVAL;
556     u_perror("mv");
557     return NULL;
558   }
559
```

```
  char* source = args[1];
  char* dest = args[2];

  // check if they're trying to rename to the same name
  if (strcmp(source, dest) == 0) {
    return NULL;
  }

  // check if source file exists
  dir_entry_t source_entry;
  int source_offset = find_file(source, &source_entry);
  if (source_offset < 0) {
    u_perror("mv");
    return NULL;
  }

  // check if the destination file already exists
  dir_entry_t dest_entry;
  int dest_offset = find_file(dest, &dest_entry);

  // destination file exists
  if (dest_offset >= 0) {
    // check if the destination file is currently open by any process
    for (int i = 0; i < MAX_FDS; i++) {
      if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {
        P_ERRNO = P_EBUSY;
        u_perror("mv");
        return NULL;
      }
    }

    // if destination file exists, delete it
    if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
      u_perror("mv");
      return NULL;
    }
  }

  // rename file
  strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
  source_entry.name[sizeof(source_entry.name) - 1] ='\0';

  // write the updated entry back to disk
  if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    u_perror("mv");
    return NULL;
  }

  if (write(fs_fd, &source_entry, sizeof(source_entry)) != sizeof(source_entry)) {
    P_ERRNO = P_EWRITE;
    u_perror("mv");
    return NULL;
  }

  return NULL;
}
```

4.2.2.9 rm()

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

Removes files.

Deletes one or more files from the filesystem. Each file is processed as a separate transaction.

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 679 of file fat_routines.c.

```
679                 {
680   char** args = (char**)arg;
681
682   // verify that the file system is mounted
683   if (!is_mounted) {
684     P_ERRNO = P_EFS_NOT_MOUNTED;
685     u_perror("rm");
686     return NULL;
687   }
688
689   // check if we have any arguments
690   if (args[1] == NULL) {
691     P_ERRNO = P_EINVAL;
692     u_perror("rm");
693     return NULL;
694   }
695
696   // process each file argument
697   for (int i = 1; args[i] != NULL; i++) {
698     // find the file in the directory
699     dir_entry_t entry;
700     int entry_offset = find_file(args[i], &entry);
701
702     if (entry_offset < 0) {
703       // file doesn't exist
704       P_ERRNO = P_ENOENT;
705       u_perror("rm");
706       continue;
707     }
708
709     // check if file is currently open
710     for (int j = 0; j < MAX_FDS; j++) {
711       if (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
712         P_ERRNO = P_EBUSY;
713         u_perror("rm");
714         continue;
715       }
716     }
717
718     // mark the directory entry as deleted
719     if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
720       P_ERRNO = P_ELSEEK;
721       u_perror("rm");
722       continue;
723     }
724
725     char deleted = 1;  // mark as deleted
726     if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
727       P_ERRNO = P_EWRITE;
728       u_perror("rm");
729       continue;
730     }
731
732     // free the FAT chain for this file
733     uint16_t block = entry.firstBlock;
734     while (block != FAT_FREE && block != FAT_EOF) {
735       uint16_t next_block = fat[block];
736       fat[block] = FAT_FREE;
737       block = next_block;
738     }
739   }
740
741   return NULL;
742 }
```

4.2.2.10 touch()

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

Creates empty files or updates timestamps.

For each file argument, creates the file if it doesn't exist, or updates its timestamp if it already exists.

Parameters

arg	Arguments array (command line arguments)

Returns

void pointer (unused)

Creates empty files or updates timestamps.

For each file argument, creates the file if it doesn't exist, or updates its timestamp if it already exists.

Definition at line 483 of file fat_routines.c.

```
483                               {
484   char** args = (char**)arg;
485
486   // verify that the file system is mounted
487   if (!is_mounted) {
488     P_ERRNO = P_EFS_NOT_MOUNTED;
489     u_perror("touch");
490     return NULL;
491   }
492
493   // check if we have any arguments
494   if (args[1] == NULL) {
495     P_ERRNO = P_EINVAL;
496     u_perror("touch");
497     return NULL;
498   }
499
500   // process each file argument
501   for (int i = 1; args[i] != NULL; i++) {
502     dir_entry_t entry;
503     int entry_offset = find_file(args[i], &entry);
504
505     // file exists
506     if (entry_offset >= 0) {
507       entry.mtime = time(NULL);
508
509       // write the updated entry back to the directory
510       if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
511         P_ERRNO = P_ELSEEK;
512         u_perror("touch");
513         continue;
514       }
515       if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
516         P_ERRNO = P_EWRITE;
517         u_perror("touch");
518         continue;
519       }
520     } else {
521       // file doesn't exist, create a new empty file
522
523       // check if the fat is full
524       if (P_ERRNO == P_EFULL) {
525         u_perror("touch");
526         return NULL;
527       }
528
529       // add the file entry to root directory
530       if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
531         u_perror("touch");
532         continue;
533       }
534     }
535   }
536
537   return NULL;
538 }
```

4.2.2.11 unmount()

`int unmount ( )`

Unmounts the currently mounted filesystem.

This function flushes any pending changes and unmounts the filesystem.

Returns

0 on success, -1 on failure with P_ERRNO set.

Unmounts the currently mounted filesystem.

Definition at line 169 of file fat_routines.c.

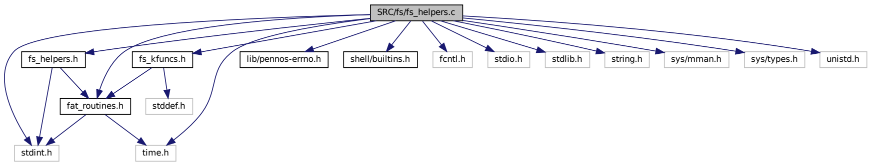
```
169                 {
170     // first check that a file system is actually mounted
171     if (!is_mounted) {
172       P_ERRNO = P_EFS_NOT_MOUNTED;
173       return -1;
174     }
175
176     // unmap the FAT
177     if (fat != NULL) {
178       if (munmap(fat, fat_size) == -1) {
179         P_ERRNO = P_EMAP;
180         return -1;
181       }
182       fat = NULL;
183     }
184
185     // close fs_fd
186     if (fs_fd != -1) {
187       if (close(fs_fd) == -1) {
188         P_ERRNO = P_ECLOSE;
189         return -1;
190       }
191       fs_fd = -1;
192     }
193
194     // reset the other globals
195     num_fat_blocks = 0;
196     block_size = 0;
197     fat_size = 0;
198     is_mounted = false;
199     return 0;
200 }
```

4.3 SRC/fs/fs_helpers.c File Reference

#include "fs_helpers.h"
#include "fat_routines.h"
#include "fs_kfuncs.h"
#include "lib/pennos-errno.h"
#include "shell/builtins.h"
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <time.h>

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

Include dependency graph for fs_helpers.c:



Functions

- void init_fd_table (fd_entry_t *fd_table)

    Initializes the global kernel-level file descriptor table.
- int get_free_fd (fd_entry_t *fd_table)

    Gets a free file descriptor.
- int increment_fd_ref_count (int fd)

    Increments the reference count of a file descriptor.
- int decrement_fd_ref_count (int fd)

    Decrements the reference count of a file descriptor.
- int has_executable_permission (int fd)

    Checks if a file has executable permissions.
- uint16_t allocate_block ()

    Allocates a block.
- int find_file (const char *filename, dir_entry_t *entry)

    Searches for a file in the root directory.
- int add_file_entry (const char *filename, uint32_t size, uint16_t first_block, uint8_t type, uint8_t perm)

    Adds a file to the root directory.
- int mark_entry_as_deleted (dir_entry_t *entry, int absolute_offset)

    Marks a file entry as deleted and frees its blocks.
- int copy_host_to_pennfat (const char *host_filename, const char *pennfat_filename)

    Copies data from host OS file to the PennFAT file.
- int copy_pennfat_to_host (const char *pennfat_filename, const char *host_filename)

    Copies data from PennFAT file to host OS file.
- int copy_source_to_dest (const char *source_filename, const char *dest_filename)

    Copies data from source file to destination file.
- int compact_directory ()

    Compacts a directory.

Variables

- int fs_fd = -1
- int block_size = 0
- int num_fat_blocks = 0
- int fat_size = 0
- uint16_t * fat = NULL
- bool is_mounted = false
- int MAX_FDS = 100
- fd_entry_t fd_table [100]

4.3.1 Function Documentation

4.3.1.1 add_file_entry()

```
int add_file_entry (
            const char * filename,
            uint32_t size,
            uint16_t first_block,
            uint8_t type,
            uint8_t perm )
```

Adds a file to the root directory.

Adds a new file entry to the root directory.

Definition at line 266 of file fs_helpers.c.

```
                                     {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // check if file already exists
  dir_entry_t existing;
  if (find_file(filename, &existing) >= 0) {
    P_ERRNO = P_EEXIST;
    return -1;
  }

  // start with root directory block (block 1)
  uint16_t current_block = 1;
  int offset = 0;
  dir_entry_t dir_entry;

  while (1) {
    // position at the start of current block of the root directory
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      return -1;
    }

    // reset offset for new block
    offset = 0;

    // search current block for free slot
    while (offset < block_size) {
      if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
        P_ERRNO = P_EREAD;
        return -1;
      }

      // found a free slot
      if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
        // initialize the new entry
        memset(&dir_entry, 0, sizeof(dir_entry));
        strncpy(dir_entry.name, filename, 31);
        dir_entry.size = size;
        dir_entry.firstBlock = first_block;
        dir_entry.type = type;
        dir_entry.perm = perm;
        dir_entry.mtime = time(NULL);

        // write the entry
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset, SEEK_SET) == -1) {
          P_ERRNO = P_ELSEEK;
          return -1;
        }
        if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
          P_ERRNO = P_EWRITE;
          return -1;
        }

        return offset;
```

```
      }

      offset += sizeof(dir_entry);
    }

    // current block is full, check if there's a next block
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
      continue;
    }

    // allocate a new block for the root directory
    uint16_t new_block = allocate_block();
    if (new_block == 0) {
      P_ERRNO = P_EFULL;
      return -1;
    }

    // chain the new block
    fat[current_block] = new_block;
    fat[new_block] = FAT_EOF;

    // initialize new block
    uint8_t* zero_block = calloc(block_size, 1);
    if (!zero_block) {
      P_ERRNO = P_EINVAL;
      return -1;
    }

    // write this new block to the file system
    if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      free(zero_block);
      return -1;
    }
    if (write(fs_fd, zero_block, block_size) != block_size) {
      P_ERRNO = P_EWRITE;
      free(zero_block);
      return -1;
    }

    free(zero_block);

    // initialize the new entry
    memset(&dir_entry, 0, sizeof(dir_entry));
    strncpy(dir_entry.name, filename, 31);
    dir_entry.size = size;
    dir_entry.firstBlock = first_block;
    dir_entry.type = type;
    dir_entry.perm = perm;
    dir_entry.mtime = time(NULL);

    // write the new entry at the start of the new block in the file system
    if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      return -1;
    }
    if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
      P_ERRNO = P_EWRITE;
      return -1;
    }

    return 0;
  }
}
```

4.3.1.2 allocate_block()

```
uint16_t allocate_block ( )
```

Allocates a block.

Allocates a free block in the FAT.

If no block found, we try compacting the directory.

Definition at line 167 of file fs_helpers.c.

```
167                         {
168   for (int i = 2; i < fat_size / 2; i++) {
169     if (fat[i] == FAT_FREE) {
170       fat[i] = FAT_EOF;
171       return i;
172     }
173   }
174
175   if (compact_directory() == 0) {
176     for (int i = 2; i < fat_size / 2; i++) {
177       if (fat[i] == FAT_FREE) {
178         fat[i] = FAT_EOF;
179         return i;
180       }
181     }
182   }
183
184   return 0;
185 }
```

4.3.1.3 compact_directory()

```
int compact_directory ( )
```

Compacts a directory.

Compacts the root directory by removing all deleted entries.

Definition at line 692 of file fs_helpers.c.

```
692                       {
693   if (!is_mounted) {
694     P_ERRNO = P_EFS_NOT_MOUNTED;
695     return -1;
696   }
697
698   // buffer for temp storage of a block
699   uint8_t* dir_buffer = malloc(block_size);
700   if (!dir_buffer) {
701     P_ERRNO = P_EMALLOC;
702     return -1;
703   }
704
705   // start at root directory
706   uint16_t current_block = 1;
707   int dir_entries_count = 0;
708   int deleted_entries_count = 0;
709
710   // calculate number of entries and deleted entries in the root directory
711   while (current_block != FAT_EOF) {
712     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
713       P_ERRNO = P_ELSEEK;
714       free(dir_buffer);
715       return -1;
716     }
717
718     if (read(fs_fd, dir_buffer, block_size) != block_size) {
719       P_ERRNO = P_EREAD;
720       free(dir_buffer);
721       return -1;
722     }
723
724     // count entries and deleted entries in this block
725     for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {
726       dir_entry_t* entry = (dir_entry_t*)(dir_buffer + offset);
727
728       // check if we've reached the end of directory
729       if (entry->name[0] == 0) {
730         break;
731       }
732
733       dir_entries_count++;
734
735       // check if it's a deleted entry
736       if (entry->name[0] == 1) {
737         deleted_entries_count++;
738       }
```

```
    }

    // move onto next block, if there is one
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
    } else {
      break;
    }
  }

  // if no deleted entries, no compaction needed
  if (deleted_entries_count == 0) {
    free(dir_buffer);
    return 0;
  }

  // allocate space for all valid entries
  dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
  if (!all_entries) {
    P_ERRNO = P_EMALLOC;
    free(dir_buffer);
    return -1;
  }

  // read all entries into the buffer, skipping deleted ones
  current_block = 1;
  int valid_entry_idx = 0;

  while (current_block != FAT_EOF) {
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    if (read(fs_fd, dir_buffer, block_size) != block_size) {
      P_ERRNO = P_EREAD;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    // process entries in this block
    for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {
      dir_entry_t* entry = (dir_entry_t*)(dir_buffer + offset);

      // check if we've reached the end of directory
      if (entry->name[0] == 0) {
        break;
      }

      // skip deleted entries
      if (entry->name[0] == 1) {
        continue;
      }

      // copy valid entry to our array
      memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
    }

    // move to the next block
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
    } else {
      break;
    }
  }

  // rewrite the directory with only valid entries
  current_block = 1;
  int entries_per_block = block_size / sizeof(dir_entry_t);
  int blocks_needed = (valid_entry_idx + entries_per_block - 1) / entries_per_block;

  // clean up any excess directory blocks in the FAT chain
  uint16_t next_block = fat[current_block];
  if (blocks_needed == 1) {
    // only need one block, free all others
    while (next_block != FAT_EOF) {
      uint16_t temp = fat[next_block];
      fat[next_block] = FAT_FREE;
      next_block = temp;
    }
    fat[current_block] = FAT_EOF;
  } else {
    // navigate through needed blocks
    int block_count = 1;
```

```
    uint16_t prev_block = current_block;

    while (block_count < blocks_needed) {
      if (next_block == FAT_EOF) {
        // need to allocate a new block
        uint16_t new_block = allocate_block();
        if (new_block == 0) {
          P_ERRNO = P_EFULL;
          free(dir_buffer);
          free(all_entries);
          return -1;
        }
        fat[prev_block] = new_block;
        next_block = new_block;
      }

      prev_block = next_block;
      next_block = fat[next_block];
      block_count++;
    }

    // free any excess blocks
    fat[prev_block] = FAT_EOF;
    while (next_block != FAT_EOF) {
      uint16_t temp = fat[next_block];
      fat[next_block] = FAT_FREE;
      next_block = temp;
    }
  }

  // write the valid entries back to the directory blocks
  current_block = 1;
  int entries_written = 0;

  while (entries_written < valid_entry_idx) {
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    memset(dir_buffer, 0, block_size);

    // copy entries to the buffer
    int entries_in_this_block = 0;
    while (entries_written < valid_entry_idx && entries_in_this_block < entries_per_block) {
      memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
              &all_entries[entries_written],
              sizeof(dir_entry_t));
      entries_written++;
      entries_in_this_block++;
    }

    // write the buffer to the file system
    if (write(fs_fd, dir_buffer, block_size) != block_size) {
      P_ERRNO = P_EINVAL;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    // move to the next block if needed
    if (entries_written < valid_entry_idx) {
      current_block = fat[current_block];
    }
  }

  free(dir_buffer);
  free(all_entries);
  return 0;
}
```

4.3.1.4 copy_host_to_pennfat()

```
int copy_host_to_pennfat (
              const char * host_filename,
              const char * pennfat_filename )
```

Copies data from host OS file to the PennFAT file.

Copies a file from the host OS to the PennFAT filesystem.

Definition at line 434 of file fs_helpers.c.

```
                                                            {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // open the host file
  int host_fd = open(host_filename, O_RDONLY);
  if (host_fd == -1) {
    P_ERRNO = P_EOPEN;
    return -1;
  }

  // determine file size by seeking to the end and getting position
  off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
  if (host_file_size_in_bytes == -1) {
    P_ERRNO = P_ELSEEK;
    close(host_fd);
    return -1;
  }

  // go back to beginning of file for reading
  if (lseek(host_fd, 0, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    close(host_fd);
    return -1;
  }

  // open the destination file in PennFAT
  int pennfat_fd = k_open(pennfat_filename, F_WRITE);
  if (pennfat_fd < 0) {
    close(host_fd);
    return -1;
  }

  // copy the data into this buffer
  uint8_t* buffer = (uint8_t*)malloc(block_size);
  if (!buffer) {
    P_ERRNO = P_EMALLOC;
    k_close(pennfat_fd);
    close(host_fd);
    return -1;
  }

  uint32_t bytes_remaining = host_file_size_in_bytes;
  ssize_t bytes_read;

  // read from host file
  while (bytes_remaining > 0) {
    // ensure bytes to read never exceeds the block size
    ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
    bytes_read = read(host_fd, buffer, bytes_to_read);

    if (bytes_read <= 0) {
      break;
    }

    // write to pennfat_fd using k_write
    if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
      free(buffer);
      k_close(pennfat_fd);
      close(host_fd);
      return -1;
    }

    bytes_remaining -= bytes_read;
  }

  // check for read error
  if (bytes_read < 0) {
    P_ERRNO = P_EREAD;
    free(buffer);
    k_close(pennfat_fd);
    close(host_fd);
    return -1;
  }

  // otherwise, cleanup and return success
  free(buffer);
  k_close(pennfat_fd);
```

```
  close(host_fd);
  return 0;
}
```

4.3.1.5 copy_pennfat_to_host()

```
int copy_pennfat_to_host (
            const char * pennfat_filename,
            const char * host_filename )
```

Copies data from PennFAT file to host OS file.

Copies a file from the PennFAT filesystem to the host OS.

Definition at line 522 of file fs_helpers.c.

```
                                                          {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // open the PennFAT file
  int pennfat_fd = k_open(pennfat_filename, F_READ);
  if (pennfat_fd < 0) {
    return -1;
  }

  // get the pennfat file size
  off_t pennfat_file_size_in_bytes = k_lseek(pennfat_fd, 0, SEEK_END);
  if (pennfat_file_size_in_bytes == -1) {
    k_close(pennfat_fd);
    return -1;
  }

  // go back to beginning of file for reading
  if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
    k_close(pennfat_fd);
    return -1;
  }

  // open the host file
  int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (host_fd == -1) {
    P_ERRNO = P_EOPEN;
    k_close(pennfat_fd);
    return -1;
  }

  // allocate buffer for data transfer
  char* buffer = (char*)malloc(block_size);
  if (!buffer) {
    P_ERRNO = P_EMALLOC;
    k_close(pennfat_fd);
    close(host_fd);
    return -1;
  }

  uint32_t bytes_remaining = pennfat_file_size_in_bytes;
  ssize_t bytes_read;

  // read from PennFAT file and write to host file
  while (bytes_remaining > 0) {
    // ensure bytes to read never exceeds the block size
    ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
    bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);

    if (bytes_read <= 0) {
      break;
    }

    if (write(host_fd, buffer, bytes_read) != bytes_read) {
      P_ERRNO = P_EINVAL;
      free(buffer);
      close(host_fd);
      k_close(pennfat_fd);
      return -1;
```

```
    }

    bytes_remaining -= bytes_read;
  }

  // check for read error
  if (bytes_read < 0) {
    P_ERRNO = P_EREAD;
    free(buffer);
    close(host_fd);
    k_close(pennfat_fd);
    return -1;
  }

  // otherwise, cleanup and return success
  free(buffer);
  close(host_fd);
  k_close(pennfat_fd);
  return 0;
}
```

4.3.1.6 copy_source_to_dest()

```
int copy_source_to_dest (
            const char * source_filename,
            const char * dest_filename )
```

Copies data from source file to destination file.

Copies a file from a source file to a destination file.

Definition at line 608 of file fs_helpers.c.

```
                                                   {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // open the source file
  int source_fd = k_open(source_filename, F_READ);
  if (source_fd < 0) {
    return -1;
  }

  // get the source file size
  off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
  if (source_file_size_in_bytes == -1) {
    k_close(source_fd);
    return -1;
  }

  // move to the beginning of the source file for reading
  if (k_lseek(source_fd, 0, SEEK_SET) < 0) {
    k_close(source_fd);
    return -1;
  }

  // open the destination file
  int dest_fd = k_open(dest_filename, F_WRITE);
  if (dest_fd < 0) {
    k_close(source_fd);
    return -1;
  }

  // read from source to destination
  char* buffer = (char*)malloc(block_size);
  if (!buffer) {
    P_ERRNO = P_EMALLOC;
    k_close(source_fd);
    k_close(dest_fd);
    return -1;
  }

  uint32_t bytes_remaining = source_file_size_in_bytes;
  ssize_t bytes_read;

```

```
  while (bytes_remaining > 0) {
    // make sure the bytes to read doesn't exceed block size
    ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
    bytes_read = k_read(source_fd, buffer, bytes_to_read);

    if (bytes_read <= 0) {
      break;
    }

    if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
      free(buffer);
      k_close(source_fd);
      k_close(dest_fd);
      return -1;
    }
  }

  // check for read error
  if (bytes_read < 0) {
    free(buffer);
    k_close(source_fd);
    k_close(dest_fd);
    return -1;
  }

  // otherwise, cleanup and return success
  free(buffer);
  k_close(source_fd);
  k_close(dest_fd);
  return 0;
}
```

4.3.1.7 decrement_fd_ref_count()

```
int decrement_fd_ref_count (
              int fd )
```

Decrements the reference count of a file descriptor.

If reference count reaches 0, flush field values.

Definition at line 107 of file fs_helpers.c.

```
107                                      {
108   if (fd < 0 || fd >= MAX_FDS) {
109     P_ERRNO = P_EBADF;
110     return -1;
111   }
112
113   if (!fd_table[fd].in_use) {
114     P_ERRNO = P_EBADF;
115     return -1;
116   }
117
118   fd_table[fd].ref_count--;
119   if (fd_table[fd].ref_count == 0) {
120     fd_table[fd].in_use = 0;
121     memset(fd_table[fd].filename, 0, sizeof(fd_table[fd].filename));
122     fd_table[fd].size = 0;
123     fd_table[fd].first_block = 0;
124     fd_table[fd].position = 0;
125     fd_table[fd].mode = 0;
126   }
127   return fd_table[fd].ref_count;
128 }
```

4.3.1.8 find_file()

```
int find_file (
            const char * filename,
            dir_entry_t * entry )
```

Searches for a file in the root directory.

Retrieves the file's absolute offset in the filesystem.

Definition at line 192 of file fs_helpers.c.

```
192                                                    {
193   if (!is_mounted) {
194     P_ERRNO = P_EFS_NOT_MOUNTED;
195     return -1;
196   }
197
198   // Start with root directory block (block 1)
199   uint16_t current_block = 1;
200   int offset_in_block = 0;
201   int absolute_offset = 0;
202   dir_entry_t dir_entry;
203
204   while (1) {
205     // Position at the start of current block
206     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
207       P_ERRNO = P_ELSEEK;
208       return -1;
209     }
210
211     // reset offset for new block
212     offset_in_block = 0;
213
214     // calculate the absolute offset
215     absolute_offset = fat_size + (current_block - 1) * block_size;
216
217     // search current block
218     while (offset_in_block < block_size) {
219       if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
220         P_ERRNO = P_EREAD;
221         return -1;
222       }
223
224       // check if we've reached the end of directory
225       if (dir_entry.name[0] == 0) {
226         break;
227       }
228
229       // check if this is a deleted entry
230       if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
231         offset_in_block += sizeof(dir_entry);
232         absolute_offset += sizeof(dir_entry);
233         continue;
234       }
235
236       // check if we found the file
237       if (strcmp(dir_entry.name, filename) == 0) {
238         if (entry) {
239           memcpy(entry, &dir_entry, sizeof(dir_entry));
240         }
241         return absolute_offset; // return the absolute file offset
242       }
243
244       offset_in_block += sizeof(dir_entry);
245       absolute_offset += sizeof(dir_entry);
246     }
247
248     // if we've reached the end of the current block, check if there's a next block
249     if (fat[current_block] != FAT_EOF) {
250       current_block = fat[current_block];
251       continue;
252     }
253
254     // no more blocks to search
255     break;
256   }
257
258   // file not found
259   P_ERRNO = P_ENOENT;
260   return -1;
261 }
```

4.3.1.9 get_free_fd()

```
int get_free_fd (
            fd_entry_t * fd_table )
```

Gets a free file descriptor.

Finds the first available file descriptor in the table.

Definition at line 77 of file fs_helpers.c.

```
77                                          {
78   for (int i = 3; i < MAX_FDS; i++) {
79     if (!fd_table[i].in_use) {
80       return i;
81     }
82   }
83   return -1;
84 }
```

4.3.1.10 has_executable_permission()

```
int has_executable_permission (
            int fd )
```

Checks if a file has executable permissions.

Checks if a file has executable permissions in the PennFAT filesystem.

Definition at line 133 of file fs_helpers.c.

```
133                                       {
134   // check if fs is mounted
135   if (!is_mounted) {
136     P_ERRNO = P_EFS_NOT_MOUNTED;
137     return -1;
138   }
139
140   // validate fd argument
141   if (fd < 0 || fd >= MAX_FDS) {
142     P_ERRNO = P_EINVAL;
143     return -1;
144   }
145
146
147   // determine whether the file exists
148   dir_entry_t entry;
149   int entry_offset = find_file(fd_table[fd].filename, &entry);
150   if (entry_offset < 0) {
151     return -1;
152   }
153
154   // if it exists, get its permission
155   if (entry.perm & PERM_EXEC) {
156     return 1;
157   }
158
159   return 0;
160 }
```

4.3.1.11 increment_fd_ref_count()

```
int increment_fd_ref_count (
            int fd )
```

Increments the reference count of a file descriptor.

Parameters

fd	file descriptor to increment

Returns

new reference count, or -1 on error

Definition at line 89 of file fs_helpers.c.

```
89                                    {
90    if (fd < 0 || fd >= MAX_FDS) {
91      P_ERRNO = P_EBADF;
92      return -1;
93    }
94    if (!fd_table[fd].in_use) {
95      P_ERRNO = P_EBADF;
96      return -1;
97    }
98    fd_table[fd].ref_count++;
99    return fd_table[fd].ref_count;
100 }
```

4.3.1.12 init_fd_table()

```
void init_fd_table (
            fd_entry_t * fd_table )
```

Initializes the global kernel-level file descriptor table.

Initializes all entries in the file descriptor table to not in use.

Definition at line 43 of file fs_helpers.c.

```
43                                          {
44    // STDIN (fd 0)
45    fd_table[0].in_use = 1;
46    fd_table[0].ref_count = 1;
47    strncpy(fd_table[0].filename, "<stdin>", 31);
48    fd_table[0].mode = F_READ;
49
50    // STDOUT (fd 1)
51    fd_table[1].in_use = 1;
52    strncpy(fd_table[1].filename, "<stdout>", 31);
53    fd_table[1].mode = F_WRITE; // write-only
54    fd_table[1].ref_count = 1;
55
56    // STDERR (fd 2)
57    fd_table[2].in_use = 1;
58    strncpy(fd_table[2].filename, "<stderr>", 31);
59    fd_table[2].mode = F_WRITE; // write-only
60    fd_table[2].ref_count = 1;
61
62    // other file descriptors (fd 3 and above)
63    for (int i = 3; i < MAX_FDS; i++) {
64      fd_table[i].in_use = 0;
65      fd_table[i].ref_count = 0;
66      memset(fd_table[i].filename, 0, sizeof(fd_table[i].filename));
67      fd_table[i].size = 0;
68      fd_table[i].first_block = 0;
69      fd_table[i].position = 0;
70      fd_table[i].mode = 0;
71    }
72 }
```

4.3.1.13 mark_entry_as_deleted()

```
int mark_entry_as_deleted (
            dir_entry_t * entry,
            int absolute_offset )
```

Marks a file entry as deleted and frees its blocks.

Marks a file entry as deleted and frees its blocks in the FAT.

Definition at line 396 of file fs_helpers.c.

```
396                                                                    {
397   if (!is_mounted || entry == NULL || absolute_offset < 0) {
398     P_ERRNO = P_EINVAL;
399     return -1;
400   }
401
402   // free the blocks
403   uint16_t current_block = entry->firstBlock;
404   while (current_block != FAT_FREE && current_block != FAT_EOF) {
405     uint16_t next_block = fat[current_block];
406     fat[current_block] = FAT_FREE;
407     current_block = next_block;
408   }
409
410   // mark the entry as deleted in the root directory
411   dir_entry_t deleted_entry = *entry;
412   deleted_entry.name[0] = 1;
413   if (lseek(fs_fd, absolute_offset, SEEK_SET) == -1) {
414     P_ERRNO = P_ELSEEK;
415     return -1;
416   }
417   if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) != sizeof(deleted_entry)) {
418     P_ERRNO = P_EINVAL;
419     return -1;
420   }
421
422   // mark the passed entry as deleted
423   entry->name[0] = 1;
424   return 0;
425 }
```

4.3.2 Variable Documentation

4.3.2.1 block_size

```
int block_size = 0
```

Definition at line 28 of file fs_helpers.c.

4.3.2.2 fat

```
uint16_t* fat = NULL
```

Definition at line 31 of file fs_helpers.c.

4.3.2.3 fat_size

```
int fat_size = 0
```

Definition at line 30 of file fs_helpers.c.

4.3.2.4 fd_table

```
fd_entry_t fd_table[100]
```

Definition at line 34 of file fs_helpers.c.

4.3.2.5 fs_fd

```
int fs_fd = -1
```

Definition at line 27 of file fs_helpers.c.

4.3.2.6 is_mounted

```
bool is_mounted = false
```

Definition at line 32 of file fs_helpers.c.

4.3.2.7 MAX_FDS

```
int MAX_FDS = 100
```

Definition at line 33 of file fs_helpers.c.

4.3.2.8 num_fat_blocks

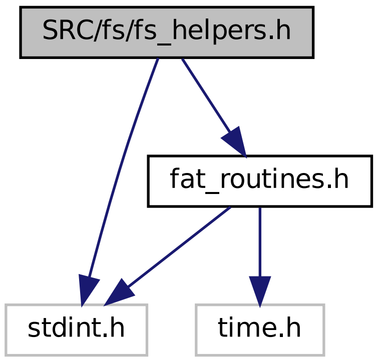
```
int num_fat_blocks = 0
```

Definition at line 29 of file fs_helpers.c.

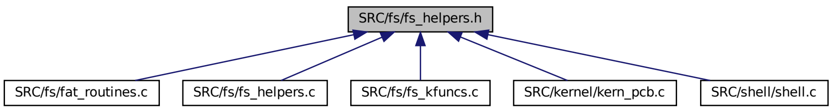
4.4 SRC/fs/fs_helpers.h File Reference

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

Include dependency graph for fs_helpers.h:



This graph shows which files directly or indirectly include this file:



Functions

- void init_fd_table (fd_entry_t ∗fd_table)

  Initializes all entries in the file descriptor table to not in use.
- int get_free_fd (fd_entry_t ∗fd_table)

  Finds the first available file descriptor in the table.
- int increment_fd_ref_count (int fd)

  Increments the reference count of a file descriptor.
- int decrement_fd_ref_count (int fd)

  Decrements the reference count of a file descriptor.
- int has_executable_permission (int fd)

  Checks if a file has executable permissions in the PennFAT filesystem.
- uint16_t allocate_block ()

  Allocates a free block in the FAT.
- int find_file (const char ∗filename, dir_entry_t ∗entry)

  Searches for a file in the root directory.
- int add_file_entry (const char ∗filename, uint32_t size, uint16_t first_block, uint8_t type, uint8_t perm)

Adds a new file entry to the root directory.
- int mark_entry_as_deleted (dir_entry_t *entry, int offset)

  Marks a file entry as deleted and frees its blocks in the FAT.
- int copy_host_to_pennfat (const char *host_filename, const char *pennfat_filename)

  Copies a file from the host OS to the PennFAT filesystem.
- int copy_pennfat_to_host (const char *pennfat_filename, const char *host_filename)

  Copies a file from the PennFAT filesystem to the host OS.
- int copy_source_to_dest (const char *source_filename, const char *dest_filename)

  Copies a file from a source file to a destination file.
- int compact_directory ()

  Compacts the root directory by removing all deleted entries.

Variables

- int fs_fd
- int block_size
- int num_fat_blocks
- int fat_size
- uint16_t * fat
- bool is_mounted
- int MAX_FDS
- fd_entry_t fd_table [100]

4.4.1 Function Documentation

4.4.1.1 add_file_entry()

```
int add_file_entry (
             const char * filename,
             uint32_t size,
             uint16_t first_block,
             uint8_t type,
             uint8_t perm )
```

Adds a new file entry to the root directory.

Parameters

filename	name of the file to add
size	size of the file in bytes
first_block	block number of the first block of the file
type	file type (regular, directory, etc.)
perm	file permissions

Returns

offset of the new entry in the directory if successful, -1 on error

Adds a new file entry to the root directory.

Definition at line 266 of file fs_helpers.c.

```
                                 {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // check if file already exists
  dir_entry_t existing;
  if (find_file(filename, &existing) >= 0) {
    P_ERRNO = P_EEXIST;
    return -1;
  }

  // start with root directory block (block 1)
  uint16_t current_block = 1;
  int offset = 0;
  dir_entry_t dir_entry;

  while (1) {
    // position at the start of current block of the root directory
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      return -1;
    }

    // reset offset for new block
    offset = 0;

    // search current block for free slot
    while (offset < block_size) {
      if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
        P_ERRNO = P_EREAD;
        return -1;
      }

      // found a free slot
      if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
        // initialize the new entry
        memset(&dir_entry, 0, sizeof(dir_entry));
        strncpy(dir_entry.name, filename, 31);
        dir_entry.size = size;
        dir_entry.firstBlock = first_block;
        dir_entry.type = type;
        dir_entry.perm = perm;
        dir_entry.mtime = time(NULL);

        // write the entry
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset, SEEK_SET) == -1) {
          P_ERRNO = P_ELSEEK;
          return -1;
        }
        if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
          P_ERRNO = P_EWRITE;
          return -1;
        }

        return offset;
      }

      offset += sizeof(dir_entry);
    }

    // current block is full, check if there's a next block
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
      continue;
    }

    // allocate a new block for the root directory
    uint16_t new_block = allocate_block();
    if (new_block == 0) {
      P_ERRNO = P_EFULL;
      return -1;
    }

    // chain the new block
    fat[current_block] = new_block;
```

```
    fat[new_block] = FAT_EOF;

    // initialize new block
    uint8_t* zero_block = calloc(block_size, 1);
    if (!zero_block) {
      P_ERRNO = P_EINVAL;
      return -1;
    }

    // write this new block to the file system
    if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      free(zero_block);
      return -1;
    }
    if (write(fs_fd, zero_block, block_size) != block_size) {
      P_ERRNO = P_EWRITE;
      free(zero_block);
      return -1;
    }

    free(zero_block);

    // initialize the new entry
    memset(&dir_entry, 0, sizeof(dir_entry));
    strncpy(dir_entry.name, filename, 31);
    dir_entry.size = size;
    dir_entry.firstBlock = first_block;
    dir_entry.type = type;
    dir_entry.perm = perm;
    dir_entry.mtime = time(NULL);

    // write the new entry at the start of the new block in the file system
    if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      return -1;
    }
    if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
      P_ERRNO = P_EWRITE;
      return -1;
    }

    return 0;
  }
}
```

4.4.1.2 allocate_block()

```
uint16_t allocate_block ( )
```

Allocates a free block in the FAT.

Returns

block number of the allocated block, or 0 if no free blocks available

Allocates a free block in the FAT.

If no block found, we try compacting the directory.

Definition at line 167 of file fs_helpers.c.

```
167                         {
168   for (int i = 2; i < fat_size / 2; i++) {
169     if (fat[i] == FAT_FREE) {
170       fat[i] = FAT_EOF;
171       return i;
172     }
173   }
174
175   if (compact_directory() == 0) {
176     for (int i = 2; i < fat_size / 2; i++) {
177       if (fat[i] == FAT_FREE) {
178         fat[i] = FAT_EOF;
```

```
179         return i;
180       }
181     }
182   }
183
184   return 0;
185 }
```

4.4.1.3 compact_directory()

```
int compact_directory ( )
```

Compacts the root directory by removing all deleted entries.

Returns

0 on success, -1 on error

Compacts the root directory by removing all deleted entries.

Definition at line 692 of file fs_helpers.c.

```
692                             {
693   if (!is_mounted) {
694     P_ERRNO = P_EFS_NOT_MOUNTED;
695     return -1;
696   }
697
698   // buffer for temp storage of a block
699   uint8_t* dir_buffer = malloc(block_size);
700   if (!dir_buffer) {
701     P_ERRNO = P_EMALLOC;
702     return -1;
703   }
704
705   // start at root directory
706   uint16_t current_block = 1;
707   int dir_entries_count = 0;
708   int deleted_entries_count = 0;
709
710   // calculate number of entries and deleted entries in the root directory
711   while (current_block != FAT_EOF) {
712     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
713       P_ERRNO = P_ELSEEK;
714       free(dir_buffer);
715       return -1;
716     }
717
718     if (read(fs_fd, dir_buffer, block_size) != block_size) {
719       P_ERRNO = P_EREAD;
720       free(dir_buffer);
721       return -1;
722     }
723
724     // count entries and deleted entries in this block
725     for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {
726       dir_entry_t* entry = (dir_entry_t*)(dir_buffer + offset);
727
728       // check if we've reached the end of directory
729       if (entry->name[0] == 0) {
730         break;
731       }
732
733       dir_entries_count++;
734
735       // check if it's a deleted entry
736       if (entry->name[0] == 1) {
737         deleted_entries_count++;
738       }
739     }
740
741     // move onto next block, if there is one
742     if (fat[current_block] != FAT_EOF) {
743       current_block = fat[current_block];
744     } else {
```

```
      break;
    }
  }

  // if no deleted entries, no compaction needed
  if (deleted_entries_count == 0) {
    free(dir_buffer);
    return 0;
  }

  // allocate space for all valid entries
  dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
  if (!all_entries) {
    P_ERRNO = P_EMALLOC;
    free(dir_buffer);
    return -1;
  }

  // read all entries into the buffer, skipping deleted ones
  current_block = 1;
  int valid_entry_idx = 0;

  while (current_block != FAT_EOF) {
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    if (read(fs_fd, dir_buffer, block_size) != block_size) {
      P_ERRNO = P_EREAD;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    // process entries in this block
    for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {
      dir_entry_t* entry = (dir_entry_t*)(dir_buffer + offset);

      // check if we've reached the end of directory
      if (entry->name[0] == 0) {
        break;
      }

      // skip deleted entries
      if (entry->name[0] == 1) {
        continue;
      }

      // copy valid entry to our array
      memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
    }

    // move to the next block
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
    } else {
      break;
    }
  }

  // rewrite the directory with only valid entries
  current_block = 1;
  int entries_per_block = block_size / sizeof(dir_entry_t);
  int blocks_needed = (valid_entry_idx + entries_per_block - 1) / entries_per_block;

  // clean up any excess directory blocks in the FAT chain
  uint16_t next_block = fat[current_block];
  if (blocks_needed == 1) {
    // only need one block, free all others
    while (next_block != FAT_EOF) {
      uint16_t temp = fat[next_block];
      fat[next_block] = FAT_FREE;
      next_block = temp;
    }
    fat[current_block] = FAT_EOF;
  } else {
    // navigate through needed blocks
    int block_count = 1;
    uint16_t prev_block = current_block;

    while (block_count < blocks_needed) {
      if (next_block == FAT_EOF) {
        // need to allocate a new block
        uint16_t new_block = allocate_block();
```

```
        if (new_block == 0) {
          P_ERRNO = P_EFULL;
          free(dir_buffer);
          free(all_entries);
          return -1;
        }
        fat[prev_block] = new_block;
        next_block = new_block;
      }

      prev_block = next_block;
      next_block = fat[next_block];
      block_count++;
    }

    // free any excess blocks
    fat[prev_block] = FAT_EOF;
    while (next_block != FAT_EOF) {
      uint16_t temp = fat[next_block];
      fat[next_block] = FAT_FREE;
      next_block = temp;
    }
  }

  // write the valid entries back to the directory blocks
  current_block = 1;
  int entries_written = 0;

  while (entries_written < valid_entry_idx) {
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    memset(dir_buffer, 0, block_size);

    // copy entries to the buffer
    int entries_in_this_block = 0;
    while (entries_written < valid_entry_idx && entries_in_this_block < entries_per_block) {
      memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
             &all_entries[entries_written],
             sizeof(dir_entry_t));
      entries_written++;
      entries_in_this_block++;
    }

    // write the buffer to the file system
    if (write(fs_fd, dir_buffer, block_size) != block_size) {
      P_ERRNO = P_EINVAL;
      free(dir_buffer);
      free(all_entries);
      return -1;
    }

    // move to the next block if needed
    if (entries_written < valid_entry_idx) {
      current_block = fat[current_block];
    }
  }

  free(dir_buffer);
  free(all_entries);
  return 0;
}
```

4.4.1.4 copy_host_to_pennfat()

```
int copy_host_to_pennfat (
            const char * host_filename,
            const char * pennfat_filename )
```

Copies a file from the host OS to the PennFAT filesystem.

Parameters

host_filename	path to the file on the host OS
pennfat_filename	name to give the file in PennFAT

Returns

0 on success, -1 on error

Copies a file from the host OS to the PennFAT filesystem.

Definition at line 434 of file fs_helpers.c.

```
                                                            {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // open the host file
  int host_fd = open(host_filename, O_RDONLY);
  if (host_fd == -1) {
    P_ERRNO = P_EOPEN;
    return -1;
  }

  // determine file size by seeking to the end and getting position
  off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
  if (host_file_size_in_bytes == -1) {
    P_ERRNO = P_ELSEEK;
    close(host_fd);
    return -1;
  }

  // go back to beginning of file for reading
  if (lseek(host_fd, 0, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    close(host_fd);
    return -1;
  }

  // open the destination file in PennFAT
  int pennfat_fd = k_open(pennfat_filename, F_WRITE);
  if (pennfat_fd < 0) {
    close(host_fd);
    return -1;
  }

  // copy the data into this buffer
  uint8_t* buffer = (uint8_t*)malloc(block_size);
  if (!buffer) {
    P_ERRNO = P_EMALLOC;
    k_close(pennfat_fd);
    close(host_fd);
    return -1;
  }

  uint32_t bytes_remaining = host_file_size_in_bytes;
  ssize_t bytes_read;

  // read from host file
  while (bytes_remaining > 0) {
    // ensure bytes to read never exceeds the block size
    ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
    bytes_read = read(host_fd, buffer, bytes_to_read);

    if (bytes_read <= 0) {
      break;
    }

    // write to pennfat_fd using k_write
    if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
      free(buffer);
      k_close(pennfat_fd);
      close(host_fd);
      return -1;
    }

    bytes_remaining -= bytes_read;
  }
```

```
  // check for read error
  if (bytes_read < 0) {
    P_ERRNO = P_EREAD;
    free(buffer);
    k_close(pennfat_fd);
    close(host_fd);
    return -1;
  }

  // otherwise, cleanup and return success
  free(buffer);
  k_close(pennfat_fd);
  close(host_fd);
  return 0;
}
```

4.4.1.5 copy_pennfat_to_host()

```
int copy_pennfat_to_host (
            const char * pennfat_filename,
            const char * host_filename )
```

Copies a file from the PennFAT filesystem to the host OS.

Parameters

pennfat_filename	name of the file in PennFAT
host_filename	path to save the file on the host OS

Returns

0 on success, -1 on error

Copies a file from the PennFAT filesystem to the host OS.

Definition at line 522 of file fs_helpers.c.

```
                                                                {
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // open the PennFAT file
  int pennfat_fd = k_open(pennfat_filename, F_READ);
  if (pennfat_fd < 0) {
    return -1;
  }

  // get the pennfat file size
  off_t pennfat_file_size_in_bytes = k_lseek(pennfat_fd, 0, SEEK_END);
  if (pennfat_file_size_in_bytes == -1) {
    k_close(pennfat_fd);
    return -1;
  }

  // go back to beginning of file for reading
  if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
    k_close(pennfat_fd);
    return -1;
  }

  // open the host file
  int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (host_fd == -1) {
    P_ERRNO = P_EOPEN;
    k_close(pennfat_fd);
```

```
    return -1;
  }

  // allocate buffer for data transfer
  char* buffer = (char*)malloc(block_size);
  if (!buffer) {
    P_ERRNO = P_EMALLOC;
    k_close(pennfat_fd);
    close(host_fd);
    return -1;
  }

  uint32_t bytes_remaining = pennfat_file_size_in_bytes;
  ssize_t bytes_read;

  // read from PennFAT file and write to host file
  while (bytes_remaining > 0) {
    // ensure bytes to read never exceeds the block size
    size_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
    bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);

    if (bytes_read <= 0) {
      break;
    }

    if (write(host_fd, buffer, bytes_read) != bytes_read) {
      P_ERRNO = P_EINVAL;
      free(buffer);
      close(host_fd);
      k_close(pennfat_fd);
      return -1;
    }

    bytes_remaining -= bytes_read;
  }

  // check for read error
  if (bytes_read < 0) {
    P_ERRNO = P_EREAD;
    free(buffer);
    close(host_fd);
    k_close(pennfat_fd);
    return -1;
  }

  // otherwise, cleanup and return success
  free(buffer);
  close(host_fd);
  k_close(pennfat_fd);
  return 0;
}
```

4.4.1.6 copy_source_to_dest()

```
int copy_source_to_dest (
        const char * source_filename,
        const char * dest_filename )
```

Copies a file from a source file to a destination file.

Parameters

source_filename	name of the source filename
dest_filename	name of the destination filename

Returns

0 on success, -1 on error

Copies a file from a source file to a destination file.

Definition at line 608 of file fs_helpers.c.

```
{
  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // open the source file
  int source_fd = k_open(source_filename, F_READ);
  if (source_fd < 0) {
    return -1;
  }

  // get the source file size
  off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
  if (source_file_size_in_bytes == -1) {
    k_close(source_fd);
    return -1;
  }

  // move to the beginning of the source file for reading
  if (k_lseek(source_fd, 0, SEEK_SET) < 0) {
    k_close(source_fd);
    return -1;
  }

  // open the destination file
  int dest_fd = k_open(dest_filename, F_WRITE);
  if (dest_fd < 0) {
    k_close(source_fd);
    return -1;
  }

  // read from source to destination
  char* buffer = (char*)malloc(block_size);
  if (!buffer) {
    P_ERRNO = P_EMALLOC;
    k_close(source_fd);
    k_close(dest_fd);
    return -1;
  }

  uint32_t bytes_remaining = source_file_size_in_bytes;
  ssize_t bytes_read;

  while (bytes_remaining > 0) {
    // make sure the bytes to read doesn't exceed block size
    ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
    bytes_read = k_read(source_fd, buffer, bytes_to_read);

    if (bytes_read <= 0) {
      break;
    }

    if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
      free(buffer);
      k_close(source_fd);
      k_close(dest_fd);
      return -1;
    }
  }

  // check for read error
  if (bytes_read < 0) {
    free(buffer);
    k_close(source_fd);
    k_close(dest_fd);
    return -1;
  }

  // otherwise, cleanup and return success
  free(buffer);
  k_close(source_fd);
  k_close(dest_fd);
  return 0;
}
```

4.4.1.7 decrement_fd_ref_count()

```
int decrement_fd_ref_count (
        int fd )
```

Decrements the reference count of a file descriptor.

Parameters

fd	file descriptor to decrement

Returns

new reference count, or -1 on error

If reference count reaches 0, flush field values.

Definition at line 107 of file fs_helpers.c.

```
107                                    {
108   if (fd < 0 || fd >= MAX_FDS) {
109     P_ERRNO = P_EBADF;
110     return -1;
111   }
112 
113   if (!fd_table[fd].in_use) {
114     P_ERRNO = P_EBADF;
115     return -1;
116   }
117 
118   fd_table[fd].ref_count--;
119   if (fd_table[fd].ref_count == 0) {
120     fd_table[fd].in_use = 0;
121     memset(fd_table[fd].filename, 0, sizeof(fd_table[fd].filename));
122     fd_table[fd].size = 0;
123     fd_table[fd].first_block = 0;
124     fd_table[fd].position = 0;
125     fd_table[fd].mode = 0;
126   }
127   return fd_table[fd].ref_count;
128 }
```

4.4.1.8 find_file()

```
int find_file (
            const char * filename,
            dir_entry_t * entry )
```

Searches for a file in the root directory.

Parameters

filename	name of the file to find
entry	pointer to store the directory entry if found

Returns

offset of the entry in the directory if found, -1 if not found

Retrieves the file's absolute offset in the filesystem.

Definition at line 192 of file fs_helpers.c.

```
192                                                         {
193   if (!is_mounted) {
```

```
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // Start with root directory block (block 1)
  uint16_t current_block = 1;
  int offset_in_block = 0;
  int absolute_offset = 0;
  dir_entry_t dir_entry;

  while (1) {
    // Position at the start of current block
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      return -1;
    }

    // reset offset for new block
    offset_in_block = 0;

    // calculate the absolute offset
    absolute_offset = fat_size + (current_block - 1) * block_size;

    // search current block
    while (offset_in_block < block_size) {
      if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
        P_ERRNO = P_EREAD;
        return -1;
      }

      // check if we've reached the end of directory
      if (dir_entry.name[0] == 0) {
        break;
      }

      // check if this is a deleted entry
      if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
        offset_in_block += sizeof(dir_entry);
        absolute_offset += sizeof(dir_entry);
        continue;
      }

      // check if we found the file
      if (strcmp(dir_entry.name, filename) == 0) {
        if (entry) {
          memcpy(entry, &dir_entry, sizeof(dir_entry));
        }
        return absolute_offset; // return the absolute file offset
      }

      offset_in_block += sizeof(dir_entry);
      absolute_offset += sizeof(dir_entry);
    }

    // if we've reached the end of the current block, check if there's a next block
    if (fat[current_block] != FAT_EOF) {
      current_block = fat[current_block];
      continue;
    }

    // no more blocks to search
    break;
  }

  // file not found
  P_ERRNO = P_ENOENT;
  return -1;
}
```

4.4.1.9 get_free_fd()

```
int get_free_fd (
        fd_entry_t * fd_table )
```

Finds the first available file descriptor in the table.

Parameters

fd_table	pointer to the file descriptor table to search

Returns

index of the first free file descriptor, or -1 if none available

Finds the first available file descriptor in the table.

Definition at line 77 of file fs_helpers.c.

```
77                                          {
78   for (int i = 3; i < MAX_FDS; i++) {
79     if (!fd_table[i].in_use) {
80       return i;
81     }
82   }
83   return -1;
84 }
```

4.4.1.10 has_executable_permission()

```
int has_executable_permission (
            int fd )
```

Checks if a file has executable permissions in the PennFAT filesystem.

Parameters

fd	The fd of the file to check.

Returns

1 if the file has executable permissions, 0 if it doesn't, -1 if an error occurred.

Checks if a file has executable permissions in the PennFAT filesystem.

Definition at line 133 of file fs_helpers.c.

```
133                                         {
134   // check if fs is mounted
135   if (!is_mounted) {
136     P_ERRNO = P_EFS_NOT_MOUNTED;
137     return -1;
138   }
139
140   // validate fd argument
141   if (fd < 0 || fd >= MAX_FDS) {
142     P_ERRNO = P_EINVAL;
143     return -1;
144   }
145
146
147   // determine whether the file exists
148   dir_entry_t entry;
149   int entry_offset = find_file(fd_table[fd].filename, &entry);
150   if (entry_offset < 0) {
151     return -1;
152   }
153
154   // if it exists, get its permission
```

```
155   if (entry.perm & PERM_EXEC) {
156     return 1;
157   }
158
159   return 0;
160 }
```

4.4.1.11 increment_fd_ref_count()

```
int increment_fd_ref_count (
            int fd )
```

Increments the reference count of a file descriptor.

Parameters

fd	file descriptor to increment

Returns

new reference count, or -1 on error

Definition at line 89 of file fs_helpers.c.

```
89                                   {
90   if (fd < 0 || fd >= MAX_FDS) {
91     P_ERRNO = P_EBADF;
92     return -1;
93   }
94   if (!fd_table[fd].in_use) {
95     P_ERRNO = P_EBADF;
96     return -1;
97   }
98   fd_table[fd].ref_count++;
99   return fd_table[fd].ref_count;
100 }
```

4.4.1.12 init_fd_table()

```
void init_fd_table (
            fd_entry_t * fd_table )
```

Initializes all entries in the file descriptor table to not in use.

Parameters

fd_table	pointer to the file descriptor table to initialize

Initializes all entries in the file descriptor table to not in use.

Definition at line 43 of file fs_helpers.c.

```
43                                          {
44   // STDIN (fd 0)
45   fd_table[0].in_use = 1;
```

```
  fd_table[0].ref_count = 1;
  strncpy(fd_table[0].filename, "<stdin>", 31);
  fd_table[0].mode = F_READ;

  // STDOUT (fd 1)
  fd_table[1].in_use = 1;
  strncpy(fd_table[1].filename, "<stdout>", 31);
  fd_table[1].mode = F_WRITE; // write-only
  fd_table[1].ref_count = 1;

  // STDERR (fd 2)
  fd_table[2].in_use = 1;
  strncpy(fd_table[2].filename, "<stderr>", 31);
  fd_table[2].mode = F_WRITE; // write-only
  fd_table[2].ref_count = 1;

  // other file descriptors (fd 3 and above)
  for (int i = 3; i < MAX_FDS; i++) {
    fd_table[i].in_use = 0;
    fd_table[i].ref_count = 0;
    memset(fd_table[i].filename, 0, sizeof(fd_table[i].filename));
    fd_table[i].size = 0;
    fd_table[i].first_block = 0;
    fd_table[i].position = 0;
    fd_table[i].mode = 0;
  }
}
```

4.4.1.13 mark_entry_as_deleted()

```
int mark_entry_as_deleted (
            dir_entry_t * entry,
            int absolute_offset )
```

Marks a file entry as deleted and frees its blocks in the FAT.

This function takes a directory entry and its offset in the directory, marks it as deleted in the directory, and frees all blocks in its FAT chain.

Parameters

entry	the entry struct of the file to mark as deleted.
offset	the offset of the entry in the directory

Returns

0 on success, -1 on error

Marks a file entry as deleted and frees its blocks in the FAT.

Definition at line 396 of file fs_helpers.c.

```
396                                                             {
397   if (!is_mounted || entry == NULL || absolute_offset < 0) {
398     P_ERRNO = P_EINVAL;
399     return -1;
400   }
401
402   // free the blocks
403   uint16_t current_block = entry->firstBlock;
404   while (current_block != FAT_FREE && current_block != FAT_EOF) {
405     uint16_t next_block = fat[current_block];
406     fat[current_block] = FAT_FREE;
407     current_block = next_block;
408   }
409
410   // mark the entry as deleted in the root directory
```

```
  dir_entry_t deleted_entry = *entry;
  deleted_entry.name[0] = 1;
  if (lseek(fs_fd, absolute_offset, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    return -1;
  }
  if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) != sizeof(deleted_entry)) {
    P_ERRNO = P_EINVAL;
    return -1;
  }

  // mark the passed entry as deleted
  entry->name[0] = 1;
  return 0;
}
```

4.4.2 Variable Documentation

4.4.2.1 block_size

```
int block_size  [extern]
```

Definition at line 28 of file fs_helpers.c.

4.4.2.2 fat

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

Definition at line 31 of file fs_helpers.c.

4.4.2.3 fat_size

```
int fat_size  [extern]
```

Definition at line 30 of file fs_helpers.c.

4.4.2.4 fd_table

```
fd_entry_t fd_table[100]  [extern]
```

Definition at line 34 of file fs_helpers.c.

4.4.2.5 fs_fd

```
int fs_fd [extern]
```

Definition at line 27 of file fs_helpers.c.

4.4.2.6 is_mounted

```
bool is_mounted [extern]
```

Definition at line 32 of file fs_helpers.c.

4.4.2.7 MAX_FDS

```
int MAX_FDS [extern]
```

Definition at line 33 of file fs_helpers.c.

4.4.2.8 num_fat_blocks

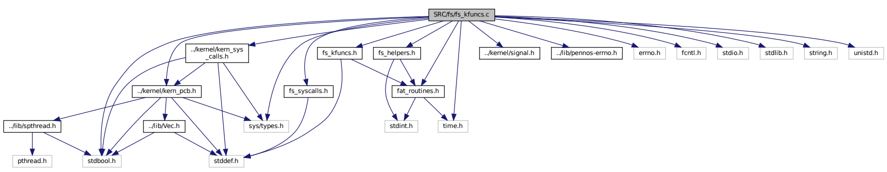
```
int num_fat_blocks [extern]
```

Definition at line 29 of file fs_helpers.c.

4.5 SRC/fs/fs_kfuncs.c File Reference

```
#include "fs_kfuncs.h"
#include "../kernel/kern_pcb.h"
#include "../kernel/kern_sys_calls.h"
#include "../kernel/signal.h"
#include "../lib/pennos-errno.h"
#include "fat_routines.h"
#include "fs_helpers.h"
#include "fs_syscalls.h"
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
```

Include dependency graph for fs_kfuncs.c:



Functions

- int k_open (const char ∗fname, int mode)

  Kernel-level call to open a file.
- int k_read (int fd, char ∗buf, int n)

  Kernel-level call to read a file.
- int k_write (int fd, const char ∗str, int n)

  Kernel-level call to write to a file.
- int k_close (int fd)

  Kernel-level call to close a file.
- int k_unlink (const char ∗fname)

  Kernel-level call to remove a file.
- int k_lseek (int fd, int offset, int whence)

  Kernel-level call to re-position a file offset.
- void format_file_info (dir_entry_t ∗entry, char ∗buffer)
- int k_ls (const char ∗filename)

  Kernel-level call to list files.

Variables

- pcb_t ∗ current_running_pcb
- pid_t current_fg_pid

4.5.1 Function Documentation

4.5.1.1 format_file_info()

```
void format_file_info (
            dir_entry_t * entry,
            char * buffer )
```

Definition at line 666 of file fs_kfuncs.c.

```
                                                    {
  // convert permissions to string
  char perms[4] = "---";
  if (entry->perm & PERM_READ)
    perms[0] = 'r';
  if (entry->perm & PERM_WRITE)
    perms[1] = 'w';
  if (entry->perm & PERM_READ_EXEC & ~PERM_READ)
    perms[2] = 'x';

  // convert time to string
  char time_str[20];
  struct tm* tm = localtime(&entry->mtime);
  strftime(time_str, sizeof(time_str), "%b %d %H:%M", tm);

  // format the output string
  snprintf(buffer, 256, "%4d %s %6d %s %s\n", entry->firstBlock, perms,
           entry->size, time_str, entry->name);
}
```

4.5.1.2 k_close()

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

Kernel-level call to close a file.

Closes an open file.

Definition at line 528 of file fs_kfuncs.c.

```
528                 {
529   /*if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
530       P_ERRNO = P_EINVAL;
531       return -1;
532   }*/
533
534   // validate the file descriptor
535   if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
536     P_ERRNO = P_EBADF;
537     return -1;
538   }
539
540   // ensure any pending changes are written to disk
541   // update the directory entry with the current file size
542   dir_entry_t entry;
543   int file_offset = find_file(fd_table[fd].filename, &entry);
544
545   if (file_offset >= 0) {
546     // update file size if it changed
547     if (entry.size != fd_table[fd].size) {
548       entry.size = fd_table[fd].size;
549       entry.mtime = time(NULL);
550
551       if (lseek(fs_fd, file_offset, SEEK_SET) != -1) {
552         write(fs_fd, &entry, sizeof(entry));
553       }
554     }
555   }
556
557   // decrement the reference count
558   decrement_fd_ref_count(fd);
559
560   return 0;
561 }
```

4.5.1.3 k_ls()

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

Kernel-level call to list files.

Lists files or file information.

Definition at line 689 of file fs_kfuncs.c.

```
689                                 {
690   if (!is_mounted) {
691     P_ERRNO = P_EFS_NOT_MOUNTED;
692     return -1;
693   }
694
695   // start with root directory block
696   uint16_t current_block = 1;
697   dir_entry_t entry;
698   uint32_t offset_in_block = 0;
699
700   // if filename is null, list all files in the current directory
701   if (filename == NULL) {
702     while (current_block != FAT_EOF) {
703       // calculate absolute offset in filesystem
704       off_t abs_offset =
```

```
          fat_size + (current_block - 1) * block_size + offset_in_block;

      // read directory entry
      if (lseek(fs_fd, abs_offset, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        return -1;
      }
      if (read(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
        P_ERRNO = P_EREAD;
        return -1;
      }

      // check for end of directory
      if (entry.name[0] == 0)
        break;

      // skip deleted entries
      if (entry.name[0] == 1 || entry.name[0] == 2) {
        offset_in_block += sizeof(entry);
        // check if we need to move to next block
        if (offset_in_block + sizeof(entry) > block_size) {
          current_block = fat[current_block];
          offset_in_block = 0;
        }
        continue;
      }

      // format and write entry information
      char output_buffer[256];
      format_file_info(&entry, output_buffer);
      if (k_write(STDOUT_FILENO, output_buffer, strlen(output_buffer)) < 0) {
        return -1;
      }

      // move to next entry
      offset_in_block += sizeof(entry);
      // check if we need to move to next block
      if (offset_in_block + sizeof(entry) > block_size) {
        current_block = fat[current_block];
        offset_in_block = 0;
      }
    }
  } else {
    // find and display specific file
    int file_offset = find_file(filename, &entry);
    if (file_offset < 0) {
      P_ERRNO = P_ENOENT;
      return -1;
    }

    char output_buffer[256];
    format_file_info(&entry, output_buffer);
    if (k_write(STDOUT_FILENO, output_buffer, strlen(output_buffer)) < 0) {
      return -1;
    }
  }

  return 0;
}
```

4.5.1.4 k_lseek()

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

Kernel-level call to re-position a file offset.

Repositions the file offset of an open file.

Definition at line 622 of file fs_kfuncs.c.

```
622                                              {
623   // standard file descriptors don't support lseek
624   if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
```

```
    P_ERRNO = P_EINVAL;
    return -1;
  }

  // validate the file descriptor
  if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
    P_ERRNO = P_EBADF;
    return -1;
  }

  // calculate new position based on whence
  int32_t new_position;

  switch (whence) {
    case SEEK_SET:
      new_position = offset;
      break;
    case SEEK_CUR:
      new_position = fd_table[fd].position + offset;
      break;
    case SEEK_END:
      new_position = fd_table[fd].size + offset;
      break;
    default:
      P_ERRNO = P_EINVAL;
      return -1;
  }

  // check if new position is valid
  if (new_position < 0) {
    P_ERRNO = P_EINVAL;
    return -1;
  }

  // update file position
  fd_table[fd].position = new_position;

  return new_position;
}
```

4.5.1.5 k_open()

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

Kernel-level call to open a file.

Opens a file with the specified mode.

Definition at line 33 of file fs_kfuncs.c.

```
33                                        {
34   // validate arguments
35   if (fname == NULL || *fname == '\0') {
36     P_ERRNO = P_EINVAL;
37     return -1;
38   }
39   if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
40     P_ERRNO = P_EINVAL;
41     return -1;
42   }
43
44   // check if the file system is mounted
45   if (!is_mounted) {
46     P_ERRNO = P_EFS_NOT_MOUNTED;
47     return -1;
48   }
49
50   // get a free file descriptor
51   int fd = get_free_fd(fd_table);
52   if (fd < 0) {
53     P_ERRNO = P_EFULL;  // no free file descriptors
54     return -1;
55   }
56
57   // check if the file exists
```

```c
dir_entry_t entry;
int file_offset = find_file(fname, &entry);

// file exists
if (file_offset >= 0) {
  // check if the file is already open in write mode by another descriptor
  if ((mode & (F_WRITE | F_APPEND)) != 0) {
    for (int i = 0; i < MAX_FDS; i++) {
      if (i != fd && fd_table[i].in_use &&
          strcmp(fd_table[i].filename, fname) == 0 &&
          (fd_table[i].mode & (F_WRITE | F_APPEND)) != 0) {
        P_ERRNO = P_EBUSY;  // file is already open for writing
        return -1;
      }
    }
  }

  // fill in the file descriptor entry
  fd_table[fd].in_use = 1;
  fd_table[fd].ref_count++;
  strncpy(fd_table[fd].filename, fname, 31);
  fd_table[fd].filename[31] = '\0';
  fd_table[fd].size = entry.size;
  fd_table[fd].first_block = entry.firstBlock;
  fd_table[fd].mode = mode;

  // set the initial position
  if (mode & F_APPEND) {
    fd_table[fd].position = entry.size;
  } else {
    fd_table[fd].position = 0;
  }

  // if mode includes F_WRITE and not F_APPEND, truncate the file
  if ((mode & F_WRITE) && !(mode & F_APPEND)) {
    // free all blocks except the first one
    uint16_t block = entry.firstBlock;
    uint16_t next_block;

    if (block != 0 && block != FAT_EOF) {
      next_block = fat[block];
      fat[block] = FAT_EOF;  // terminate the chain at the first block
      block = next_block;

      // free the rest of the chain
      while (block != 0 && block != FAT_EOF) {
        next_block = fat[block];
        fat[block] = FAT_FREE;
        block = next_block;
      }
    }

    // update file size to 0
    fd_table[fd].size = 0;
    entry.size = 0;
    entry.mtime = time(NULL);

    // update the file system with the truncated file
    if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      return -1;
    }
    if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
      P_ERRNO = P_EWRITE;
      return -1;
    }
  }
} else {
  // file doesn't exist

  // we can only create it if we are reading the file
  if (!(mode & F_WRITE)) {
    P_ERRNO = P_ENOENT;
    return -1;
  }

  // allocate the first block
  uint16_t first_block = allocate_block();
  if (first_block == 0) {
    P_ERRNO = P_EFULL;
    return -1;
  }

  // create a new file entry
  if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
      -1) {
    // error code already set by add_file_entry
```

```
      fat[first_block] = FAT_FREE;
      return -1;
    }

    // fill in the file descriptor entry
    fd_table[fd].in_use = 1;
    fd_table[fd].ref_count++;
    strncpy(fd_table[fd].filename, fname, 31);
    fd_table[fd].filename[31] = '\0';
    fd_table[fd].size = 0;
    fd_table[fd].first_block = first_block;
    fd_table[fd].position = 0;
    fd_table[fd].mode = mode;
  }

  return fd;
}
```

4.5.1.6 k_read()

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

Kernel-level call to read a file.

Reads data from an open file.

Definition at line 166 of file fs_kfuncs.c.

```
166                                  {
167     // handle terminal control (if doesn't control, send a STOP signal)
168     if (fd == STDIN_FILENO && current_running_pcb != NULL) {
169         if (current_running_pcb->pid != current_fg_pid) {
170             s_kill(current_running_pcb->pid, P_SIGSTOP);
171         }
172     }
173
174   // handle standard input
175   if (fd == STDIN_FILENO) {
176     return read(STDIN_FILENO, buf, n);
177   }
178
179   // validate inputs
180   if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
181     P_ERRNO = P_EBADF;
182     return -1;
183   }
184   if (buf == NULL || n < 0) {
185     P_ERRNO = P_EINVAL;
186     return -1;
187   }
188   if (n == 0) {
189     return 0;
190   }
191
192   // check if we're at EOF already
193   if (fd_table[fd].position >= fd_table[fd].size) {
194     return 0;
195   }
196
197   // determine how many bytes we can actually read
198   uint32_t bytes_to_read = n;
199   if (fd_table[fd].position + bytes_to_read > fd_table[fd].size) {
200     bytes_to_read = fd_table[fd].size - fd_table[fd].position;
201   }
202
203   // find the block containing the current position
204   uint16_t current_block = fd_table[fd].first_block;
205   uint32_t block_index = fd_table[fd].position / block_size;
206   uint32_t block_offset = fd_table[fd].position % block_size;
207
208   // navigate to the correct block in the chain
209   for (uint32_t i = 0; i < block_index; i++) {
210     if (current_block == 0 || current_block == FAT_EOF) {
```

```
      // unexpected end of chain
      P_ERRNO = P_EINVAL;
      return -1;
    }
    current_block = fat[current_block];
  }

  // now we're at the right block, start reading
  uint32_t bytes_read = 0;

  while (bytes_read < bytes_to_read) {
    // how much data can we read from the current block
    uint32_t bytes_left_in_block = block_size - block_offset;
    uint32_t bytes_to_read_now =
        (bytes_to_read - bytes_read) < bytes_left_in_block
            ? (bytes_to_read - bytes_read)
            : bytes_left_in_block;

    // seek to the right position in the file
    if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
              SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
      if (bytes_read > 0) {
        fd_table[fd].position += bytes_read;
        return bytes_read;
      }
      return -1;
    }

    // read the data from the file
    ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
    if (read_result <= 0) {
      P_ERRNO = P_EREAD;
      // if we already read some data, return that count
      if (bytes_read > 0) {
        fd_table[fd].position += bytes_read;
        return bytes_read;
      }
      return -1;
    }

    bytes_read += read_result;
    block_offset += read_result;

    // if we've read all data from this block and still have more to read, go to
    // the next block
    if (block_offset == block_size && bytes_read < bytes_to_read) {
      if (current_block == FAT_EOF) {
        // unexpected end of chain
        break;
      }
      current_block = fat[current_block];
      block_offset = 0;
    }

    // if we read less than expected, we might have hit EOF
    if (read_result < bytes_to_read_now) {
      break;
    }
  }

  // update file position
  fd_table[fd].position += bytes_read;

  return bytes_read;
}
```

4.5.1.7 k_unlink()

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

Kernel-level call to remove a file.

Removes a file from the file system.

Definition at line 566 of file fs_kfuncs.c.

```
{
  if (fname == NULL || *fname == '\0') {
    P_ERRNO = P_EINVAL;
    return -1;
  }

  if (!is_mounted) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // check if file is currently open by any process
  for (int i = 0; i < MAX_FDS; i++) {
    if (fd_table[i].in_use && strcmp(fd_table[i].filename, fname) == 0) {
      P_ERRNO = P_EBUSY;
      return -1;
    }
  }

  // find the file in directory
  dir_entry_t entry;
  int file_offset = find_file(fname, &entry);
  if (file_offset < 0) {
    P_ERRNO = P_ENOENT;
    return -1;
  }

  // mark the directory entry as deleted (set first byte to 1)
  entry.name[0] = 1;

  // write the modified directory entry back
  if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    return -1;
  }
  if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
    P_ERRNO = P_EWRITE;
    return -1;
  }

  // free all blocks in the file chain
  uint16_t current_block = entry.firstBlock;
  uint16_t next_block;

  while (current_block != FAT_FREE && current_block != FAT_EOF) {
    next_block = fat[current_block];
    fat[current_block] = FAT_FREE;
    current_block = next_block;
  }

  return 0;
}
```

4.5.1.8 k_write()

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

Kernel-level call to write to a file.

Writes data to an open file.

Definition at line 281 of file fs_kfuncs.c.

```
281                                   {
282   // handle standard output and error
283   if (fd == STDOUT_FILENO) {
284     return write(STDOUT_FILENO, str, n);
285   }
286   if (fd == STDERR_FILENO) {
287     return write(STDERR_FILENO, str, n);
288   }
289
290   // validate inputs
```

```
    if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
      P_ERRNO = P_EBADF;
      return -1;
    }
    if (str == NULL || n < 0) {
      P_ERRNO = P_EINVAL;
      return -1;
    }
    if (n == 0) {
      return 0;
    }

    // check if filesystem is mounted and FAT is valid
    if (!is_mounted || fat == NULL) {
      P_ERRNO = P_EFS_NOT_MOUNTED;
      return -1;
    }

    // get file information
    uint16_t current_block = fd_table[fd].first_block;
    uint32_t current_position = fd_table[fd].position;

    // create a local buffer for block data
    char* block_buffer = (char*)malloc(block_size);
    if (block_buffer == NULL) {
      P_ERRNO = P_EMALLOC;
      return -1;
    }

    // calculate initial block position
    uint32_t block_index = current_position / block_size;
    uint32_t block_offset = current_position % block_size;

    // if the file doesn't have a first block yet, allocate one
    if (current_block == 0) {
      current_block = allocate_block();
      if (current_block == 0) {
        P_ERRNO = P_EFULL;
        free(block_buffer);
        return -1;
      }
      fd_table[fd].first_block = current_block;
    }

    // navigate to the appropriate block
    uint16_t prev_block = 0;
    for (uint32_t i = 0; i < block_index; i++) {
      if (current_block == 0 || current_block == FAT_EOF ||
          current_block >= fat_size / 2) {
        // reached the end of chain prematurely, need to allocate a new block
        uint16_t new_block = allocate_block();
        if (new_block == 0) {
          P_ERRNO = P_EFULL;
          free(block_buffer);
          return -1;
        }

        // update the chain
        if (prev_block != 0 && prev_block < fat_size / 2) {
          fat[prev_block] = new_block;
        } else {
          // if there's no previous block, this must be the first one
          fd_table[fd].first_block = new_block;
        }

        current_block = new_block;
      }

      prev_block = current_block;

      // validate the block number before accessing FAT
      if (current_block >= fat_size / 2) {
        P_ERRNO = P_EINVAL;
        free(block_buffer);
        return -1;
      }

      current_block = fat[current_block];
    }

    // if we ended up without a valid block, go back to the last valid one
    if (current_block == 0 || current_block == FAT_EOF ||
        current_block >= fat_size / 2) {
      if (prev_block != 0 && prev_block < fat_size / 2) {
        uint16_t new_block = allocate_block();
        if (new_block == 0) {
          P_ERRNO = P_EFULL;
```

```
        free(block_buffer);
        return -1;
      }

      fat[prev_block] = new_block;
      current_block = new_block;
    } else {
      P_ERRNO = P_EINVAL;
      free(block_buffer);
      return -1;
    }
  }

  // start writing data
  uint32_t bytes_written = 0;

  while (bytes_written < n) {
    // validate current block
    if (current_block == 0 || current_block == FAT_EOF ||
        current_block >= fat_size / 2) {
      P_ERRNO = P_EINVAL;
      break;
    }

    // how much can we write to this block
    uint32_t space_in_block = block_size - block_offset;
    uint32_t bytes_to_write = (n - bytes_written) < space_in_block
                                  ? (n - bytes_written)
                                  : space_in_block;

    // position in filesystem
    off_t block_position = fat_size + (current_block - 1) * block_size;

    // if we're not writing a full block or not starting at the beginning, we
    // need to read-modify-write
    if (bytes_to_write < block_size || block_offset > 0) {
      // read the current block
      if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        break;
      }

      // read the current block data
      ssize_t read_result = read(fs_fd, block_buffer, block_size);
      if (read_result < 0) {
        P_ERRNO = P_EREAD;
        break;
      }

      // copy the new data into the block buffer
      memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);

      // seek back to write the modified block
      if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        break;
      }

      // write the full block back
      ssize_t write_result = write(fs_fd, block_buffer, block_size);
      if (write_result != block_size) {
        P_ERRNO = P_EWRITE;
        // we might have a partial write, but that's hard to handle correctly
        break;
      }
    } else {
      // we're writing a full block from the beginning
      if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        break;
      }

      ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
      if (write_result != bytes_to_write) {
        P_ERRNO = P_EWRITE;
        break;
      }
    }

    // update counters
    bytes_written += bytes_to_write;
    block_offset = (block_offset + bytes_to_write) % block_size;

    // if we've filled this block and still have more to write, go to the next
    // block
    if (block_offset == 0 && bytes_written < n) {
      // validate current block before accessing FAT
```

```
      if (current_block >= fat_size / 2) {
        P_ERRNO = P_EINVAL;
        break;
      }

      // check if there's a next block
      if (fat[current_block] == FAT_EOF) {
        // allocate a new block
        uint16_t new_block = allocate_block();
        if (new_block == 0) {
          P_ERRNO = P_EFULL;
          break;
        }

        // Update the FAT safely
        if (current_block < fat_size / 2) {
          fat[current_block] = new_block;
        } else {
          P_ERRNO = P_EINVAL;
          break;
        }

        current_block = new_block;
      } else {
        current_block = fat[current_block];
      }
    }
  }

  // free the block buffer
  free(block_buffer);

  // update file position
  fd_table[fd].position += bytes_written;

  // update file size if needed
  if (fd_table[fd].position > fd_table[fd].size) {
    fd_table[fd].size = fd_table[fd].position;

    // update the directory entry
    dir_entry_t entry;
    int dir_offset = find_file(fd_table[fd].filename, &entry);
    if (dir_offset >= 0) {
      entry.size = fd_table[fd].size;
      entry.mtime = time(NULL);

      if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        return -1;
      }
      if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
        P_ERRNO = P_EWRITE;
        return -1;
      }
    }
  }

  return bytes_written;
}
```

4.5.2 Variable Documentation

4.5.2.1 current_fg_pid

`pid_t current_fg_pid [extern]`

Definition at line 31 of file kern_sys_calls.c.

4.5.2.2 current_running_pcb

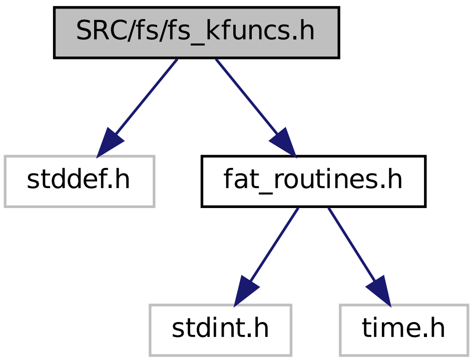
```
pcb_t* current_running_pcb [extern]
```

Definition at line 38 of file scheduler.c.

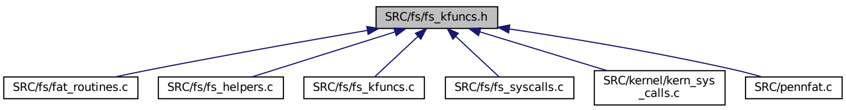
4.6 SRC/fs/fs_kfuncs.h File Reference

```
#include <stddef.h>
#include "fat_routines.h"
```

Include dependency graph for fs_kfuncs.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define SEEK_SET 0
- #define SEEK_CUR 1
- #define SEEK_END 2

Functions

- int k_open (const char ∗fname, int mode)
  Opens a file with the specified mode.
- int k_read (int fd, char ∗buf, int n)
  Reads data from an open file.
- int k_write (int fd, const char ∗str, int n)
  Writes data to an open file.
- int k_close (int fd)
  Closes an open file.
- int k_unlink (const char ∗fname)
  Removes a file from the file system.
- int k_lseek (int fd, int offset, int whence)
  Repositions the file offset of an open file.
- int k_ls (const char ∗filename)
  Lists files or file information.

4.6.1 Macro Definition Documentation

4.6.1.1 SEEK_CUR

```
#define SEEK_CUR 1
```

Definition at line 17 of file fs_kfuncs.h.

4.6.1.2 SEEK_END

```
#define SEEK_END 2
```

Definition at line 18 of file fs_kfuncs.h.

4.6.1.3 SEEK_SET

```
#define SEEK_SET 0
```

Definition at line 16 of file fs_kfuncs.h.

4.6.2 Function Documentation

4.6.2.1 k_close()

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

Closes an open file.

This is a kernel-level function that closes an open file and releases the associated file descriptor. Any unsaved changes are flushed to disk.

Parameters

fd	File descriptor of the open file.

Returns

0 on success, -1 on error with P_ERRNO set. Possible error codes:

- P_EBADF: Invalid file descriptor.

Closes an open file.

Definition at line 528 of file fs_kfuncs.c.

```
528                     {
529   /*if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
530       P_ERRNO = P_EINVAL;
531       return -1;
532   }*/
533
534   // validate the file descriptor
535   if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
536     P_ERRNO = P_EBADF;
537     return -1;
538   }
539
540   // ensure any pending changes are written to disk
541   // update the directory entry with the current file size
542   dir_entry_t entry;
543   int file_offset = find_file(fd_table[fd].filename, &entry);
544
545   if (file_offset >= 0) {
546     // update file size if it changed
547     if (entry.size != fd_table[fd].size) {
548       entry.size = fd_table[fd].size;
549       entry.mtime = time(NULL);
550
551       if (lseek(fs_fd, file_offset, SEEK_SET) != -1) {
552         write(fs_fd, &entry, sizeof(entry));
553       }
554     }
555   }
556
557   // decrement the reference count
558   decrement_fd_ref_count(fd);
559
560   return 0;
561 }
```

4.6.2.2 k_ls()

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

Lists files or file information.

This is a kernel-level function that provides directory listing functionality. If filename is NULL or refers to a directory, it lists all files in that directory. If filename refers to a specific file, it displays detailed information about that file.

Parameters

filename	The name of the file or directory to list, or NULL for the current directory.

Returns

0 on success, -1 on error with P_ERRNO set. Possible error codes:

- P_ENOENT: Specified file or directory doesn't exist.

Lists files or file information.

Definition at line 689 of file fs_kfuncs.c.

```
689                                  {
690   if (!is_mounted) {
691     P_ERRNO = P_EFS_NOT_MOUNTED;
692     return -1;
693   }
694
695   // start with root directory block
696   uint16_t current_block = 1;
697   dir_entry_t entry;
698   uint32_t offset_in_block = 0;
699
700   // if filename is null, list all files in the current directory
701   if (filename == NULL) {
702     while (current_block != FAT_EOF) {
703       // calculate absolute offset in filesystem
704       off_t abs_offset =
705           fat_size + (current_block - 1) * block_size + offset_in_block;
706
707       // read directory entry
708       if (lseek(fs_fd, abs_offset, SEEK_SET) == -1) {
709         P_ERRNO = P_ELSEEK;
710         return -1;
711       }
712       if (read(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
713         P_ERRNO = P_EREAD;
714         return -1;
715       }
716
717       // check for end of directory
718       if (entry.name[0] == 0)
719         break;
720
721       // skip deleted entries
722       if (entry.name[0] == 1 || entry.name[0] == 2) {
723         offset_in_block += sizeof(entry);
724         // check if we need to move to next block
725         if (offset_in_block + sizeof(entry) > block_size) {
726           current_block = fat[current_block];
727           offset_in_block = 0;
728         }
729         continue;
730       }
731
732       // format and write entry information
733       char output_buffer[256];
734       format_file_info(&entry, output_buffer);
735       if (k_write(STDOUT_FILENO, output_buffer, strlen(output_buffer)) < 0) {
736         return -1;
737       }
738
739       // move to next entry
740       offset_in_block += sizeof(entry);
741       // check if we need to move to next block
742       if (offset_in_block + sizeof(entry) > block_size) {
743         current_block = fat[current_block];
744         offset_in_block = 0;
745       }
746     }
747   } else {
748     // find and display specific file
749     int file_offset = find_file(filename, &entry);
750     if (file_offset < 0) {
751       P_ERRNO = P_ENOENT;
752       return -1;
753     }
754
755     char output_buffer[256];
756     format_file_info(&entry, output_buffer);
757     if (k_write(STDOUT_FILENO, output_buffer, strlen(output_buffer)) < 0) {
758       return -1;
759     }
760   }
761
762   return 0;
763 }
```

4.6.2.3 k_lseek()

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

Repositions the file offset of an open file.

This is a kernel-level function that changes the current position within an open file. The interpretation of the offset depends on the whence parameter.

Parameters

fd	File descriptor of the open file.
offset	The offset in bytes to set the position to.
whence	How to interpret the offset: • SEEK_SET (0): Offset is from the beginning of the file. • SEEK_CUR (1): Offset is from the current position. • SEEK_END (2): Offset is from the end of the file.

Returns

The new offset location on success, -1 on error with P_ERRNO set. Possible error codes:

- P_EBADF: Invalid file descriptor.
- P_EINVAL: Invalid whence or the resulting position would be negative.

Repositions the file offset of an open file.

Definition at line 622 of file fs_kfuncs.c.

```
622                                          {
623   // standard file descriptors don't support lseek
624   if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
625     P_ERRNO = P_EINVAL;
626     return -1;
627   }
628
629   // validate the file descriptor
630   if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
631     P_ERRNO = P_EBADF;
632     return -1;
633   }
634
635   // calculate new position based on whence
636   int32_t new_position;
637
638   switch (whence) {
639     case SEEK_SET:
640       new_position = offset;
641       break;
642     case SEEK_CUR:
643       new_position = fd_table[fd].position + offset;
644       break;
645     case SEEK_END:
646       new_position = fd_table[fd].size + offset;
647       break;
648     default:
649       P_ERRNO = P_EINVAL;
650       return -1;
651   }
652
653   // check if new position is valid
654   if (new_position < 0) {
```

```
        P_ERRNO = P_EINVAL;
        return -1;
    }

    // update file position
    fd_table[fd].position = new_position;

    return new_position;
}
```

4.6.2.4 k_open()

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

Opens a file with the specified mode.

This is a kernel-level function that opens a file and returns a file descriptor. The file is created if it doesn't exist and the mode includes F_WRITE. If the file exists and F_APPEND is specified, the file position is set to the end.

Parameters

fname	The name of the file to open.
mode	A combination of F_READ, F_WRITE, and F_APPEND.

Returns

A non-negative file descriptor on success, -1 on error with P_ERRNO set. Possible error codes:

- P_ENOENT: File doesn't exist and F_READ only.
- P_EFULL: Cannot create file (file system full).
- P_EINVAL: Invalid mode or filename.

Opens a file with the specified mode.

Definition at line 33 of file fs_kfuncs.c.

```
33                                        {
34   // validate arguments
35   if (fname == NULL || *fname == '\0') {
36     P_ERRNO = P_EINVAL;
37     return -1;
38   }
39   if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
40     P_ERRNO = P_EINVAL;
41     return -1;
42   }
43
44   // check if the file system is mounted
45   if (!is_mounted) {
46     P_ERRNO = P_EFS_NOT_MOUNTED;
47     return -1;
48   }
49
50   // get a free file descriptor
51   int fd = get_free_fd(fd_table);
52   if (fd < 0) {
53     P_ERRNO = P_EFULL;   // no free file descriptors
54     return -1;
55   }
56
57   // check if the file exists
58   dir_entry_t entry;
```

```
  int file_offset = find_file(fname, &entry);

  // file exists
  if (file_offset >= 0) {
    // check if the file is already open in write mode by another descriptor
    if ((mode & (F_WRITE | F_APPEND)) != 0) {
      for (int i = 0; i < MAX_FDS; i++) {
        if (i != fd && fd_table[i].in_use &&
            strcmp(fd_table[i].filename, fname) == 0 &&
            (fd_table[i].mode & (F_WRITE | F_APPEND)) != 0) {
          P_ERRNO = P_EBUSY;  // file is already open for writing
          return -1;
        }
      }
    }

    // fill in the file descriptor entry
    fd_table[fd].in_use = 1;
    fd_table[fd].ref_count++;
    strncpy(fd_table[fd].filename, fname, 31);
    fd_table[fd].filename[31] = '\0';
    fd_table[fd].size = entry.size;
    fd_table[fd].first_block = entry.firstBlock;
    fd_table[fd].mode = mode;

    // set the initial position
    if (mode & F_APPEND) {
      fd_table[fd].position = entry.size;
    } else {
      fd_table[fd].position = 0;
    }

    // if mode includes F_WRITE and not F_APPEND, truncate the file
    if ((mode & F_WRITE) && !(mode & F_APPEND)) {
      // free all blocks except the first one
      uint16_t block = entry.firstBlock;
      uint16_t next_block;

      if (block != 0 && block != FAT_EOF) {
        next_block = fat[block];
        fat[block] = FAT_EOF;  // terminate the chain at the first block
        block = next_block;

        // free the rest of the chain
        while (block != 0 && block != FAT_EOF) {
          next_block = fat[block];
          fat[block] = FAT_FREE;
          block = next_block;
        }
      }

      // update file size to 0
      fd_table[fd].size = 0;
      entry.size = 0;
      entry.mtime = time(NULL);

      // update the file system with the truncated file
      if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        return -1;
      }
      if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
        P_ERRNO = P_EWRITE;
        return -1;
      }
    }
  } else {
    // file doesn't exist

    // we can only create it if we are reading the file
    if (!(mode & F_WRITE)) {
      P_ERRNO = P_ENOENT;
      return -1;
    }

    // allocate the first block
    uint16_t first_block = allocate_block();
    if (first_block == 0) {
      P_ERRNO = P_EFULL;
      return -1;
    }

    // create a new file entry
    if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
        -1) {
      // error code already set by add_file_entry
      fat[first_block] = FAT_FREE;
```

```
      return -1;
    }

    // fill in the file descriptor entry
    fd_table[fd].in_use = 1;
    fd_table[fd].ref_count++;
    strncpy(fd_table[fd].filename, fname, 31);
    fd_table[fd].filename[31] = '\0';
    fd_table[fd].size = 0;
    fd_table[fd].first_block = first_block;
    fd_table[fd].position = 0;
    fd_table[fd].mode = mode;
  }

  return fd;
}
```

4.6.2.5 k_read()

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

Reads data from an open file.

This is a kernel-level function that reads up to n bytes from an open file into the provided buffer. The file position is advanced by the number of bytes read.

Parameters

fd	File descriptor of the open file.
n	Maximum number of bytes to read.
buf	Buffer to store the read data.

Returns

The number of bytes read on success, -1 on error with P_ERRNO set. Possible error codes:

- P_EBADF: Invalid file descriptor.
- P_EINVAL: Invalid buffer or count.

Reads data from an open file.

Definition at line 166 of file fs_kfuncs.c.

```
166                                     {
167     // handle terminal control (if doesn't control, send a STOP signal)
168     if (fd == STDIN_FILENO && current_running_pcb != NULL) {
169         if (current_running_pcb->pid != current_fg_pid) {
170             s_kill(current_running_pcb->pid, P_SIGSTOP);
171         }
172     }
173
174   // handle standard input
175   if (fd == STDIN_FILENO) {
176     return read(STDIN_FILENO, buf, n);
177   }
178
179   // validate inputs
180   if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
181     P_ERRNO = P_EBADF;
182     return -1;
183   }
```

```
if (buf == NULL || n < 0) {
  P_ERRNO = P_EINVAL;
  return -1;
}
if (n == 0) {
  return 0;
}

// check if we're at EOF already
if (fd_table[fd].position >= fd_table[fd].size) {
  return 0;
}

// determine how many bytes we can actually read
uint32_t bytes_to_read = n;
if (fd_table[fd].position + bytes_to_read > fd_table[fd].size) {
  bytes_to_read = fd_table[fd].size - fd_table[fd].position;
}

// find the block containing the current position
uint16_t current_block = fd_table[fd].first_block;
uint32_t block_index = fd_table[fd].position / block_size;
uint32_t block_offset = fd_table[fd].position % block_size;

// navigate to the correct block in the chain
for (uint32_t i = 0; i < block_index; i++) {
  if (current_block == 0 || current_block == FAT_EOF) {
    // unexpected end of chain
    P_ERRNO = P_EINVAL;
    return -1;
  }
  current_block = fat[current_block];
}

// now we're at the right block, start reading
uint32_t bytes_read = 0;

while (bytes_read < bytes_to_read) {
  // how much data can we read from the current block
  uint32_t bytes_left_in_block = block_size - block_offset;
  uint32_t bytes_to_read_now =
      (bytes_to_read - bytes_read) < bytes_left_in_block
          ? (bytes_to_read - bytes_read)
          : bytes_left_in_block;

  // seek to the right position in the file
  if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
            SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    if (bytes_read > 0) {
      fd_table[fd].position += bytes_read;
      return bytes_read;
    }
    return -1;
  }

  // read the data from the file
  ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
  if (read_result <= 0) {
    P_ERRNO = P_EREAD;
    // if we already read some data, return that count
    if (bytes_read > 0) {
      fd_table[fd].position += bytes_read;
      return bytes_read;
    }
    return -1;
  }

  bytes_read += read_result;
  block_offset += read_result;

  // if we've read all data from this block and still have more to read, go to
  // the next block
  if (block_offset == block_size && bytes_read < bytes_to_read) {
    if (current_block == FAT_EOF) {
      // unexpected end of chain
      break;
    }
    current_block = fat[current_block];
    block_offset = 0;
  }

  // if we read less than expected, we might have hit EOF
  if (read_result < bytes_to_read_now) {
    break;
  }
}
```

```

  // update file position
  fd_table[fd].position += bytes_read;

  return bytes_read;
}
```

4.6.2.6 k_unlink()

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

Removes a file from the file system.

This is a kernel-level function that deletes the specified file from the file system. The file must not be open by any process.

Parameters

fname	The name of the file to remove.

Returns

0 on success, -1 on error with P_ERRNO set. Possible error codes:

- P_ENOENT: File doesn't exist.
- P_EBUSY: File is still open by some process.

Removes a file from the file system.

Definition at line 566 of file fs_kfuncs.c.

```
566                                     {
567   if (fname == NULL || *fname == '\0') {
568     P_ERRNO = P_EINVAL;
569     return -1;
570   }
571
572   if (!is_mounted) {
573     P_ERRNO = P_EFS_NOT_MOUNTED;
574     return -1;
575   }
576
577   // check if file is currently open by any process
578   for (int i = 0; i < MAX_FDS; i++) {
579     if (fd_table[i].in_use && strcmp(fd_table[i].filename, fname) == 0) {
580       P_ERRNO = P_EBUSY;
581       return -1;
582     }
583   }
584
585   // find the file in directory
586   dir_entry_t entry;
587   int file_offset = find_file(fname, &entry);
588   if (file_offset < 0) {
589     P_ERRNO = P_ENOENT;
590     return -1;
591   }
592
593   // mark the directory entry as deleted (set first byte to 1)
594   entry.name[0] = 1;
595
596   // write the modified directory entry back
597   if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
598     P_ERRNO = P_ELSEEK;
599     return -1;
600   }
```

```
  if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
    P_ERRNO = P_EWRITE;
    return -1;
  }

  // free all blocks in the file chain
  uint16_t current_block = entry.firstBlock;
  uint16_t next_block;

  while (current_block != FAT_FREE && current_block != FAT_EOF) {
    next_block = fat[current_block];
    fat[current_block] = FAT_FREE;
    current_block = next_block;
  }

  return 0;
}
```

4.6.2.7 k_write()

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

Writes data to an open file.

This is a kernel-level function that writes n bytes from the provided buffer to an open file. The file position is advanced by the number of bytes written. If necessary, the file is extended.

Parameters

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

Returns

The number of bytes written on success, -1 on error with P_ERRNO set. Possible error codes:

- P_EBADF: Invalid file descriptor.
- P_EINVAL: Invalid buffer or count.
- P_EFULL: File system is full.

Writes data to an open file.

Definition at line 281 of file fs_kfuncs.c.

```
                                                   {
  // handle standard output and error
  if (fd == STDOUT_FILENO) {
    return write(STDOUT_FILENO, str, n);
  }
  if (fd == STDERR_FILENO) {
    return write(STDERR_FILENO, str, n);
  }

  // validate inputs
  if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
    P_ERRNO = P_EBADF;
    return -1;
  }
  if (str == NULL || n < 0) {
```

```
    P_ERRNO = P_EINVAL;
    return -1;
  }
  if (n == 0) {
    return 0;
  }

  // check if filesystem is mounted and FAT is valid
  if (!is_mounted || fat == NULL) {
    P_ERRNO = P_EFS_NOT_MOUNTED;
    return -1;
  }

  // get file information
  uint16_t current_block = fd_table[fd].first_block;
  uint32_t current_position = fd_table[fd].position;

  // create a local buffer for block data
  char* block_buffer = (char*)malloc(block_size);
  if (block_buffer == NULL) {
    P_ERRNO = P_EMALLOC;
    return -1;
  }

  // calculate initial block position
  uint32_t block_index = current_position / block_size;
  uint32_t block_offset = current_position % block_size;

  // if the file doesn't have a first block yet, allocate one
  if (current_block == 0) {
    current_block = allocate_block();
    if (current_block == 0) {
      P_ERRNO = P_EFULL;
      free(block_buffer);
      return -1;
    }
    fd_table[fd].first_block = current_block;
  }

  // navigate to the appropriate block
  uint16_t prev_block = 0;
  for (uint32_t i = 0; i < block_index; i++) {
    if (current_block == 0 || current_block == FAT_EOF ||
        current_block >= fat_size / 2) {
      // reached the end of chain prematurely, need to allocate a new block
      uint16_t new_block = allocate_block();
      if (new_block == 0) {
        P_ERRNO = P_EFULL;
        free(block_buffer);
        return -1;
      }

      // update the chain
      if (prev_block != 0 && prev_block < fat_size / 2) {
        fat[prev_block] = new_block;
      } else {
        // if there's no previous block, this must be the first one
        fd_table[fd].first_block = new_block;
      }

      current_block = new_block;
    }

    prev_block = current_block;

    // validate the block number before accessing FAT
    if (current_block >= fat_size / 2) {
      P_ERRNO = P_EINVAL;
      free(block_buffer);
      return -1;
    }

    current_block = fat[current_block];
  }

  // if we ended up without a valid block, go back to the last valid one
  if (current_block == 0 || current_block == FAT_EOF ||
      current_block >= fat_size / 2) {
    if (prev_block != 0 && prev_block < fat_size / 2) {
      uint16_t new_block = allocate_block();
      if (new_block == 0) {
        P_ERRNO = P_EFULL;
        free(block_buffer);
        return -1;
      }

      fat[prev_block] = new_block;
```

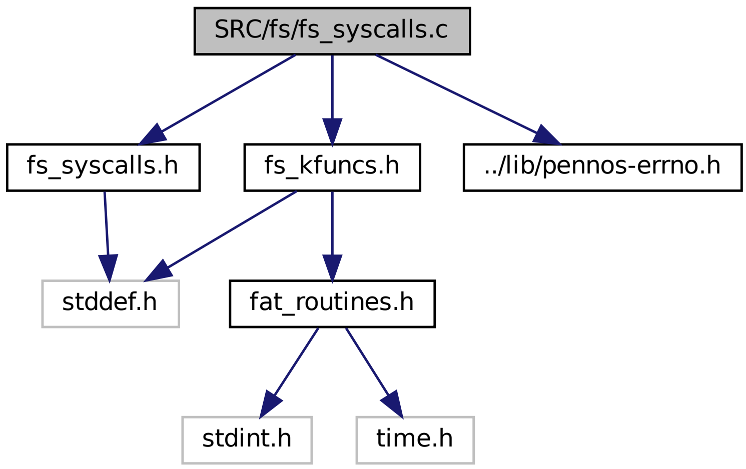
```
      current_block = new_block;
    } else {
      P_ERRNO = P_EINVAL;
      free(block_buffer);
      return -1;
    }
  }

  // start writing data
  uint32_t bytes_written = 0;

  while (bytes_written < n) {
    // validate current block
    if (current_block == 0 || current_block == FAT_EOF ||
        current_block >= fat_size / 2) {
      P_ERRNO = P_EINVAL;
      break;
    }

    // how much can we write to this block
    uint32_t space_in_block = block_size - block_offset;
    uint32_t bytes_to_write = (n - bytes_written) < space_in_block
                                  ? (n - bytes_written)
                                  : space_in_block;

    // position in filesystem
    off_t block_position = fat_size + (current_block - 1) * block_size;

    // if we're not writing a full block or not starting at the beginning, we
    // need to read-modify-write
    if (bytes_to_write < block_size || block_offset > 0) {
      // read the current block
      if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        break;
      }

      // read the current block data
      ssize_t read_result = read(fs_fd, block_buffer, block_size);
      if (read_result < 0) {
        P_ERRNO = P_EREAD;
        break;
      }

      // copy the new data into the block buffer
      memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);

      // seek back to write the modified block
      if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        break;
      }

      // write the full block back
      ssize_t write_result = write(fs_fd, block_buffer, block_size);
      if (write_result != block_size) {
        P_ERRNO = P_EWRITE;
        // we might have a partial write, but that's hard to handle correctly
        break;
      }
    } else {
      // we're writing a full block from the beginning
      if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        break;
      }

      ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
      if (write_result != bytes_to_write) {
        P_ERRNO = P_EWRITE;
        break;
      }
    }

    // update counters
    bytes_written += bytes_to_write;
    block_offset = (block_offset + bytes_to_write) % block_size;

    // if we've filled this block and still have more to write, go to the next
    // block
    if (block_offset == 0 && bytes_written < n) {
      // validate current block before accessing FAT
      if (current_block >= fat_size / 2) {
        P_ERRNO = P_EINVAL;
        break;
      }

```

```
      // check if there's a next block
      if (fat[current_block] == FAT_EOF) {
        // allocate a new block
        uint16_t new_block = allocate_block();
        if (new_block == 0) {
          P_ERRNO = P_EFULL;
          break;
        }

        // Update the FAT safely
        if (current_block < fat_size / 2) {
          fat[current_block] = new_block;
        } else {
          P_ERRNO = P_EINVAL;
          break;
        }

        current_block = new_block;
      } else {
        current_block = fat[current_block];
      }
    }
  }

  // free the block buffer
  free(block_buffer);

  // update file position
  fd_table[fd].position += bytes_written;

  // update file size if needed
  if (fd_table[fd].position > fd_table[fd].size) {
    fd_table[fd].size = fd_table[fd].position;

    // update the directory entry
    dir_entry_t entry;
    int dir_offset = find_file(fd_table[fd].filename, &entry);
    if (dir_offset >= 0) {
      entry.size = fd_table[fd].size;
      entry.mtime = time(NULL);

      if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
        return -1;
      }
      if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
        P_ERRNO = P_EWRITE;
        return -1;
      }
    }
  }

  return bytes_written;
}
```

4.7 SRC/fs/fs_syscalls.c File Reference

```
#include "fs_syscalls.h"
#include "fs_kfuncs.h"
#include "../lib/pennos-errno.h"
```

Include dependency graph for fs_syscalls.c:



Functions

- int s_open (const char ∗fname, int mode)

    System call to open a file.
- int s_read (int fd, char ∗buf, int n)

    System call to read from a file.
- int s_write (int fd, const char ∗str, int n)

    System call to write to a file.
- int s_close (int fd)

    System call to close a file.
- int s_unlink (const char ∗fname)

    System call to remove a file.
- int s_lseek (int fd, int offset, int whence)

    System call to reposition the file offset.
- int s_ls (const char ∗filename)

    System call to list files.

4.7.1 Function Documentation

4.7.1.1 s_close()

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

System call to close a file.

Closes an open file descriptor.

This is a wrapper around the kernel function k_close.

Definition at line 42 of file fs_syscalls.c.

```
                      {
    return k_close(fd);
}
```

4.7.1.2 s_ls()

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

System call to list files.

Lists files in the current directory or displays file information.

This is a wrapper around the kernel function k_ls.

Definition at line 69 of file fs_syscalls.c.

```
                                {
    return k_ls(filename);
}
```

4.7.1.3 s_lseek()

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

System call to reposition the file offset.

Repositions the file offset of an open file.

This is a wrapper around the kernel function k_lseek.

Definition at line 60 of file fs_syscalls.c.

```
                                              {
    return k_lseek(fd, offset, whence);
}
```

4.7.1.4 s_open()

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

System call to open a file.

Opens a file with the specified access mode.

This is a wrapper around the kernel function k_open.

Definition at line 15 of file fs_syscalls.c.

```
15                                  {
16     return k_open(fname, mode);
17 }
```

4.7.1.5 s_read()

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

System call to read from a file.

Reads data from an open file.

This is a wrapper around the kernel function k_read.

Definition at line 24 of file fs_syscalls.c.

```
24                               {
25     return k_read(fd, buf, n);
26 }
```

4.7.1.6 s_unlink()

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

System call to remove a file.

Removes a file from the file system.

This is a wrapper around the kernel function k_unlink.

Definition at line 51 of file fs_syscalls.c.

```
51                          {
52     return k_unlink(fname);
53 }
```

4.7.1.7 s_write()

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

System call to write to a file.

Writes data to an open file.

This is a wrapper around the kernel function k_write.

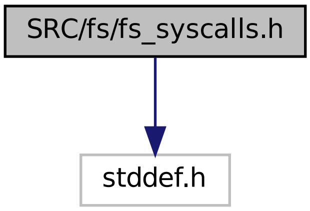
Definition at line 33 of file fs_syscalls.c.

```
33                                         {
34     return k_write(fd, str, n);
35 }
```

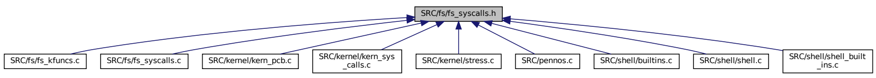
4.8 SRC/fs/fs_syscalls.h File Reference

`#include <stddef.h>`

Include dependency graph for fs_syscalls.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define STDIN_FILENO 0
- #define STDOUT_FILENO 1
- #define STDERR_FILENO 2

Functions

- int s_open (const char ∗fname, int mode)
  Opens a file with the specified access mode.
- int s_read (int fd, char ∗buf, int n)
  Reads data from an open file.
- int s_write (int fd, const char ∗str, int n)
  Writes data to an open file.
- int s_close (int fd)
  Closes an open file descriptor.
- int s_unlink (const char ∗fname)
  Removes a file from the file system.
- int s_lseek (int fd, int offset, int whence)
  Repositions the file offset of an open file.
- int s_ls (const char ∗filename)
  Lists files in the current directory or displays file information.

4.8.1 Macro Definition Documentation

4.8.1.1 STDERR_FILENO

```
#define STDERR_FILENO 2
```

Definition at line 18 of file fs_syscalls.h.

4.8.1.2 STDIN_FILENO

```
#define STDIN_FILENO 0
```

Definition at line 16 of file fs_syscalls.h.

4.8.1.3 STDOUT_FILENO

```
#define STDOUT_FILENO 1
```

Definition at line 17 of file fs_syscalls.h.

4.8.2 Function Documentation

4.8.2.1 s_close()

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

Closes an open file descriptor.

This function closes the file descriptor fd, making it available for reuse. If this is the last reference to the underlying file, any necessary cleanup is performed.

Parameters

fd	The file descriptor to close.

Returns

On success, returns 0. On error, returns -1 and sets P_ERRNO appropriately:

- P_EBADF: fd is not a valid file descriptor.

Closes an open file descriptor.

This is a wrapper around the kernel function k_close.

Definition at line 42 of file fs_syscalls.c.

```
42                  {
43      return k_close(fd);
44 }
```

4.8.2.2 s_ls()

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

Lists files in the current directory or displays file information.

If filename is NULL, this function lists all files in the current directory. If filename refers to a specific file, it displays detailed information about that file.

Parameters

filename	The name of the file to get information about, or NULL to list all files.

Returns

On success, returns 0. On error, returns -1 and sets P_ERRNO appropriately:

- P_ENOENT: The specified file does not exist.

Lists files in the current directory or displays file information.

This is a wrapper around the kernel function k_ls.

Definition at line 69 of file fs_syscalls.c.

```
69                              {
70      return k_ls(filename);
71 }
```

4.8.2.3 s_lseek()

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

Repositions the file offset of an open file.

This function repositions the offset of the file descriptor fd to the argument offset according to the directive whence.

Parameters

fd	The file descriptor of an open file.
offset	The offset in bytes.
whence	Specifies the reference position: • SEEK_SET (0): The offset is set relative to the start of the file. • SEEK_CUR (1): The offset is set relative to the current position. • SEEK_END (2): The offset is set relative to the end of the file.

Returns

On success, returns the resulting offset from the beginning of the file. On error, returns -1 and sets P_ERRNO appropriately:

- P_EBADF: fd is not a valid file descriptor.
- P_EINVAL: whence is not valid or the resulting offset would be negative.

Repositions the file offset of an open file.

This is a wrapper around the kernel function k_lseek.

Definition at line 60 of file fs_syscalls.c.

```
60                                        {
61     return k_lseek(fd, offset, whence);
62 }
```

4.8.2.4 s_open()

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

Opens a file with the specified access mode.

This function provides a user-level interface to the kernel's file open operation. It opens the specified file with the given access mode and returns a file descriptor that can be used in subsequent operations on the file.

Parameters

fname	The name of the file to open.
mode	A combination of F_READ, F_WRITE, and F_APPEND.

Returns

On success, returns a non-negative integer representing the file descriptor. On error, returns -1 and sets P_ERRNO appropriately:

- P_ENOENT: The file does not exist and F_READ was specified.
- P_EINVAL: Invalid parameters (NULL filename or invalid mode).
- P_EFULL: No space left on device or file descriptor table is full.

Opens a file with the specified access mode.

This is a wrapper around the kernel function k_open.

Definition at line 15 of file fs_syscalls.c.

```
15                                            {
16     return k_open(fname, mode);
17 }
```

4.8.2.5 s_read()

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

Reads data from an open file.

This function reads up to n bytes from the file associated with the file descriptor fd into the buffer starting at buf. The file offset is advanced by the number of bytes read.

Parameters

fd	The file descriptor of an open file.
n	The maximum number of bytes to read.
buf	The buffer to store the read data.

Returns

On success, returns the number of bytes read (0 indicates end of file). On error, returns -1 and sets P_ERRNO appropriately:

- P_EBADF: fd is not a valid file descriptor or is not open for reading.
- P_EINVAL: Invalid parameters (NULL buffer or negative count).

Reads data from an open file.

This is a wrapper around the kernel function k_read.

Definition at line 24 of file fs_syscalls.c.
```
24                                    {
25     return k_read(fd, buf, n);
26 }
```

4.8.2.6 s_unlink()

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

Removes a file from the file system.

This function removes the specified file from the file system. If the file is currently open, the behavior depends on the implementation.

Parameters

fname	The name of the file to remove.

Returns

On success, returns 0. On error, returns -1 and sets P_ERRNO appropriately:

- P_ENOENT: The file does not exist.
- P_EBUSY: The file is currently in use.
- P_EINVAL: Invalid parameter (NULL filename).

Removes a file from the file system.

This is a wrapper around the kernel function k_unlink.

Definition at line 51 of file fs_syscalls.c.
```
51                                {
52     return k_unlink(fname);
53 }
```

4.8.2.7 s_write()

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

Writes data to an open file.

This function writes up to n bytes from the buffer starting at str to the file associated with the file descriptor fd. The file offset is advanced by the number of bytes written.

Parameters

fd	The file descriptor of an open file.
str	The buffer containing the data to be written.
n	The number of bytes to write.

Returns

On success, returns the number of bytes written. On error, returns -1 and sets P_ERRNO appropriately:

- P_EBADF: fd is not a valid file descriptor or is not open for writing.
- P_EINVAL: Invalid parameters (NULL buffer or negative count).
- P_EFULL: No space left on device.

Writes data to an open file.

This is a wrapper around the kernel function k_write.

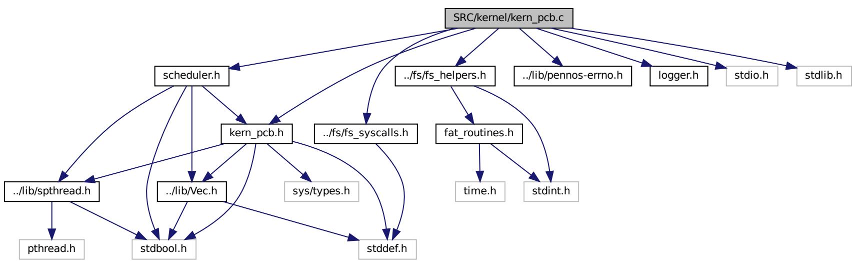
Definition at line 33 of file fs_syscalls.c.

```
33                                   {
34     return k_write(fd, str, n);
35 }
```

4.9 SRC/kernel/kern_pcb.c File Reference

```
#include "kern_pcb.h"
#include "../fs/fs_helpers.h"
#include "../fs/fs_syscalls.h"
#include "../lib/pennos-errno.h"
#include "logger.h"
#include "scheduler.h"
#include "stdio.h"
#include "stdlib.h"
```

Include dependency graph for kern_pcb.c:



Functions

- void free_pcb (void ∗pcb)

  Free resources associated with a PCB.
- pcb_t ∗ create_pcb (pid_t pid, pid_t par_pid, int priority, int input_fd, int output_fd)

  Initializes a PCB with the given parameters.
- void remove_child_in_parent (pcb_t ∗parent, pcb_t ∗child)

  Removes a child PCB from its parent's child list.
- pcb_t ∗ k_proc_create (pcb_t ∗parent, int priority)

  Creates a new process. If the parent is NULL, it creates the init process.
- void k_proc_cleanup (pcb_t ∗proc)

  Cleans up a process by removing it from its parent's child list, removing its children, decrementing file descriptor reference counts, closing files, and freeing the PCB.

Variables

- int next_pid = 2
- Vec current_pcbs
- pcb_t ∗ current_running_pcb

4.9.1 Function Documentation

4.9.1.1 create_pcb()

```
pcb_t* create_pcb (
            pid_t pid,
            pid_t par_pid,
            int priority,
            int input_fd,
            int output_fd )
```

Initializes a PCB with the given parameters.

Creates a new PCB and initializes its fields. Notably, the thread handle and cmd are left out. It's up to the user to assign them post-call.

Definition at line 41 of file kern_pcb.c.

```
{
  pcb_t* ret_pcb = malloc(sizeof(pcb_t));
  if (ret_pcb == NULL) {
    perror("malloc failed for PCB creation");
    return NULL;
  }

  ret_pcb->pid = pid;
  ret_pcb->par_pid = par_pid;
  ret_pcb->priority = priority;
  ret_pcb->process_state = 'R';  // running by default
  ret_pcb->input_fd = input_fd;
  ret_pcb->output_fd = output_fd;
  ret_pcb->process_status = 0;  // default status

  ret_pcb->child_pcbs = vec_new(0, NULL); // NULL deconstructor prevents
                                          // double free

  for (int i = 0; i < 3; i++) {
    ret_pcb->signals[i] = false;
  }

  ret_pcb->is_sleeping = false;
  ret_pcb->time_to_wake = -1;  // default to not sleeping

  return ret_pcb;
}
```

4.9.1.2 free_pcb()

```
void free_pcb (
            void * pcb )
```

Free resources associated with a PCB.

Frees all malloced memory associated with the PCB. Note that this will destroy children too, so be careful when using it. In particular, make sure to remove any children pcbs you want to preserve.

Definition at line 29 of file kern_pcb.c.

```
29                       {
30   pcb_t* casted_pcb = (pcb_t*)pcb;
31
32   free(casted_pcb->cmd_str);
33   vec_destroy(&casted_pcb->child_pcbs);  // will free any remaining
34                                          // children too!
35   free(casted_pcb);
36 }
```

4.9.1.3 k_proc_cleanup()

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

Cleans up a process by removing it from its parent's child list, removing its children, decrementing file descriptor reference counts, closing files, and freeing the PCB.

Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc. If a child is orphaned, the INIT process becomes its parent.

Definition at line 150 of file kern_pcb.c.

```
150                              {
151   // if proc has parent (i.e. isn't init) then remove it from parent's child
152   // list
153   pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
154   if (par_pcb != NULL) {
155     remove_child_in_parent(par_pcb, proc);
156   } else {
157     P_ERRNO = P_ENULL;
158     return;
159   }
160
161   // if proc has children, remove them and assign them to init parent
162   if (vec_len(&proc->child_pcbs) > 0) {
163     // retrieve the init process
164     pcb_t* init_pcb =
165         get_pcb_in_queue(&current_pcbs, 1);  // init process has pid 1
166
167     while (vec_len(&proc->child_pcbs) > 0) {
168       pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
169       vec_push_back(&init_pcb->child_pcbs, curr_child);
170       vec_erase_no_deletor(&proc->child_pcbs, 0);  // don't free in erase
171       curr_child->par_pid = 1;  // update parent to init (pid 1)
172       log_generic_event('O', curr_child->pid, curr_child->priority,
173                         curr_child->cmd_str);
174     }
175   }
176
177   // decr reference counts + close files if necessary
178   for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
179     if (proc->fd_table[i] != -1) {
180       if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
181         s_close(
182             proc->fd_table[i]);  // close the fd since no other process using
183       }
184     }
185   }
186
187   // cancel + join this thread
```

```
  spthread_cancel(proc->thread_handle);
  spthread_continue(proc->thread_handle);
  spthread_suspend(proc->thread_handle);
  spthread_join(proc->thread_handle, NULL);

  // delete this process from any queue it's in + free it
  delete_process_from_all_queues(proc);
  free_pcb(proc);
}
```

4.9.1.4 k_proc_create()

```
pcb_t* k_proc_create (
            pcb_t * parent,
            int priority )
```

Creates a new process. If the parent is NULL, it creates the init process.

Create a new child process, inheriting applicable properties from the parent. Also inserts the created child into the correct scheduler queue based on its priority.

Definition at line 93 of file kern_pcb.c.

```
93                                                      {
94   if (parent == NULL) {  // init creation case
95     pcb_t* init = create_pcb(1, 0, 0, 0, 1);
96     if (init == NULL) {
97       P_ERRNO = P_ENULL;
98       return NULL;
99     }
100    init->fd_table[0] = STDIN_FILENO;
101    init->fd_table[1] = STDOUT_FILENO;
102    init->fd_table[2] = STDERR_FILENO;
103    for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
104      init->fd_table[i] = -1;
105    }
106
107    increment_fd_ref_count(STDIN_FILENO);
108    increment_fd_ref_count(STDOUT_FILENO);
109    increment_fd_ref_count(STDERR_FILENO);
110
111    current_running_pcb = init;
112    put_pcb_into_correct_queue(init);
113    vec_push_back(&current_pcbs, init);
114    return init;
115  }
116
117  pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
118                            parent->output_fd);
119  if (child == NULL) {
120    P_ERRNO = P_ENULL;
121    return NULL;
122  }
123
124  // copy parent's fd table
125  for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
126    child->fd_table[i] = parent->fd_table[i];
127  }
128
129  for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
130    if (child->fd_table[i] != -1) {
131      increment_fd_ref_count(child->fd_table[i]);
132    }
133  }
134
135  // update parent as needed
136  vec_push_back(&parent->child_pcbs, child);
137
138  // add to appropriate queue
139  put_pcb_into_correct_queue(child);
140  vec_push_back(&current_pcbs, child);
141
142  return child;
143 }
```

4.9.1.5 remove_child_in_parent()

```
void remove_child_in_parent (
            pcb_t * parent,
            pcb_t * child )
```

Removes a child PCB from its parent's child list.

Given a parent, removes the child from the parent's child vector if its exists. Notably, it does not free the child but simply removes it via the vec_erase_no_deletor function.

Definition at line 76 of file kern_pcb.c.

```
                                                            {
  for (int i = 0; i < vec_len(&parent->child_pcbs); i++) {
    pcb_t* curr_child = (pcb_t*)vec_get(&parent->child_pcbs, i);
    if (curr_child->pid == child->pid) {
      vec_erase_no_deletor(&parent->child_pcbs, i);
      return;
    }
  }
}
```

4.9.2 Variable Documentation

4.9.2.1 current_pcbs

```
Vec current_pcbs [extern]
```

Definition at line 30 of file scheduler.c.

4.9.2.2 current_running_pcb

```
pcb_t* current_running_pcb [extern]
```

Definition at line 38 of file scheduler.c.

4.9.2.3 next_pid

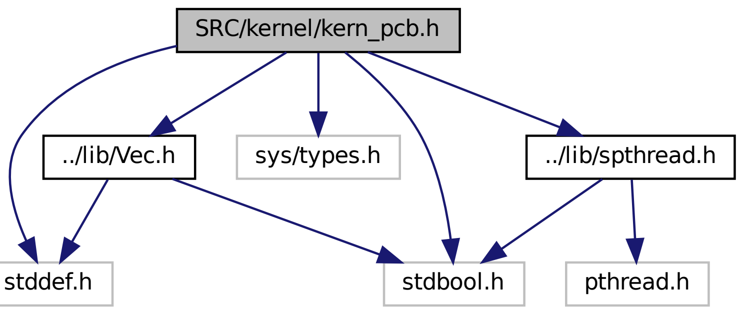
```
int next_pid = 2
```

Definition at line 16 of file kern_pcb.c.

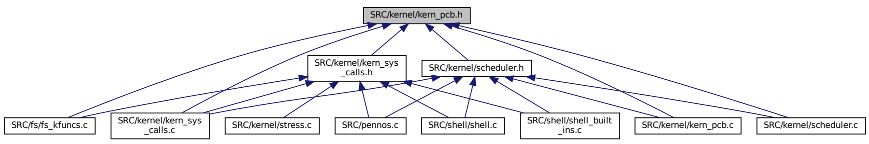
4.10 SRC/kernel/kern_pcb.h File Reference

```
#include <stddef.h>
#include <stdbool.h>
#include <sys/types.h>
#include "../lib/spthread.h"
#include "../lib/Vec.h"
```

Include dependency graph for kern_pcb.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct pcb_st

  The PCB structure, which contains all the information about a process. Notably, it contains the thread handle, pid, parent pid, child pcbs, priority level, process state, command string, signals to be sent, input and output file descriptors, process status, sleeping status, and time to wake.

Macros

- #define FILE_DESCRIPTOR_TABLE_SIZE 100

Typedefs

- typedef struct pcb_st pcb_t

  The PCB structure, which contains all the information about a process. Notably, it contains the thread handle, pid, parent pid, child pcbs, priority level, process state, command string, signals to be sent, input and output file descriptors, process status, sleeping status, and time to wake.

Functions

- pcb_t ∗ create_pcb (pid_t pid, pid_t par_pid, int priority, int input_fd, int output_fd)

  Creates a new PCB and initializes its fields. Notably, the thread handle and cmd are left out. It's up to the user to assign them post-call.
- void free_pcb (void ∗pcb)

  Frees all malloced memory associated with the PCB. Note that this will destroy children too, so be careful when using it. In particular, make sure to remove any children pcbs you want to preserve.
- void remove_child_in_parent (pcb_t ∗parent, pcb_t ∗child)

  Given a parent, removes the child from the parent's child vector if its exists. Notably, it does not free the child but simply removes it via the vec_erase_no_deletor function.
- pcb_t ∗ k_proc_create (pcb_t ∗parent, int priority)

  Create a new child process, inheriting applicable properties from the parent. Also inserts the created child into the correct scheduler queue based on its priority.
- void k_proc_cleanup (pcb_t ∗proc)

  Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc. If a child is orphaned, the INIT process becomes its parent.

4.10.1 Macro Definition Documentation

4.10.1.1 FILE_DESCRIPTOR_TABLE_SIZE

```
#define FILE_DESCRIPTOR_TABLE_SIZE 100
```

Definition at line 17 of file kern_pcb.h.

4.10.2 Typedef Documentation

4.10.2.1 pcb_t

```
typedef struct pcb_st pcb_t
```

The PCB structure, which contains all the information about a process. Notably, it contains the thread handle, pid, parent pid, child pcbs, priority level, process state, command string, signals to be sent, input and output file descriptors, process status, sleeping status, and time to wake.

4.10.3 Function Documentation

4.10.3.1 create_pcb()

```
pcb_t* create_pcb (
            pid_t pid,
            pid_t par_pid,
            int priority,
            int input_fd,
            int output_fd )
```

Creates a new PCB and initializes its fields. Notably, the thread handle and cmd are left out. It's up to the user to assign them post-call.

Parameters

pid	the new process id
par_pid	the parent process id
priority	the priority level (0,1,2)
input_fd	input fd
output↩_fd	output fd

Returns

pointer to the newly created and malloced PCB or NULL if failure

Creates a new PCB and initializes its fields. Notably, the thread handle and cmd are left out. It's up to the user to assign them post-call.

Definition at line 41 of file kern_pcb.c.

```
                                       {
  pcb_t* ret_pcb = malloc(sizeof(pcb_t));
  if (ret_pcb == NULL) {
    perror("malloc failed for PCB creation");
    return NULL;
  }

  ret_pcb->pid = pid;
  ret_pcb->par_pid = par_pid;
  ret_pcb->priority = priority;
  ret_pcb->process_state = 'R';  // running by default
  ret_pcb->input_fd = input_fd;
  ret_pcb->output_fd = output_fd;
  ret_pcb->process_status = 0;  // default status

  ret_pcb->child_pcbs = vec_new(0, NULL); // NULL deconstructor prevents
                                          // double free

  for (int i = 0; i < 3; i++) {
    ret_pcb->signals[i] = false;
  }

  ret_pcb->is_sleeping = false;
  ret_pcb->time_to_wake = -1;  // default to not sleeping

  return ret_pcb;
}
```

4.10.3.2 free_pcb()

```
void free_pcb (
            void * pcb )
```

Frees all malloced memory associated with the PCB. Note that this will destroy children too, so be careful when using it. In particular, make sure to remove any children pcbs you want to preserve.

Parameters

pcb	Pointer to the PCB to be freed, NULL if error

Frees all malloced memory associated with the PCB. Note that this will destroy children too, so be careful when using it. In particular, make sure to remove any children pcbs you want to preserve.

Definition at line 29 of file kern_pcb.c.

```
29                     {
30   pcb_t* casted_pcb = (pcb_t*)pcb;
31 
32   free(casted_pcb->cmd_str);
33   vec_destroy(&casted_pcb->child_pcbs);  // will free any remaining
34                                          // children too!
35   free(casted_pcb);
36 }
```

4.10.3.3 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. If a child is orphaned, the INIT process becomes its parent.

Parameters

proc	a pcb ptr to the terminated/finished thread

Clean up a terminated/finished thread's resources. This may include freeing the PCB, handling children, etc. If a child is orphaned, the INIT process becomes its parent.

Definition at line 150 of file kern_pcb.c.

```
150                               {
151   // if proc has parent (i.e. isn't init) then remove it from parent's child
152   // list
153   pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
154   if (par_pcb != NULL) {
155     remove_child_in_parent(par_pcb, proc);
156   } else {
157     P_ERRNO = P_ENULL;
158     return;
159   }
160 
161   // if proc has children, remove them and assign them to init parent
162   if (vec_len(&proc->child_pcbs) > 0) {
163     // retrieve the init process
164     pcb_t* init_pcb =
165         get_pcb_in_queue(&current_pcbs, 1);  // init process has pid 1
166 
167     while (vec_len(&proc->child_pcbs) > 0) {
168       pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
169       vec_push_back(&init_pcb->child_pcbs, curr_child);
170       vec_erase_no_deletor(&proc->child_pcbs, 0);  // don't free in erase
171       curr_child->par_pid = 1;  // update parent to init (pid 1)
172       log_generic_event('O', curr_child->pid, curr_child->priority,
173                         curr_child->cmd_str);
174     }
175   }
176 
177   // decr reference counts + close files if necessary
178   for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
179     if (proc->fd_table[i] != -1) {
180       if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
181         s_close(
182             proc->fd_table[i]);  // close the fd since no other process using
183       }
184     }
185   }
186 
187   // cancel + join this thread
188   spthread_cancel(proc->thread_handle);
189   spthread_continue(proc->thread_handle);
190   spthread_suspend(proc->thread_handle);
191   spthread_join(proc->thread_handle, NULL);
192 
193   // delete this process from any queue it's in + free it
194   delete_process_from_all_queues(proc);
```

```
195   free_pcb(proc);
196 }
```

4.10.3.4 k_proc_create()

```
pcb_t* k_proc_create (
            pcb_t * parent,
            int priority )
```

Create a new child process, inheriting applicable properties from the parent. Also inserts the created child into the correct scheduler queue based on its priority.

Parameters

parent	a pointer to the parent pcb
priority	the priority of the child, usually 1 but exceptions like shell exist

Returns

Reference to the child PCB or NULL if error

Create a new child process, inheriting applicable properties from the parent. Also inserts the created child into the correct scheduler queue based on its priority.

Definition at line 93 of file kern_pcb.c.

```
93                                                      {
94    if (parent == NULL) {  // init creation case
95      pcb_t* init = create_pcb(1, 0, 0, 0, 1);
96      if (init == NULL) {
97        P_ERRNO = P_ENULL;
98        return NULL;
99      }
100     init->fd_table[0] = STDIN_FILENO;
101     init->fd_table[1] = STDOUT_FILENO;
102     init->fd_table[2] = STDERR_FILENO;
103     for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
104       init->fd_table[i] = -1;
105     }
106
107     increment_fd_ref_count(STDIN_FILENO);
108     increment_fd_ref_count(STDOUT_FILENO);
109     increment_fd_ref_count(STDERR_FILENO);
110
111     current_running_pcb = init;
112     put_pcb_into_correct_queue(init);
113     vec_push_back(&current_pcbs, init);
114     return init;
115   }
116
117   pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
118                             parent->output_fd);
119   if (child == NULL) {
120     P_ERRNO = P_ENULL;
121     return NULL;
122   }
123
124   // copy parent's fd table
125   for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
126     child->fd_table[i] = parent->fd_table[i];
127   }
128
129   for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
130     if (child->fd_table[i] != -1) {
131       increment_fd_ref_count(child->fd_table[i]);
132     }
133   }
```

```c

  // update parent as needed
  vec_push_back(&parent->child_pcbs, child);

  // add to appropriate queue
  put_pcb_into_correct_queue(child);
  vec_push_back(&current_pcbs, child);

  return child;
}
```

4.10.3.5 remove_child_in_parent()

```
void remove_child_in_parent (
            pcb_t * parent,
            pcb_t * child )
```

Given a parent, removes the child from the parent's child vector if its exists. Notably, it does not free the child but simply removes it via the vec_erase_no_deletor function.

Parameters

parent	a ptr to the parent pcb with the child list
child	a ptr to the child pcb that we'd like to remove

Given a parent, removes the child from the parent's child vector if its exists. Notably, it does not free the child but simply removes it via the vec_erase_no_deletor function.

Definition at line 76 of file kern_pcb.c.

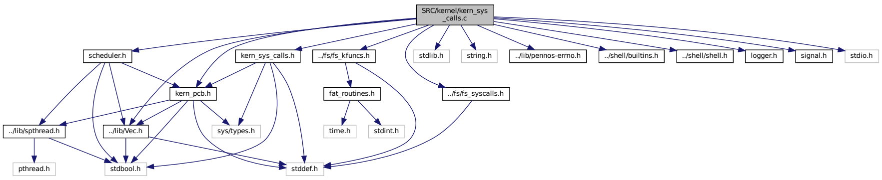
```c
76                                                                {
77   for (int i = 0; i < vec_len(&parent->child_pcbs); i++) {
78     pcb_t* curr_child = (pcb_t*)vec_get(&parent->child_pcbs, i);
79     if (curr_child->pid == child->pid) {
80       vec_erase_no_deletor(&parent->child_pcbs, i);
81       return;
82     }
83   }
84 }
```

4.11 SRC/kernel/kern_sys_calls.c File Reference

```c
#include "kern_sys_calls.h"
#include <stdlib.h>
#include <string.h>
#include "../fs/fs_kfuncs.h"
#include "../lib/Vec.h"
#include "../lib/pennos-errno.h"
#include "../shell/builtins.h"
#include "../shell/shell.h"
#include "kern_pcb.h"
#include "logger.h"
#include "scheduler.h"
#include "signal.h"
#include "../fs/fs_syscalls.h"
```

`#include <stdio.h>`

Include dependency graph for kern_sys_calls.c:



Functions

- int determine_index_in_queue (Vec ∗queue, int pid)

  Determines the index of a PCB in a given queue.
- void move_pcb_correct_queue (int prev_priority, int new_priority, pcb_t ∗curr_pcb)

  Moves a PCB from its previous priority queue to its new priority queue.
- void delete_from_queue (int queue_id, int pid)

  Deletes a PCB from the specified queue based on its PID.
- void delete_from_explicit_queue (Vec ∗queue_to_delete_from, int pid)

  Deletes a PCB from the specified explicit queue based on its PID.
- void ∗ init_func (void ∗input)

  The function that runs the shell process.
- pid_t s_spawn_init ()

  Creates the init process and spawns the shell process.
- void s_cleanup_init_process ()

  Cleans up Init's resources.
- pid_t s_spawn (void ∗(∗func)(void ∗), char ∗argv[ ], int fd0, int fd1)

  Spawns a child process with the given function and arguments.
- pid_t s_waitpid (pid_t pid, int ∗wstatus, bool nohang)

  Waits for a child of the calling process.
- int s_kill (pid_t pid, int signal)

  Sends a signal to a process with specified pid.
- void s_exit (void)

  Exits the current process and cleans up its resources.
- int s_nice (pid_t pid, int priority)

  Sets the priority of a process with specified pid.
- void s_sleep (unsigned int ticks)

  Suspends the current process for a specified number of ticks.
- void ∗ s_echo (void ∗arg)

  System-level wrapper for the shell built-in command "echo".
- void ∗ s_ps (void ∗arg)

  System-level wrapper for the shell built-in command "ps".

Variables

- Vec zero_priority_queue
- Vec one_priority_queue
- Vec two_priority_queue
- Vec zombie_queue
- Vec sleep_blocked_queue
- Vec current_pcbs
- pcb_t ∗ current_running_pcb
- int tick_counter
- pid_t current_fg_pid = 2

4.11.1 Function Documentation

4.11.1.1 delete_from_explicit_queue()

```
void delete_from_explicit_queue (
            Vec * queue_to_delete_from,
            int pid )
```

Deletes a PCB from the specified explicit queue based on its PID.

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.

Definition at line 107 of file kern_sys_calls.c.

```
107                                                              {
108   int index = determine_index_in_queue(queue_to_delete_from, pid);
109   if (index != -1) {
110     vec_erase_no_deletor(queue_to_delete_from, index);
111   }
112 }
```

4.11.1.2 delete_from_queue()

```
void delete_from_queue (
            int queue_id,
            int pid )
```

Deletes a PCB from the specified queue based on its PID.

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).

Definition at line 88 of file kern_sys_calls.c.

```
88                                               {
89   Vec* queue = NULL;
90   if (queue_id == 0) {
91     queue = &zero_priority_queue;
92   } else if (queue_id == 1) {
93     queue = &one_priority_queue;
94   } else {
95     queue = &two_priority_queue;
96   }
97
98   int index = determine_index_in_queue(queue, pid);
99   if (index != -1) {
100     vec_erase_no_deletor(queue, index);
101   }
102 }
```

4.11.1.3 determine_index_in_queue()

```
int determine_index_in_queue (
            Vec * queue,
            int pid )
```

Determines the index of a PCB in a given queue.

Given a thread pid and Vec* queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

Definition at line 40 of file kern_sys_calls.c.

```
40                                              {
41   for (int i = 0; i < vec_len(queue); i++) {
42     pcb_t* curr_pcb = vec_get(queue, i);
43     if (curr_pcb->pid == pid) {
44       return i;
45     }
46   }
47
48   return -1;  // not found
49 }
```

4.11.1.4 init_func()

```
void* init_func (
            void * input )
```

The function that runs the shell process.

The init process function. It spawns the shell process and reaps zombie children.

Definition at line 117 of file kern_sys_calls.c.

```
117                            {
118   char* shell_argv[] = {"shell", NULL};
119   s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
120
121   // continuously wait for and reap zombie children
122   while (true) {
123     int status;
124     s_waitpid(-1, &status, false);
125   }
126
127   return NULL;  // should never reach
128 }
```

4.11.1.5 move_pcb_correct_queue()

```
void move_pcb_correct_queue (
            int prev_priority,
            int new_priority,
            pcb_t * curr_pcb )
```

Moves a PCB from its previous priority queue to its new priority queue.

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

Definition at line 54 of file kern_sys_calls.c.

```
56                                         {
57   Vec* prev_queue;
58   Vec* new_queue;
59
60   if (prev_priority == 0) {
61     prev_queue = &zero_priority_queue;
62   } else if (prev_priority == 1) {
63     prev_queue = &one_priority_queue;
64   } else {
65     prev_queue = &two_priority_queue;
66   }
67
68   if (new_priority == 0) {
69     new_queue = &zero_priority_queue;
70   } else if (new_priority == 1) {
71     new_queue = &one_priority_queue;
72   } else {
73     new_queue = &two_priority_queue;
74   }
75
76   // delete from prev_queue, if it's present at all
77   int ind = determine_index_in_queue(prev_queue, curr_pcb->pid);
78   if (ind != -1) {
79     vec_erase_no_deletor(prev_queue, ind);
80   }
81
82   vec_push_back(new_queue, curr_pcb);
83 }
```

4.11.1.6 s_cleanup_init_process()

```
void s_cleanup_init_process ( )
```

Cleans up Init's resources.

Wrapper system-level function to be called in pennos's main method to clean up the init process.

Definition at line 157 of file kern_sys_calls.c.

```
157                              {
158   k_proc_cleanup(get_pcb_in_queue(&current_pcbs, 1));
159 }
```

4.11.1.7 s_echo()

```
void* s_echo (
            void * arg )
```

System-level wrapper for the shell built-in command "echo".

Parameters

arg	the pass along arguments to the u_echo function

Returns

NULL, dummy return value

Definition at line 358 of file kern_sys_calls.c.

```
                              {
  char** argv = (char**)arg;
  if (argv[1] == NULL) {  // no args case
    s_exit();
    return NULL;
  }

  int i = 1;                    // words after "echo"
  while (argv[i] != NULL) {  // while the arg isn't NULL
    s_write(current_running_pcb->output_fd, argv[i], strlen(argv[i]));
    s_write(current_running_pcb->output_fd, " ", 1);
    i++;
  }

  s_write(current_running_pcb->output_fd, "\n", 1);
  return NULL;
}
```

4.11.1.8 s_exit()

```
void s_exit (
        void )
```

Exits the current process and cleans up its resources.

Unconditionally exit the calling process.

Definition at line 293 of file kern_sys_calls.c.

```
293                 {
294   // Set process state to zombie
295   current_running_pcb->process_state = 'Z';
296   current_running_pcb->process_status = 20;  // EXITED_NORMALLY
297 
298   // Log the exit
299   log_generic_event('E', current_running_pcb->pid,
300                     current_running_pcb->priority,
301                     current_running_pcb->cmd_str);
302 
303   delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
304 
305   log_generic_event('Z', current_running_pcb->pid,
306                     current_running_pcb->priority,
307                     current_running_pcb->cmd_str);
308 }
```

4.11.1.9 s_kill()

```
int s_kill (
        pid_t pid,
        int signal )
```

Sends a signal to a process with specified pid.

Send a signal to a particular process.

Definition at line 279 of file kern_sys_calls.c.

```
279                                 {
280   pcb_t* pcb_with_pid = get_pcb_in_queue(&current_pcbs, pid);
281   if (pcb_with_pid == NULL) {
282     return -1;  // pid not found case
283   }
284 
285   pcb_with_pid->signals[signal] = true;  // signal flagged
286   log_generic_event('S', pid, pcb_with_pid->priority, pcb_with_pid->cmd_str);
287   return 0;
288 }
```

4.11.1.10 s_nice()

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

Sets the priority of a process with specified pid.

Set the priority of the specified thread.

Definition at line 313 of file kern_sys_calls.c.

```
313                                     {
314   if (priority < 0 || priority > 2) {  // error check
315     return -1;
316   }
317
318   pcb_t* curr_pcb = get_pcb_in_queue(&current_pcbs, pid);
319   if (curr_pcb != NULL) {  // found + exists
320     move_pcb_correct_queue(curr_pcb->priority, priority, curr_pcb);
321     log_nice_event(pid, curr_pcb->priority, priority, curr_pcb->cmd_str);
322     curr_pcb->priority = priority;
323     return 0;
324   }
325
326   return -1;  // pid not found
327 }
```

4.11.1.11 s_ps()

```
void* s_ps (
            void * arg )
```

System-level wrapper for the shell built-in command "ps".

Parameters

arg	the pass along arguments to the u_ps function

Returns

NULL, dummy return value

Definition at line 379 of file kern_sys_calls.c.

```
379                         {
380   char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
381   s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top));
382   for (int i = 0; i < vec_len(&current_pcbs); i++) {
383     pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);
384     char buffer[100];
385     snprintf(buffer, sizeof(buffer), "%d\t%d\t%d\t%c\t%s\n", curr_pcb->pid,
386              curr_pcb->par_pid, curr_pcb->priority, curr_pcb->process_state,
387              curr_pcb->cmd_str);
388     s_write(current_running_pcb->output_fd, buffer, strlen(buffer));
389   }
390   return NULL;
391 }
```

4.11.1.12 s_sleep()

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

Suspends the current process for a specified number of ticks.

Suspends execution of the calling proces for a specified number of clock ticks.

Definition at line 332 of file kern_sys_calls.c.

```
332 {
333   if (ticks <= 0) {
334     P_ERRNO = P_EINVAL;
335     return;
336   }
337
338   // block current process, set state to sleep
339   current_running_pcb->process_state = 'B';
340   current_running_pcb->is_sleeping = true;
341   current_running_pcb->time_to_wake = tick_counter + ticks;
342   log_generic_event('B', current_running_pcb->pid,
343                     current_running_pcb->priority,
344                     current_running_pcb->cmd_str);
345   if (spthread_suspend(current_running_pcb->thread_handle) != 0) { // give scheduler control
346     perror("Error in spthread_suspend in s_sleep call");
347   }
348 }
```

4.11.1.13 s_spawn()

```
pid_t s_spawn (
            void *(*)(void *) func,
            char * argv[],
            int fd0,
            int fd1 )
```

Spawns a child process with the given function and arguments.

Create a child process that executes the function `func`. The child will retain some attributes of the parent.

Definition at line 164 of file kern_sys_calls.c.

```
164 {
165   pcb_t* child;
166   if (strcmp(argv[0], "shell") == 0) {
167     child = k_proc_create(current_running_pcb, 0);
168   } else {
169     child = k_proc_create(current_running_pcb, 1);
170   }
171
172   if (child == NULL) {
173     P_ERRNO = P_ENULL;
174     return -1;
175   }
176
177   spthread_t thread_handle;
178
179   if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
180     perror("Error in spthread_create in s_spawn call");
181   }
182
183   child->cmd_str = strdup(argv[0]);
184   child->thread_handle = thread_handle;
185   child->input_fd = fd0;
186   child->output_fd = fd1;
187   child->fd_table[0] = fd0;
188   child->fd_table[1] = fd1;
189
190   log_generic_event('C', child->pid, child->priority, child->cmd_str);
191
192   return child->pid;
193 }
```

4.11.1.14 s_spawn_init()

```
pid_t s_spawn_init ( )
```

Creates the init process and spawns the shell process.

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

Definition at line 137 of file kern_sys_calls.c.

```
137                       {
138   pcb_t* init = k_proc_create(NULL, 0);
139   if (init == NULL) {
140     P_ERRNO = P_ENULL;
141     return -1;
142   }
143
144   spthread_t thread_handle;
145   if (spthread_create(&thread_handle, NULL, init_func, NULL) != 0) {
146     perror("Error in spthread_create in s_spawn_init call");
147   }
148
149   init->cmd_str = strdup("init");
150   init->thread_handle = thread_handle;
151   return init->pid;
152 }
```

4.11.1.15 s_waitpid()

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

Waits for a child of the calling process.

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.

Definition at line 198 of file kern_sys_calls.c.

```
198                                                    {
199   pcb_t* parent = current_running_pcb;
200   if (parent == NULL) {
201     return -1;
202   }
203
204   // if no children, return -1
205   bool has_child = false;
206   for (int i = 0; i < vec_len(&current_pcbs); i++) {
207     pcb_t* child = vec_get(&current_pcbs, i);
208     if (child->par_pid == parent->pid) {
209       has_child = true;
210       break;
211     }
212   }
213   if (!has_child) {
214     return -1;
215   }
216
217   // Scan the zombie queue first for terminated children.
218   for (int i = 0; i < vec_len(&zombie_queue); i++) {
219     pcb_t* child = vec_get(&zombie_queue, i);
220     if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
221       if (wstatus != NULL) {
222         *wstatus = child->process_status;
223       }
224       log_generic_event('W', child->pid, child->priority, child->cmd_str);
225       vec_erase_no_deletor(&zombie_queue, i);
226       delete_from_explicit_queue(&parent->child_pcbs, child->pid);
227       k_proc_cleanup(child);
```

```
      return child->pid;
    }
  }

  // If nohang is true, return immediately if no child has exited
  if (nohang) {
    return 0;
  }

  // Block the parent until a child exits
  delete_from_queue(parent->priority, parent->pid);
  parent->process_state = 'B';
  log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);

  while (true) {
    // Scan the zombie queue first for terminated children.
    for (int i = 0; i < vec_len(&zombie_queue); i++) {
      pcb_t* child = vec_get(&zombie_queue, i);
      if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
        if (wstatus != NULL) {
          *wstatus = child->process_status;
        }
        log_generic_event('W', child->pid, child->priority, child->cmd_str);
        vec_erase_no_deletor(&zombie_queue, i);
        delete_from_explicit_queue(&parent->child_pcbs, child->pid);
        k_proc_cleanup(child);
        return child->pid;
      }
    }

    // scan children of current running process for non-terminated state changes
    for (int i = 0; i < vec_len(&parent->child_pcbs); i++) {
      pcb_t* child = vec_get(&parent->child_pcbs, i);
      if ((pid == -1 || child->pid == pid) && (child->process_status == 21 || child->process_status ==
 23)) { // signaled
        if (wstatus != NULL) {
          *wstatus = child->process_status;
        }
        log_generic_event('W', child->pid, child->priority, child->cmd_str);
        child->process_status = 0; // reset status
        return child->pid;
      }
    }
  }

  // If we get here, something went wrong
  return -1;
}
```

4.11.2 Variable Documentation

4.11.2.1 current_fg_pid

`pid_t current_fg_pid = 2`

Definition at line 31 of file kern_sys_calls.c.

4.11.2.2 current_pcbs

`Vec current_pcbs [extern]`

Definition at line 30 of file scheduler.c.

4.11.2.3 current_running_pcb

```
pcb_t* current_running_pcb [extern]
```

Definition at line 38 of file scheduler.c.

4.11.2.4 one_priority_queue

```
Vec one_priority_queue [extern]
```

Definition at line 25 of file scheduler.c.

4.11.2.5 sleep_blocked_queue

```
Vec sleep_blocked_queue [extern]
```

Definition at line 28 of file scheduler.c.

4.11.2.6 tick_counter

```
int tick_counter [extern]
```

Definition at line 35 of file scheduler.c.

4.11.2.7 two_priority_queue

```
Vec two_priority_queue [extern]
```

Definition at line 26 of file scheduler.c.

4.11.2.8 zero_priority_queue

```
Vec zero_priority_queue [extern]
```

Definition at line 24 of file scheduler.c.

4.11.2.9 zombie_queue

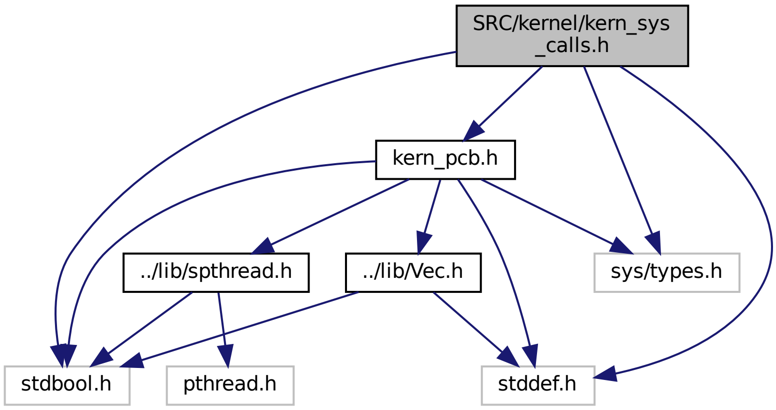
```
Vec zombie_queue [extern]
```

Definition at line 27 of file scheduler.c.

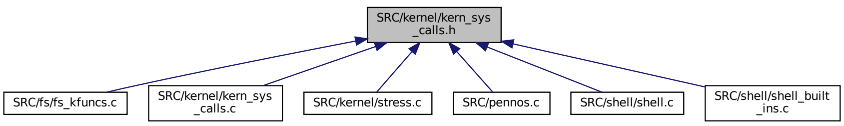
4.12 SRC/kernel/kern_sys_calls.h File Reference

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

Include dependency graph for kern_sys_calls.h:



This graph shows which files directly or indirectly include this file:



Functions

- int determine_index_in_queue (Vec ∗queue, int pid)

  Given a thread pid and Vec∗ queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.
- void move_pcb_correct_queue (int prev_priority, int new_priority, pcb_t ∗curr_pcb)

  Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

- void delete_from_queue (int queue_id, int pid)

  Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).
- void delete_from_explicit_queue (Vec *queue_to_delete_from, int pid)

  Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.
- void * init_func (void *input)

  The init process function. It spawns the shell process and reaps zombie children.
- pid_t s_spawn_init ()

  Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.
- void s_cleanup_init_process ()

  Wrapper system-level function to be called in pennos's main method to clean up the init process.
- 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_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_kill (pid_t pid, int signal)

  Send a signal to a particular process.
- void s_exit (void)

  Unconditionally exit the calling process.
- int s_nice (pid_t pid, int priority)

  Set the priority of the specified thread.
- void s_sleep (unsigned int ticks)

  Suspends execution of the calling proces for a specified number of clock ticks.
- void * s_echo (void *arg)

  System-level wrapper for the shell built-in command "echo".
- void * s_ps (void *arg)

  System-level wrapper for the shell built-in command "ps".

4.12.1 Function Documentation

4.12.1.1 delete_from_explicit_queue()

```
void delete_from_explicit_queue (
            Vec * queue_to_delete_from,
            int pid )
```

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.

Parameters

queue_to_delete_from	ptr to Vec* queue to delete from
pid	the pid of the PCB to delete

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.

Definition at line 107 of file kern_sys_calls.c.

```
107                                                                     {
108   int index = determine_index_in_queue(queue_to_delete_from, pid);
109   if (index != -1) {
110     vec_erase_no_deletor(queue_to_delete_from, index);
111   }
112 }
```

4.12.1.2 delete_from_queue()

```
void delete_from_queue (
               int queue_id,
               int pid )
```

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).

Parameters

queue_id	An integer representing the queue: 0 for zero_priority_queue, 1 for one_priority_queue, or 2 for two_priority_queue.
pid	The PID of the PCB to be removed.

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).

Definition at line 88 of file kern_sys_calls.c.

```
88                                                {
89   Vec* queue = NULL;
90   if (queue_id == 0) {
91     queue = &zero_priority_queue;
92   } else if (queue_id == 1) {
93     queue = &one_priority_queue;
94   } else {
95     queue = &two_priority_queue;
96   }
97
98   int index = determine_index_in_queue(queue, pid);
99   if (index != -1) {
100     vec_erase_no_deletor(queue, index);
101   }
102 }
```

4.12.1.3 determine_index_in_queue()

```
int determine_index_in_queue (
               Vec * queue,
               int pid )
```

Given a thread pid and Vec∗ queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

Parameters

queue	pointer to the vector queue that may contain the thread/pid
pid	the thread's pid

Returns

the index of the thread/pid in the queue, or -1 if not found

Given a thread pid and Vec∗ queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

Definition at line 40 of file kern_sys_calls.c.

```
40                                                    {
41   for (int i = 0; i < vec_len(queue); i++) {
42     pcb_t* curr_pcb = vec_get(queue, i);
43     if (curr_pcb->pid == pid) {
44       return i;
45     }
46   }
47
48   return -1;  // not found
49 }
```

4.12.1.4 init_func()

```
void* init_func (
            void * input )
```

The init process function. It spawns the shell process and reaps zombie children.

Parameters

input	unused but needed for typing reasons

Returns

irrelvant return value because never supposed to return

The init process function. It spawns the shell process and reaps zombie children.

Definition at line 117 of file kern_sys_calls.c.

```
117                             {
118   char* shell_argv[] = {"shell", NULL};
119   s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
120
121   // continuously wait for and reap zombie children
122   while (true) {
123     int status;
124     s_waitpid(-1, &status, false);
125   }
126
127   return NULL;  // should never reach
128 }
```

4.12.1.5 move_pcb_correct_queue()

```
void move_pcb_correct_queue (
            int prev_priority,
            int new_priority,
            pcb_t * curr_pcb )
```

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

Parameters

prev_priority	thread's previous priority
new_priority	thread's new priority
curr_pcb	pointer to the thread's PCB

Precondition

assumes the prev_priority and new_priority falls in integers [0, 2]

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

Definition at line 54 of file kern_sys_calls.c.

```
                                              {
  Vec* prev_queue;
  Vec* new_queue;

  if (prev_priority == 0) {
    prev_queue = &zero_priority_queue;
  } else if (prev_priority == 1) {
    prev_queue = &one_priority_queue;
  } else {
    prev_queue = &two_priority_queue;
  }

  if (new_priority == 0) {
    new_queue = &zero_priority_queue;
  } else if (new_priority == 1) {
    new_queue = &one_priority_queue;
  } else {
    new_queue = &two_priority_queue;
  }

  // delete from prev_queue, if it's present at all
  int ind = determine_index_in_queue(prev_queue, curr_pcb->pid);
  if (ind != -1) {
    vec_erase_no_deletor(prev_queue, ind);
  }

  vec_push_back(new_queue, curr_pcb);
}
```

4.12.1.6 s_cleanup_init_process()

```
void s_cleanup_init_process ( )
```

Wrapper system-level function to be called in pennos's main method to clean up the init process.

Wrapper system-level function to be called in pennos's main method to clean up the init process.

Definition at line 157 of file kern_sys_calls.c.

```
157                                  {
158   k_proc_cleanup(get_pcb_in_queue(&current_pcbs, 1));
159 }
```

4.12.1.7 s_echo()

```
void* s_echo (
            void * arg )
```

System-level wrapper for the shell built-in command "echo".

Parameters

arg	the pass along arguments to the u_echo function

Returns

NULL, dummy return value

Definition at line 358 of file kern_sys_calls.c.

```
358                      {
359   char** argv = (char**)arg;
360   if (argv[1] == NULL) {  // no args case
361     s_exit();
362     return NULL;
363   }
364
365   int i = 1;                   // words after "echo"
366   while (argv[i] != NULL) {  // while the arg isn't NULL
367     s_write(current_running_pcb->output_fd, argv[i], strlen(argv[i]));
368     s_write(current_running_pcb->output_fd, " ", 1);
369     i++;
370   }
371
372   s_write(current_running_pcb->output_fd, "\n", 1);
373   return NULL;
374 }
```

4.12.1.8 s_exit()

```
void s_exit (
            void )
```

Unconditionally exit the calling process.

Unconditionally exit the calling process.

Definition at line 293 of file kern_sys_calls.c.

```
293                {
294   // Set process state to zombie
295   current_running_pcb->process_state = 'Z';
296   current_running_pcb->process_status = 20;  // EXITED_NORMALLY
297
298   // Log the exit
299   log_generic_event('E', current_running_pcb->pid,
300                     current_running_pcb->priority,
301                     current_running_pcb->cmd_str);
302
303   delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
304
305   log_generic_event('Z', current_running_pcb->pid,
306                     current_running_pcb->priority,
307                     current_running_pcb->cmd_str);
308 }
```

4.12.1.9 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 0 = P_SIGSTOP, 1 = P_SIGCONT, 2 = P_SIGTERM

Returns

0 on success, -1 on error.

Send a signal to a particular process.

Definition at line 279 of file kern_sys_calls.c.

```
279                                     {
280   pcb_t* pcb_with_pid = get_pcb_in_queue(&current_pcbs, pid);
281   if (pcb_with_pid == NULL) {
282     return -1;  // pid not found case
283   }
284
285   pcb_with_pid->signals[signal] = true;  // signal flagged
286   log_generic_event('S', pid, pcb_with_pid->priority, pcb_with_pid->cmd_str);
287   return 0;
288 }
```

4.12.1.10 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.

Set the priority of the specified thread.

Definition at line 313 of file kern_sys_calls.c.

```
313                                       {
314   if (priority < 0 || priority > 2) {  // error check
315     return -1;
316   }
317
318   pcb_t* curr_pcb = get_pcb_in_queue(&current_pcbs, pid);
319   if (curr_pcb != NULL) {  // found + exists
320     move_pcb_correct_queue(curr_pcb->priority, priority, curr_pcb);
321     log_nice_event(pid, curr_pcb->priority, priority, curr_pcb->cmd_str);
322     curr_pcb->priority = priority;
323     return 0;
324   }
325
326   return -1;  // pid not found
327 }
```

4.12.1.11 s_ps()

```
void* s_ps (
            void * arg )
```

System-level wrapper for the shell built-in command "ps".

Parameters

arg	the pass along arguments to the u_ps function

Returns

NULL, dummy return value

Definition at line 379 of file kern_sys_calls.c.

```
379                      {
380   char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
381   s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top));
382   for (int i = 0; i < vec_len(&current_pcbs); i++) {
383     pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);
384     char buffer[100];
385     snprintf(buffer, sizeof(buffer), "%d\t%d\t%d\t%c\t%s\n", curr_pcb->pid,
386              curr_pcb->par_pid, curr_pcb->priority, curr_pcb->process_state,
387              curr_pcb->cmd_str);
388     s_write(current_running_pcb->output_fd, buffer, strlen(buffer));
389   }
390   return NULL;
391 }
```

4.12.1.12 s_sleep()

```
void 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.

Suspends execution of the calling proces for a specified number of clock ticks.

Definition at line 332 of file kern_sys_calls.c.

```
332                                 {
333   if (ticks <= 0) {
334     P_ERRNO = P_EINVAL;
335     return;
336   }
337
338   // block current process, set state to sleep
339   current_running_pcb->process_state = 'B';
340   current_running_pcb->is_sleeping = true;
```

```
  current_running_pcb->time_to_wake = tick_counter + ticks;
  log_generic_event('B', current_running_pcb->pid,
                    current_running_pcb->priority,
                    current_running_pcb->cmd_str);
  if (spthread_suspend(current_running_pcb->thread_handle) != 0) { // give scheduler control
    perror("Error in spthread_suspend in s_sleep call");
  }
}
```

4.12.1.13 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 or -1 on error

Create a child process that executes the function `func`. The child will retain some attributes of the parent.

Definition at line 164 of file kern_sys_calls.c.

```
164                                                                         {
165   pcb_t* child;
166   if (strcmp(argv[0], "shell") == 0) {
167     child = k_proc_create(current_running_pcb, 0);
168   } else {
169     child = k_proc_create(current_running_pcb, 1);
170   }
171
172   if (child == NULL) {
173     P_ERRNO = P_ENULL;
174     return -1;
175   }
176
177   spthread_t thread_handle;
178
179   if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
180     perror("Error in spthread_create in s_spawn call");
181   }
182
183   child->cmd_str = strdup(argv[0]);
184   child->thread_handle = thread_handle;
185   child->input_fd = fd0;
186   child->output_fd = fd1;
187   child->fd_table[0] = fd0;
188   child->fd_table[1] = fd1;
189
190   log_generic_event('C', child->pid, child->priority, child->cmd_str);
191
192   return child->pid;
193 }
```

4.12.1.14 s_spawn_init()

```
pid_t s_spawn_init ( )
```

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

Returns

the pid_t of the created process on success or -1 on error

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

Definition at line 137 of file kern_sys_calls.c.

```
137                           {
138   pcb_t* init = k_proc_create(NULL, 0);
139   if (init == NULL) {
140     P_ERRNO = P_ENULL;
141     return -1;
142   }
143
144   spthread_t thread_handle;
145   if (spthread_create(&thread_handle, NULL, init_func, NULL) != 0) {
146     perror("Error in spthread_create in s_spawn_init call");
147   }
148
149   init->cmd_str = strdup("init");
150   init->thread_handle = thread_handle;
151   return init->pid;
152 }
```

4.12.1.15 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.

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.

Definition at line 198 of file kern_sys_calls.c.

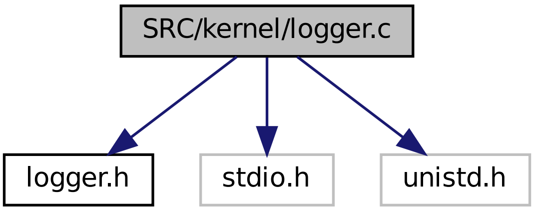
```
                                                    {
  pcb_t* parent = current_running_pcb;
  if (parent == NULL) {
    return -1;
  }

  // if no children, return -1
  bool has_child = false;
  for (int i = 0; i < vec_len(&current_pcbs); i++) {
    pcb_t* child = vec_get(&current_pcbs, i);
    if (child->par_pid == parent->pid) {
      has_child = true;
      break;
    }
  }
  if (!has_child) {
    return -1;
  }

  // Scan the zombie queue first for terminated children.
  for (int i = 0; i < vec_len(&zombie_queue); i++) {
    pcb_t* child = vec_get(&zombie_queue, i);
    if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
      if (wstatus != NULL) {
        *wstatus = child->process_status;
      }
      log_generic_event('W', child->pid, child->priority, child->cmd_str);
      vec_erase_no_deletor(&zombie_queue, i);
      delete_from_explicit_queue(&parent->child_pcbs, child->pid);
      k_proc_cleanup(child);
      return child->pid;
    }
  }

  // If nohang is true, return immediately if no child has exited
  if (nohang) {
    return 0;
  }

  // Block the parent until a child exits
  delete_from_queue(parent->priority, parent->pid);
  parent->process_state = 'B';
  log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);

  while (true) {
    // Scan the zombie queue first for terminated children.
    for (int i = 0; i < vec_len(&zombie_queue); i++) {
      pcb_t* child = vec_get(&zombie_queue, i);
      if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
        if (wstatus != NULL) {
          *wstatus = child->process_status;
        }
        log_generic_event('W', child->pid, child->priority, child->cmd_str);
        vec_erase_no_deletor(&zombie_queue, i);
        delete_from_explicit_queue(&parent->child_pcbs, child->pid);
        k_proc_cleanup(child);
        return child->pid;
      }
    }

    // scan children of current running process for non-terminated state changes
    for (int i = 0; i < vec_len(&parent->child_pcbs); i++) {
      pcb_t* child = vec_get(&parent->child_pcbs, i);
      if ((pid == -1 || child->pid == pid) && (child->process_status == 21 || child->process_status ==
 23)) { // signaled
        if (wstatus != NULL) {
          *wstatus = child->process_status;
        }
        log_generic_event('W', child->pid, child->priority, child->cmd_str);
        child->process_status = 0; // reset status
        return child->pid;
      }
    }
  }

  // If we get here, something went wrong
  return -1;
}
```

4.13 SRC/kernel/logger.c File Reference

```
#include "logger.h"
```

```
#include <stdio.h>
#include <unistd.h>
```

Include dependency graph for logger.c:



Functions

- void log_scheduling_event (int pid, int queue_num, char ∗process_name)

  Logs a scheduling event i.e. the scheduling of a process for this clock tick.
- void log_generic_event (char event_type, int pid, int nice_value, char ∗process_name)

  Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)
- void log_nice_event (int pid, int old_nice_value, int new_nice_value, char ∗process_name)

  Logs a nice event, which is the adjusting of a process's nice value.

4.13.1 Function Documentation

4.13.1.1 log_generic_event()

```
void log_generic_event (
            char event_type,
            int pid,
            int nice_value,
            char * process_name )
```

Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)

Parameters

event_type	the type of event, defined by: 'C' = CREATE, 'S' = SIGNALED, 'E' = EXITED, 'Z' = ZOMBIE, 'O' = ORPHAN, 'W' = WAITED 'B' = BLOCKED, 'U' = UNBLOCKED 's' = STOPPED, 'c' = CONTINUED (notably lower-cased)
pid	process pid
nice_value	process nice value
process_name	string containing process name

Precondition

assumes event_type matches one of the above characters

Postcondition

will perror if the write fails

Definition at line 14 of file logger.c.

```
14 {
15     char* operation;
16
17     switch(event_type) {
18         case 'C':
19             operation = "CREATE";
20             break;
21         case 'S':
22             operation = "SIGNALED";
23             break;
24         case 'E':
25             operation = "EXITED";
26             break;
27         case 'Z':
28             operation = "ZOMBIE";
29             break;
30         case 'O':
31             operation = "ORPHAN";
32             break;
33         case 'W':
34             operation = "WAITED";
35             break;
36         case 'B':
37             operation = "BLOCKED";
38             break;
39         case 'U':
40             operation = "UNBLOCKED";
41             break;
42         case 's':
43             operation = "STOPPED";
44             break;
45         default:
46             operation = "CONTINUED";
47             break;
48     }
49
50     char buffer[200];
51     int str_len = snprintf(buffer, sizeof(buffer), "[%d]\t%s\t%d\t%d\t%s\n", tick_counter, operation,
       pid, nice_value, process_name);
52     if (write(log_fd, buffer, str_len) == -1) {
53         perror("error in writing to the log file for generic event");
54     }
55 }
```

4.13.1.2 log_nice_event()

```
void log_nice_event (
            int pid,
            int old_nice_value,
            int new_nice_value,
            char * process_name )
```

Logs a nice event, which is the adjusting of a process's nice value.

Parameters

pid	process pid
old_nice_value	old nice value
new_nice_value	new nice value
process_name	string containing process name

Postcondition

will perror if the write fails

Definition at line 57 of file logger.c.

```
57                                                                                          {
58     char buffer[200];
59     int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tNICE\t%d\t%d\t%d\t%s\n", tick_counter, pid,
       old_nice_value, new_nice_value, process_name);
60     if (write(log_fd, buffer, str_len) == -1) {
61         perror("error in writing to the log file for nice event");
62     }
63 }
```

4.13.1.3 log_scheduling_event()

```
void log_scheduling_event (
            int pid,
            int queue_num,
            char * process_name )
```

Logs a scheduling event i.e. the scheduling of a process for this clock tick.

Parameters

pid	pid of the process being scheduled
queue_num	the priority queue num of the process
process_name	string containing scheduled process's name

Postcondition

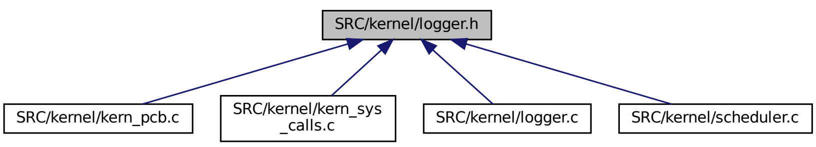
will perror if the write fails

Definition at line 6 of file logger.c.

```
6                                                                   {
7     char buffer[200];
8     int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tSCHEDULE\t%d\t%d\t%s\n", tick_counter, pid,
       queue_num, process_name);
9     if (write(log_fd, buffer, str_len) == -1) {
10         perror("error in writing to the log file for scheduling event");
11     }
12 }
```

4.14 SRC/kernel/logger.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void log_scheduling_event (int pid, int queue_num, char *process_name)

  Logs a scheduling event i.e. the scheduling of a process for this clock tick.
- void log_generic_event (char event_type, int pid, int nice_value, char *process_name)

  Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)
- void log_nice_event (int pid, int old_nice_value, int new_nice_value, char *process_name)

  Logs a nice event, which is the adjusting of a process's nice value.

Variables

- int tick_counter
- int log_fd

4.14.1 Function Documentation

4.14.1.1 log_generic_event()

```
void log_generic_event (
            char event_type,
            int pid,
            int nice_value,
            char * process_name )
```

Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)

Parameters

event_type	the type of event, defined by: 'C' = CREATE, 'S' = SIGNALED, 'E' = EXITED, 'Z' = ZOMBIE, 'O' = ORPHAN, 'W' = WAITED 'B' = BLOCKED, 'U' = UNBLOCKED 's' = STOPPED, 'c' = CONTINUED (notably lower-cased)
pid	process pid
nice_value	process nice value
process_name	string containing process name

Precondition

assumes event_type matches one of the above characters

Postcondition

will perror if the write fails

Definition at line 14 of file logger.c.

```
{
    char* operation;

    switch(event_type) {
        case 'C':
            operation = "CREATE";
            break;
        case 'S':
            operation = "SIGNALED";
            break;
        case 'E':
            operation = "EXITED";
            break;
        case 'Z':
            operation = "ZOMBIE";
            break;
        case 'O':
            operation = "ORPHAN";
            break;
        case 'W':
            operation = "WAITED";
            break;
        case 'B':
            operation = "BLOCKED";
            break;
        case 'U':
            operation = "UNBLOCKED";
            break;
        case 's':
            operation = "STOPPED";
            break;
        default:
            operation = "CONTINUED";
            break;
    }

    char buffer[200];
    int str_len = snprintf(buffer, sizeof(buffer), "[%d]\t%s\t%d\t%d\t%s\n", tick_counter, operation,
    pid, nice_value, process_name);
    if (write(log_fd, buffer, str_len) == -1) {
        perror("error in writing to the log file for generic event");
    }
}
```

4.14.1.2 log_nice_event()

```
void log_nice_event (
            int pid,
            int old_nice_value,
            int new_nice_value,
            char * process_name )
```

Logs a nice event, which is the adjusting of a process's nice value.

Parameters

pid	process pid
old_nice_value	old nice value
new_nice_value	new nice value
process_name	string containing process name

Postcondition

will perror if the write fails

Definition at line 57 of file logger.c.

```
{
    char buffer[200];
    int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tNICE\t%d\t%d\t%d\t%s\n", tick_counter, pid,
    old_nice_value, new_nice_value, process_name);
    if (write(log_fd, buffer, str_len) == -1) {
        perror("error in writing to the log file for nice event");
    }
}
```

4.14.1.3 log_scheduling_event()

```
void log_scheduling_event (
            int pid,
            int queue_num,
            char * process_name )
```

Logs a scheduling event i.e. the scheduling of a process for this clock tick.

Parameters

pid	pid of the process being scheduled
queue_num	the priority queue num of the process
process_name	string containing scheduled process's name

Postcondition

will perror if the write fails

Definition at line 6 of file logger.c.

```
6  {
7      char buffer[200];
8      int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tSCHEDULE\t%d\t%d\t%s\n", tick_counter, pid,
       queue_num, process_name);
9      if (write(log_fd, buffer, str_len) == -1) {
10         perror("error in writing to the log file for scheduling event");
11     }
12 }
```

4.14.2 Variable Documentation

4.14.2.1 log_fd

```
int log_fd [extern]
```

Definition at line 36 of file scheduler.c.

4.14.2.2 tick_counter

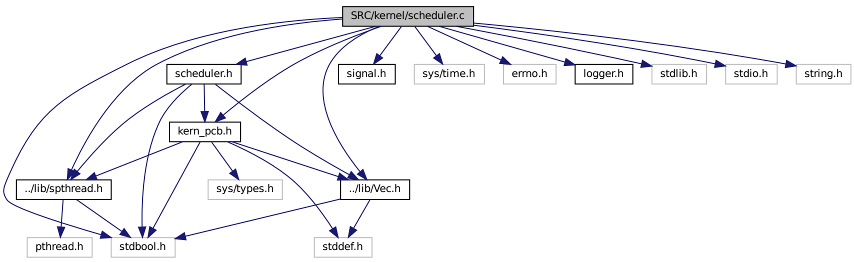
```
int tick_counter  [extern]
```

Definition at line 35 of file scheduler.c.

4.15 SRC/kernel/scheduler.c File Reference

```
#include "scheduler.h"
#include <signal.h>
#include <stdbool.h>
#include <sys/time.h>
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include "errno.h"
#include "kern_pcb.h"
#include "logger.h"
#include "stdlib.h"
#include <stdio.h>
#include <string.h>
```

Include dependency graph for scheduler.c:



Functions

- void initialize_scheduler_queues ()

  Initializes the scheduler queues.
- void free_scheduler_queues ()

  Frees the scheduler queues.
- int generate_next_priority ()

  Generates the next priority for scheduling based on the defined probabilities.
- pcb_t ∗ get_next_pcb (int priority)

  Gets the next PCB from the specified priority queue.
- void put_pcb_into_correct_queue (pcb_t ∗pcb)

  Puts the given PCB into the correct queue based on its priority and state.
- void delete_process_from_particular_queue (pcb_t ∗pcb, Vec ∗queue)

  Deletes the given PCB from the specified queue.
- void delete_process_from_all_queues_except_current (pcb_t ∗pcb)

  Deletes the given PCB from all queues except the current one.
- void delete_process_from_all_queues (pcb_t ∗pcb)

Deletes the given PCB from all queues.
- pcb_t ∗ get_pcb_in_queue (Vec ∗queue, pid_t pid)

  Gets the PCB with the specified PID from the given queue.
- bool child_in_zombie_queue (pcb_t ∗parent)

  Checks if the given parent PCB has any children in the zombie queue.
- bool child_with_changed_process_status (pcb_t ∗parent)

  Checks if the given parent PCB has any children with a changed process status.
- void alarm_handler (int signum)

  Signal handler for SIGALRM.
- void handle_signal (pcb_t ∗pcb, int signal)

  Handles the specified signal for the given PCB.
- void s_shutdown_pennos (void)

  Shuts down the scheduler and cleans up resources.
- void scheduler ()

  The main scheduler function for PennOS.

Variables

- Vec zero_priority_queue
- Vec one_priority_queue
- Vec two_priority_queue
- Vec zombie_queue
- Vec sleep_blocked_queue
- Vec current_pcbs
- int tick_counter = 0
- int log_fd
- pcb_t ∗ current_running_pcb
- int curr_priority_arr_index = 0
- int det_priorities_arr [19]

4.15.1 Function Documentation

4.15.1.1 alarm_handler()

```
void alarm_handler (
            int signum )
```

Signal handler for SIGALRM.

Handles the alarm signal.

Definition at line 225 of file scheduler.c.

```
225                              {
226   tick_counter++;
227 }
```

4.15.1.2 child_in_zombie_queue()

```
bool child_in_zombie_queue (
            pcb_t * parent )
```

Checks if the given parent PCB has any children in the zombie queue.

Checks if a child of the given parent process is in the zombie queue.

Definition at line 199 of file scheduler.c.

```
199                                      {
200   for (int i = 0; i < vec_len(&zombie_queue); i++) {
201     pcb_t* child = vec_get(&zombie_queue, i);
202     if (child->par_pid == parent->pid) {
203       return true;
204     }
205   }
206   return false;
207 }
```

4.15.1.3 child_with_changed_process_status()

```
bool child_with_changed_process_status (
            pcb_t * parent )
```

Checks if the given parent PCB has any children with a changed process status.

Checks if a child of the given parent process has a changed process status.

Definition at line 212 of file scheduler.c.

```
212                                                  {
213   for (int i = 0; i < vec_len(&current_pcbs); i++) {
214     pcb_t* child = vec_get(&current_pcbs, i);
215     if (child->par_pid == parent->pid && child->process_status != 0) {
216       return true;
217     }
218   }
219   return false;
220 }
```

4.15.1.4 delete_process_from_all_queues()

```
void delete_process_from_all_queues (
            pcb_t * pcb )
```

Deletes the given PCB from all queues.

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 177 of file scheduler.c.

```
177                                            {
178   delete_process_from_all_queues_except_current(pcb);
179   delete_process_from_particular_queue(pcb, &current_pcbs);
180 }
```

4.15.1.5 delete_process_from_all_queues_except_current()

```
void delete_process_from_all_queues_except_current (
            pcb_t * pcb )
```

Deletes the given PCB from all queues except the current one.

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 166 of file scheduler.c.

```
166                                                              {
167   delete_process_from_particular_queue(pcb, &zero_priority_queue);
168   delete_process_from_particular_queue(pcb, &one_priority_queue);
169   delete_process_from_particular_queue(pcb, &two_priority_queue);
170   delete_process_from_particular_queue(pcb, &zombie_queue);
171   delete_process_from_particular_queue(pcb, &sleep_blocked_queue);
172 }
```

4.15.1.6 delete_process_from_particular_queue()

```
void delete_process_from_particular_queue (
            pcb_t * pcb,
            Vec * queue )
```

Deletes the given PCB from the specified queue.

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

Definition at line 153 of file scheduler.c.

```
153                                                                 {
154   for (int i = 0; i < vec_len(queue); i++) {
155     pcb_t* curr_pcb = vec_get(queue, i);
156     if (curr_pcb->pid == pcb->pid) {
157       vec_erase_no_deletor(queue, i);
158       return;
159     }
160   }
161 }
```

4.15.1.7 free_scheduler_queues()

```
void free_scheduler_queues ( )
```

Frees the scheduler queues.

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

Definition at line 66 of file scheduler.c.

```
66                              {
67   vec_destroy(&zero_priority_queue);
68   vec_destroy(&one_priority_queue);
69   vec_destroy(&two_priority_queue);
70   vec_destroy(&zombie_queue);
71   vec_destroy(&sleep_blocked_queue);
72   vec_destroy(&current_pcbs);
73 }
```

4.15.1.8 generate_next_priority()

```
int generate_next_priority ( )
```

Generates the next priority for scheduling based on the defined probabilities.

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

Definition at line 82 of file scheduler.c.

```
82                                 {
83   // check if all queues are empty
84   if (vec_is_empty(&zero_priority_queue) && vec_is_empty(&one_priority_queue) &&
85       vec_is_empty(&two_priority_queue)) {
86     return -1;
87   }
88 
89   int priorities_attempted = 0;
90   while (priorities_attempted < 19) {
91     int curr_pri = det_priorities_arr[curr_priority_arr_index];
92     curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
93     if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
94       priorities_attempted++;
95       return 0;
96     } else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
97       priorities_attempted++;
98       return 1;
99     } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
100       priorities_attempted++;
101       return 2;
102     }
103   }
104 
105   return -1;  // should never reach
106 }
```

4.15.1.9 get_next_pcb()

```
pcb_t* get_next_pcb (
               int priority )
```

Gets the next PCB from the specified priority queue.

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

Definition at line 111 of file scheduler.c.

```
111                                      {
112   if (priority == -1) {  // all queues empty
113     return NULL;
114   }
115 
116   pcb_t* next_pcb = NULL;
117   if (priority == 0) {
118     next_pcb = vec_get(&zero_priority_queue, 0);
119     vec_erase_no_deletor(&zero_priority_queue, 0);
120   } else if (priority == 1) {
121     next_pcb = vec_get(&one_priority_queue, 0);
122     vec_erase_no_deletor(&one_priority_queue, 0);
123   } else if (priority == 2) {
124     next_pcb = vec_get(&two_priority_queue, 0);
125     vec_erase_no_deletor(&two_priority_queue, 0);
126   }
127 
128   return next_pcb;
129 }
```

4.15.1.10 get_pcb_in_queue()

```
pcb_t* get_pcb_in_queue (
            Vec * queue,
            pid_t pid )
```

Gets the PCB with the specified PID from the given queue.

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

Definition at line 185 of file scheduler.c.

```
185                                                    {
186   for (int i = 0; i < vec_len(queue); i++) {
187     pcb_t* curr_pcb = vec_get(queue, i);
188     if (curr_pcb->pid == pid) {
189       return curr_pcb;
190     }
191   }
192 
193   return NULL;
194 }
```

4.15.1.11 handle_signal()

```
void handle_signal (
            pcb_t * pcb,
            int signal )
```

Handles the specified signal for the given PCB.

Handles a signal for a given process.

Definition at line 232 of file scheduler.c.

```
232                                               {
233   switch (signal) {
234     case 0:  // P_SIGSTOP
235       if (pcb->process_state == 'R' || pcb->process_state == 'B') {
236         pcb->process_state = 'S';
237         log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
238         delete_process_from_all_queues_except_current(pcb);
239         pcb->process_status = 21;  // STOPPED_BY_SIG
240       }
241       pcb->signals[0] = false;
242       break;
243     case 1:                               // P_SIGCONT
244       if (pcb->process_state == 'S') {  // Only continue if stopped
245         pcb->process_state = 'R';
246         log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
247         delete_process_from_all_queues_except_current(pcb);
248         put_pcb_into_correct_queue(pcb);
249         pcb->process_status = 23;  // Reset status
250       }
251       pcb->signals[1] = false;
252       break;
253     case 2:                               // P_SIGTERM
254       if (pcb->process_state != 'Z') {  // Don't terminate if already zombie
255         pcb->process_state = 'Z';
256         pcb->process_status = 22;  // TERM_BY_SIG
257         log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
258         delete_process_from_all_queues_except_current(pcb);
259         put_pcb_into_correct_queue(pcb);
260         pcb->process_status = 22;  // TERM_BY_SIG
261       }
262       pcb->signals[2] = false;
263       break;
264   }
265 }
```

4.15.1.12 initialize_scheduler_queues()

```
void initialize_scheduler_queues ( )
```

Initializes the scheduler queues.

Initializes the scheduler queues. This function should be called before any other scheduler functions are called.

Note

The deconstructors for the queues are set to NULL to prevent double freeing when exiting PennOS.

Definition at line 54 of file scheduler.c.

```
54                                         {
55   zero_priority_queue = vec_new(0, NULL);
56   one_priority_queue = vec_new(0, NULL);
57   two_priority_queue = vec_new(0, NULL);
58   zombie_queue = vec_new(0, NULL);
59   sleep_blocked_queue = vec_new(0, NULL);
60   current_pcbs = vec_new(0, free_pcb);
61 }
```

4.15.1.13 put_pcb_into_correct_queue()

```
void put_pcb_into_correct_queue (
            pcb_t * pcb )
```

Puts the given PCB into the correct queue based on its priority and state.

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

Definition at line 134 of file scheduler.c.

```
134                                              {
135   if (pcb->process_state == 'R') {
136     if (pcb->priority == 0) {
137       vec_push_back(&zero_priority_queue, pcb);
138     } else if (pcb->priority == 1) {
139       vec_push_back(&one_priority_queue, pcb);
140     } else if (pcb->priority == 2) {
141       vec_push_back(&two_priority_queue, pcb);
142     }
143   } else if (pcb->process_state == 'Z') {
144     vec_push_back(&zombie_queue, pcb);
145   } else if (pcb->process_state == 'B' || pcb->is_sleeping) {
146     vec_push_back(&sleep_blocked_queue, pcb);
147   }
148 }
```

4.15.1.14 s_shutdown_pennos()

```
void s_shutdown_pennos (
            void )
```

Shuts down the scheduler and cleans up resources.

Shuts down the PennOS scheduler.

Definition at line 270 of file scheduler.c.

```
270                              {
271   scheduling_done = true;
272 }
```

4.15.1.15 scheduler()

```
void scheduler ( )
```

The main scheduler function for PennOS.

This function manages process scheduling, signal handling, and timer-based preemption. It ensures that processes are executed based on their priority and handles signals for both the currently running process and other processes.

Definition at line 277 of file scheduler.c.

```
277                  {
278   int curr_priority_queue_num;
279 
280   // mask for while scheduler is waiting for alarm
281   sigset_t suspend_set;
282   sigfillset(&suspend_set);
283   sigdelset(&suspend_set, SIGALRM);
284 
285   // ensure sigarlm doesn't terminate the process
286   struct sigaction act = (struct sigaction){
287       .sa_handler = alarm_handler,
288       .sa_mask = suspend_set,
289       .sa_flags = SA_RESTART,
290   };
291   sigaction(SIGALRM, &act, NULL);
292 
293   // make sure SIGALRM is unblocked
294   sigset_t alarm_set;
295   sigemptyset(&alarm_set);
296   sigaddset(&alarm_set, SIGALRM);
297   pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
298 
299   struct itimerval it;
300   it.it_interval = (struct timeval){.tv_usec = hundred_millisec};
301   it.it_value = it.it_interval;
302   setitimer(ITIMER_REAL, &it, NULL);
303 
304   while (!scheduling_done) {
305     // handle signals for the currently running process
306     if (current_running_pcb != NULL) {
307       for (int i = 0; i < 3; i++) {
308         if (current_running_pcb->signals[i]) {
309           handle_signal(current_running_pcb, i);
310           // If process was terminated, don't continue scheduling it
311           if (current_running_pcb->process_state != 'R') {
312             current_running_pcb = NULL;
313             break;
314           }
315         }
316       }
317     }
318 
319     // handle signals for all other processes (currently running or not)
320     for (int i = 0; i < vec_len(&current_pcbs); i++) {
321       pcb_t* curr_pcb = vec_get(&current_pcbs, i);
322       for (int j = 0; j < 3; j++) {
323         if (curr_pcb->signals[j]) {
324           handle_signal(curr_pcb, j);
325         }
326       }
327     }
328 
329     // Check sleep/blocked queue to move processes back to scheduable queues
330     for (int i = 0; i < vec_len(&sleep_blocked_queue); i++) {
331       pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
332       bool make_runnable = false;
333       if (blocked_proc->is_sleeping &&
334           blocked_proc->time_to_wake == tick_counter) {
335         blocked_proc->is_sleeping = false;
336         blocked_proc->time_to_wake = -1;
337         blocked_proc->signals[2] = false;  // Unlikely, but reset signal
338         make_runnable = true;
339       } else if (blocked_proc->is_sleeping &&
340                  blocked_proc->signals[2]) {  // P_SIGTERM received
341         blocked_proc->is_sleeping = false;
342         blocked_proc->process_state = 'Z';
343         blocked_proc->process_status = 22;  // TERM_BY_SIG
344         blocked_proc->signals[2] = false;
345         delete_process_from_all_queues_except_current(blocked_proc);
346         put_pcb_into_correct_queue(blocked_proc);
347         log_generic_event('Z', blocked_proc->pid, blocked_proc->priority,
348                           blocked_proc->cmd_str);
```

```
        i--;
      } else if (child_in_zombie_queue(blocked_proc)) {
        make_runnable = true;
      } else if (child_with_changed_process_status(blocked_proc)) {
        make_runnable = true;
      }

      if (make_runnable) {
        blocked_proc->process_state = 'R';
        vec_erase_no_deletor(&sleep_blocked_queue, i);
        delete_process_from_all_queues_except_current(blocked_proc);
        put_pcb_into_correct_queue(blocked_proc);
        log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
                          blocked_proc->cmd_str);
        i--;
      }
    }

    curr_priority_queue_num = generate_next_priority();

    current_running_pcb = get_next_pcb(curr_priority_queue_num);
    if (current_running_pcb == NULL) {
      sigsuspend(&suspend_set);  // idle until signal received
      continue;
    }

    log_scheduling_event(current_running_pcb->pid, curr_priority_queue_num,
                         current_running_pcb->cmd_str);

    if (spthread_continue(current_running_pcb->thread_handle) != 0 &&
        errno != EINTR) {
      perror("spthread_continue failed in scheduler");
    }
    sigsuspend(&suspend_set);
    if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
        errno != EINTR) {
      perror("spthread_suspend failed in scheduler");
    }
    put_pcb_into_correct_queue(current_running_pcb);
  }
}
```

4.15.2 Variable Documentation

4.15.2.1 curr_priority_arr_index

```
int curr_priority_arr_index = 0
```

Definition at line 40 of file scheduler.c.

4.15.2.2 current_pcbs

```
Vec current_pcbs
```

Definition at line 30 of file scheduler.c.

4.15.2.3 current_running_pcb

```
pcb_t* current_running_pcb
```

Definition at line 38 of file scheduler.c.

4.15.2.4 det_priorities_arr

```
int det_priorities_arr[19]
```

Initial value:

```
= {0, 1, 2, 0, 0, 1, 0, 1, 2, 0,
                              0, 1, 2, 0, 1, 0, 0, 1, 2}
```

Definition at line 41 of file scheduler.c.

4.15.2.5 log_fd

```
int log_fd
```

Definition at line 36 of file scheduler.c.

4.15.2.6 one_priority_queue

```
Vec one_priority_queue
```

Definition at line 25 of file scheduler.c.

4.15.2.7 sleep_blocked_queue

```
Vec sleep_blocked_queue
```

Definition at line 28 of file scheduler.c.

4.15.2.8 tick_counter

```
int tick_counter = 0
```

Definition at line 35 of file scheduler.c.

4.15.2.9 two_priority_queue

```
Vec two_priority_queue
```

Definition at line 26 of file scheduler.c.

4.15.2.10 zero_priority_queue

```
Vec zero_priority_queue
```

Definition at line 24 of file scheduler.c.

4.15.2.11 zombie_queue

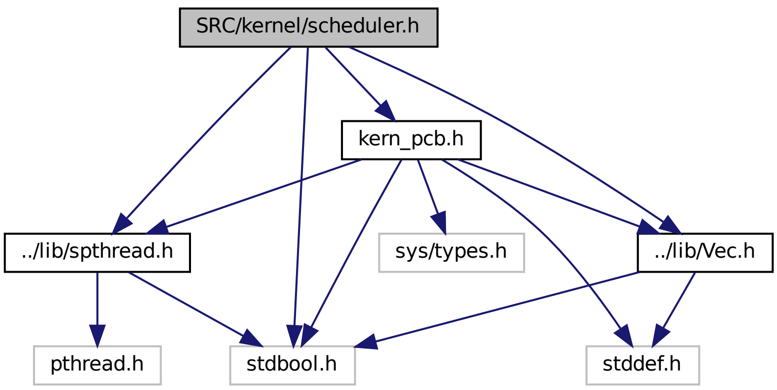
```
Vec zombie_queue
```

Definition at line 27 of file scheduler.c.

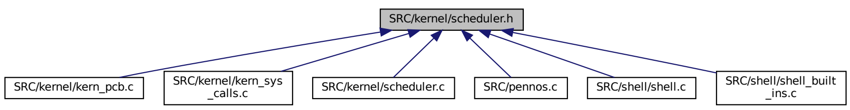
4.16 SRC/kernel/scheduler.h File Reference

```
#include <stdbool.h>
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include "kern_pcb.h"
```

Include dependency graph for scheduler.h:



This graph shows which files directly or indirectly include this file:



Functions

- void initialize_scheduler_queues ()

  Initializes the scheduler queues. This function should be called before any other scheduler functions are called.
- void free_scheduler_queues ()

  Frees the scheduler queues. This function should be called when the scheduler is no longer needed.
- int generate_next_priority ()

  Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.
- pcb_t ∗ get_next_pcb (int priority)

  Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.
- void put_pcb_into_correct_queue (pcb_t ∗pcb)

  Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).
- void delete_process_from_particular_queue (pcb_t ∗pcb, Vec ∗queue)

  Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.
- void delete_process_from_all_queues_except_current (pcb_t ∗pcb)

  Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.
- void delete_process_from_all_queues (pcb_t ∗pcb)

  Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.
- pcb_t ∗ get_pcb_in_queue (Vec ∗queue, pid_t pid)

  Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t∗ associated with that pid.
- bool child_in_zombie_queue (pcb_t ∗parent)

  Checks if a child of the given parent process is in the zombie queue.
- bool child_with_changed_process_status (pcb_t ∗parent)

  Checks if a child of the given parent process has a changed process status.
- void alarm_handler (int signum)

  Handles the alarm signal.
- void handle_signal (pcb_t ∗pcb, int signal)

  Handles a signal for a given process.
- void scheduler ()

  The main scheduler function for PennOS.
- void s_shutdown_pennos ()

  Shuts down the PennOS scheduler.

4.16.1 Function Documentation

4.16.1.1 alarm_handler()

```
void alarm_handler (
            int signum )
```

Handles the alarm signal.

This function is triggered when the alarm signal is received. It increments the global tick counter, which is used for scheduling and timing purposes.

Parameters

signum	The signal number (unused in this implementation).

Handles the alarm signal.

Definition at line 225 of file scheduler.c.

```
                                    {
  tick_counter++;
}
```

4.16.1.2 child_in_zombie_queue()

```
bool child_in_zombie_queue (
            pcb_t * parent )
```

Checks if a child of the given parent process is in the zombie queue.

This function iterates through the zombie queue to determine if any process in the queue has the given parent process as its parent.

Parameters

parent	A pointer to the parent PCB.

Returns

true if a child of the parent is in the zombie queue, false otherwise.

Checks if a child of the given parent process is in the zombie queue.

Definition at line 199 of file scheduler.c.

```
                                                 {
  for (int i = 0; i < vec_len(&zombie_queue); i++) {
    pcb_t* child = vec_get(&zombie_queue, i);
    if (child->par_pid == parent->pid) {
      return true;
    }
  }
  return false;
}
```

4.16.1.3 child_with_changed_process_status()

```
bool child_with_changed_process_status (
            pcb_t * parent )
```

Checks if a child of the given parent process has a changed process status.

This function iterates through the current PCBs to determine if any child of the given parent process has a non-zero process status, indicating a change.

Parameters

parent	A pointer to the parent PCB.

Returns

true if a child of the parent has a changed process status, false otherwise.

Checks if a child of the given parent process has a changed process status.

Definition at line 212 of file scheduler.c.

```
212                                                         {
213   for (int i = 0; i < vec_len(&current_pcbs); i++) {
214     pcb_t* child = vec_get(&current_pcbs, i);
215     if (child->par_pid == parent->pid && child->process_status != 0) {
216       return true;
217     }
218   }
219   return false;
220 }
```

4.16.1.4 delete_process_from_all_queues()

```
void delete_process_from_all_queues (
            pcb_t * pcb )
```

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Parameters

pcb	a pointer to the pcb with the pid to delete

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 177 of file scheduler.c.

```
177                                                   {
178   delete_process_from_all_queues_except_current(pcb);
179   delete_process_from_particular_queue(pcb, &current_pcbs);
180 }
```

4.16.1.5 delete_process_from_all_queues_except_current()

```
void delete_process_from_all_queues_except_current (
            pcb_t * pcb )
```

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_↩ erase. If a particular queue does not contain the pcb, nothing occurs.

Parameters

pcb	a pointer to the pcb with the pid to delete

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 166 of file scheduler.c.

```
166                                                      {
167   delete_process_from_particular_queue(pcb, &zero_priority_queue);
168   delete_process_from_particular_queue(pcb, &one_priority_queue);
169   delete_process_from_particular_queue(pcb, &two_priority_queue);
170   delete_process_from_particular_queue(pcb, &zombie_queue);
171   delete_process_from_particular_queue(pcb, &sleep_blocked_queue);
172 }
```

4.16.1.6 delete_process_from_particular_queue()

```
void delete_process_from_particular_queue (
            pcb_t * pcb,
            Vec * queue )
```

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

Definition at line 153 of file scheduler.c.

```
153                                                                  {
154   for (int i = 0; i < vec_len(queue); i++) {
155     pcb_t* curr_pcb = vec_get(queue, i);
156     if (curr_pcb->pid == pcb->pid) {
157       vec_erase_no_deletor(queue, i);
158       return;
159     }
160   }
161 }
```

4.16.1.7 free_scheduler_queues()

```
void free_scheduler_queues ( )
```

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

Definition at line 66 of file scheduler.c.

```
66                              {
67   vec_destroy(&zero_priority_queue);
68   vec_destroy(&one_priority_queue);
69   vec_destroy(&two_priority_queue);
70   vec_destroy(&zombie_queue);
71   vec_destroy(&sleep_blocked_queue);
72   vec_destroy(&current_pcbs);
73 }
```

4.16.1.8 generate_next_priority()

```
int generate_next_priority ( )
```

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

Precondition

assumes that at least one of the scheduler queues in non-empty

Returns

int 0, 1, or 2 for priority or -1 to signify that all queues are empty

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

Definition at line 82 of file scheduler.c.

```
82                               {
83   // check if all queues are empty
84   if (vec_is_empty(&zero_priority_queue) && vec_is_empty(&one_priority_queue) &&
85       vec_is_empty(&two_priority_queue)) {
86     return -1;
87   }
88
89   int priorities_attempted = 0;
90   while (priorities_attempted < 19) {
91     int curr_pri = det_priorities_arr[curr_priority_arr_index];
92     curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
93     if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
94       priorities_attempted++;
95       return 0;
96     } else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
97       priorities_attempted++;
98       return 1;
99     } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
100       priorities_attempted++;
101       return 2;
102     }
103   }
104
105   return -1;  // should never reach
106 }
```

4.16.1.9 get_next_pcb()

```
pcb_t* get_next_pcb (
              int priority )
```

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

Parameters

priority	queue priority to get next PCB from, or -1 if none

Returns

a ptr to the next pcb struct in queue or NULL if the queue is empty

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

Definition at line 111 of file scheduler.c.

```
111                                  {
112   if (priority == -1) {  // all queues empty
113     return NULL;
114   }
115
116   pcb_t* next_pcb = NULL;
117   if (priority == 0) {
118     next_pcb = vec_get(&zero_priority_queue, 0);
119     vec_erase_no_deletor(&zero_priority_queue, 0);
120   } else if (priority == 1) {
121     next_pcb = vec_get(&one_priority_queue, 0);
122     vec_erase_no_deletor(&one_priority_queue, 0);
123   } else if (priority == 2) {
124     next_pcb = vec_get(&two_priority_queue, 0);
125     vec_erase_no_deletor(&two_priority_queue, 0);
126   }
127
128   return next_pcb;
129 }
```

4.16.1.10 get_pcb_in_queue()

```
pcb_t* get_pcb_in_queue (
            Vec * queue,
            pid_t pid )
```

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

Parameters

queue	the queue of pcb_t* ptrs to search
pid	the pid to search for

Returns

a ptr to the pcb w/ the desired pid if found, NULL otherwise

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

Definition at line 185 of file scheduler.c.

```
185                                                 {
186   for (int i = 0; i < vec_len(queue); i++) {
187     pcb_t* curr_pcb = vec_get(queue, i);
188     if (curr_pcb->pid == pid) {
189       return curr_pcb;
190     }
191   }
192
193   return NULL;
194 }
```

4.16.1.11 handle_signal()

```
void handle_signal (
            pcb_t * pcb,
            int signal )
```

Handles a signal for a given process.

This function processes a signal sent to a process and updates its state accordingly. Supported signals include:

- P_SIGSTOP: Stops the process.
- P_SIGCONT: Continues a stopped process.
- P_SIGTERM: Terminates the process.

Parameters

pcb	A pointer to the PCB of the process receiving the signal.
signal	The signal to handle (0 for P_SIGSTOP, 1 for P_SIGCONT, 2 for P_SIGTERM).

Handles a signal for a given process.

Definition at line 232 of file scheduler.c.

```
232                                               {
233   switch (signal) {
234     case 0:  // P_SIGSTOP
235       if (pcb->process_state == 'R' || pcb->process_state == 'B') {
236         pcb->process_state = 'S';
237         log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
238         delete_process_from_all_queues_except_current(pcb);
239         pcb->process_status = 21;  // STOPPED_BY_SIG
240       }
241       pcb->signals[0] = false;
242       break;
243     case 1:                            // P_SIGCONT
244       if (pcb->process_state == 'S') {  // Only continue if stopped
245         pcb->process_state = 'R';
246         log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
247         delete_process_from_all_queues_except_current(pcb);
248         put_pcb_into_correct_queue(pcb);
249         pcb->process_status = 23;  // Reset status
250       }
251       pcb->signals[1] = false;
252       break;
253     case 2:                            // P_SIGTERM
254       if (pcb->process_state != 'Z') {  // Don't terminate if already zombie
255         pcb->process_state = 'Z';
256         pcb->process_status = 22;  // TERM_BY_SIG
257         log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
258         delete_process_from_all_queues_except_current(pcb);
259         put_pcb_into_correct_queue(pcb);
260         pcb->process_status = 22;  // TERM_BY_SIG
261       }
262       pcb->signals[2] = false;
263       break;
264   }
265 }
```

4.16.1.12 initialize_scheduler_queues()

```
void initialize_scheduler_queues ( )
```

Initializes the scheduler queues. This function should be called before any other scheduler functions are called.

Initializes the scheduler queues. This function should be called before any other scheduler functions are called.

Note

The deconstructors for the queues are set to NULL to prevent double freeing when exiting PennOS.

Definition at line 54 of file scheduler.c.

```
54                                    {
55   zero_priority_queue = vec_new(0, NULL);
56   one_priority_queue = vec_new(0, NULL);
57   two_priority_queue = vec_new(0, NULL);
58   zombie_queue = vec_new(0, NULL);
59   sleep_blocked_queue = vec_new(0, NULL);
60   current_pcbs = vec_new(0, free_pcb);
61 }
```

4.16.1.13 put_pcb_into_correct_queue()

```
void put_pcb_into_correct_queue (
            pcb_t * pcb )
```

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

Definition at line 134 of file scheduler.c.

```
134                                           {
135   if (pcb->process_state == 'R') {
136     if (pcb->priority == 0) {
137       vec_push_back(&zero_priority_queue, pcb);
138     } else if (pcb->priority == 1) {
139       vec_push_back(&one_priority_queue, pcb);
140     } else if (pcb->priority == 2) {
141       vec_push_back(&two_priority_queue, pcb);
142     }
143   } else if (pcb->process_state == 'Z') {
144     vec_push_back(&zombie_queue, pcb);
145   } else if (pcb->process_state == 'B' || pcb->is_sleeping) {
146     vec_push_back(&sleep_blocked_queue, pcb);
147   }
148 }
```

4.16.1.14 s_shutdown_pennos()

```
void s_shutdown_pennos (
            void )
```

Shuts down the PennOS scheduler.

This function sets the scheduling_done flag to true, signaling the scheduler to terminate its loop and shut down.

Shuts down the PennOS scheduler.

Definition at line 270 of file scheduler.c.

```
270                              {
271   scheduling_done = true;
272 }
```

4.16.1.15 scheduler()

```
void scheduler ( )
```

The main scheduler function for PennOS.

This function manages process scheduling, signal handling, and timer-based preemption. It ensures that processes are executed based on their priority and handles signals for both the currently running process and other processes.

Definition at line 277 of file scheduler.c.

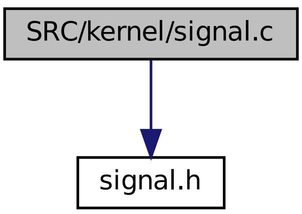
```
277                {
278 int curr_priority_queue_num;
279 
280 // mask for while scheduler is waiting for alarm
281 sigset_t suspend_set;
282 sigfillset(&suspend_set);
283 sigdelset(&suspend_set, SIGALRM);
284 
285 // ensure sigarlm doesn't terminate the process
286 struct sigaction act = (struct sigaction){
287     .sa_handler = alarm_handler,
288     .sa_mask = suspend_set,
289     .sa_flags = SA_RESTART,
290 };
291 sigaction(SIGALRM, &act, NULL);
292 
293 // make sure SIGALRM is unblocked
294 sigset_t alarm_set;
295 sigemptyset(&alarm_set);
296 sigaddset(&alarm_set, SIGALRM);
297 pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
298 
299 struct itimerval it;
300 it.it_interval = (struct timeval){.tv_usec = hundred_millisec};
301 it.it_value = it.it_interval;
302 setitimer(ITIMER_REAL, &it, NULL);
303 
304 while (!scheduling_done) {
305   // handle signals for the currently running process
306   if (current_running_pcb != NULL) {
307     for (int i = 0; i < 3; i++) {
308       if (current_running_pcb->signals[i]) {
309         handle_signal(current_running_pcb, i);
310         // If process was terminated, don't continue scheduling it
311         if (current_running_pcb->process_state != 'R') {
312           current_running_pcb = NULL;
313           break;
314         }
315       }
316     }
317   }
318 
319   // handle signals for all other processes (currently running or not)
320   for (int i = 0; i < vec_len(&current_pcbs); i++) {
321     pcb_t* curr_pcb = vec_get(&current_pcbs, i);
322     for (int j = 0; j < 3; j++) {
323       if (curr_pcb->signals[j]) {
324         handle_signal(curr_pcb, j);
325       }
326     }
327   }
328 
329   // Check sleep/blocked queue to move processes back to scheduable queues
330   for (int i = 0; i < vec_len(&sleep_blocked_queue); i++) {
331     pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
332     bool make_runnable = false;
333     if (blocked_proc->is_sleeping &&
334         blocked_proc->time_to_wake == tick_counter) {
335       blocked_proc->is_sleeping = false;
336       blocked_proc->time_to_wake = -1;
337       blocked_proc->signals[2] = false;  // Unlikely, but reset signal
338       make_runnable = true;
339     } else if (blocked_proc->is_sleeping &&
340              blocked_proc->signals[2]) {  // P_SIGTERM received
341       blocked_proc->is_sleeping = false;
342       blocked_proc->process_state = 'Z';
343       blocked_proc->process_status = 22;  // TERM_BY_SIG
344       blocked_proc->signals[2] = false;
345       delete_process_from_all_queues_except_current(blocked_proc);
346       put_pcb_into_correct_queue(blocked_proc);
347       log_generic_event('Z', blocked_proc->pid, blocked_proc->priority,
348                         blocked_proc->cmd_str);
```

```
        i--;
      } else if (child_in_zombie_queue(blocked_proc)) {
        make_runnable = true;
      } else if (child_with_changed_process_status(blocked_proc)) {
        make_runnable = true;
      }

      if (make_runnable) {
        blocked_proc->process_state = 'R';
        vec_erase_no_deletor(&sleep_blocked_queue, i);
        delete_process_from_all_queues_except_current(blocked_proc);
        put_pcb_into_correct_queue(blocked_proc);
        log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
                          blocked_proc->cmd_str);
        i--;
      }
    }

    curr_priority_queue_num = generate_next_priority();

    current_running_pcb = get_next_pcb(curr_priority_queue_num);
    if (current_running_pcb == NULL) {
      sigsuspend(&suspend_set);  // idle until signal received
      continue;
    }

    log_scheduling_event(current_running_pcb->pid, curr_priority_queue_num,
                         current_running_pcb->cmd_str);

    if (spthread_continue(current_running_pcb->thread_handle) != 0 &&
        errno != EINTR) {
      perror("spthread_continue failed in scheduler");
    }
    sigsuspend(&suspend_set);
    if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
        errno != EINTR) {
      perror("spthread_suspend failed in scheduler");
    }
    put_pcb_into_correct_queue(current_running_pcb);
  }
}
```

4.17 SRC/kernel/signal.c File Reference

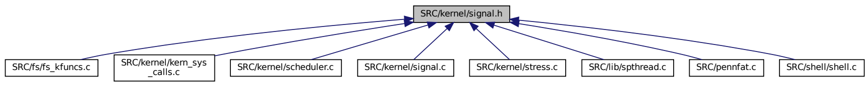
`#include <signal.h>`

Include dependency graph for signal.c:



4.18 SRC/kernel/signal.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define P_SIGSTOP 0

    Signals for PennOS.
- #define P_SIGCONT 1
- #define P_SIGTERM 2
- #define EXITED_NORMALLY 20

    Status definitions.
- #define STOPPED_BY_SIG 21
- #define TERM_BY_SIG 22
- #define CONT_BY_SIG 23
- #define P_WIFEXITED(status) ((status) == EXITED_NORMALLY)

    User-level macros for waitpid status.
- #define P_WIFSTOPPED(status) ((status) == STOPPED_BY_SIG)
- #define P_WIFSIGNALED(status) ((status) == TERM_BY_SIG)

4.18.1 Macro Definition Documentation

4.18.1.1 CONT_BY_SIG

```
#define CONT_BY_SIG 23
```

Definition at line 17 of file signal.h.

4.18.1.2 EXITED_NORMALLY

```
#define EXITED_NORMALLY 20
```

Status definitions.

Definition at line 14 of file signal.h.

4.18.1.3 P_SIGCONT

```
#define P_SIGCONT 1
```

Definition at line 8 of file signal.h.

4.18.1.4 P_SIGSTOP

```
#define P_SIGSTOP 0
```

Signals for PennOS.

Definition at line 7 of file signal.h.

4.18.1.5 P_SIGTERM

```
#define P_SIGTERM 2
```

Definition at line 9 of file signal.h.

4.18.1.6 P_WIFEXITED

```
#define P_WIFEXITED(
              status ) ((status) == EXITED_NORMALLY)
```

User-level macros for waitpid status.

Definition at line 22 of file signal.h.

4.18.1.7 P_WIFSIGNALED

```
#define P_WIFSIGNALED(
              status ) ((status) == TERM_BY_SIG)
```

Definition at line 24 of file signal.h.

4.18.1.8 P_WIFSTOPPED

```
#define P_WIFSTOPPED(
              status ) ((status) == STOPPED_BY_SIG)
```

Definition at line 23 of file signal.h.

4.18.1.9 STOPPED_BY_SIG

```
#define STOPPED_BY_SIG 21
```

Definition at line 15 of file signal.h.

4.18.1.10 TERM_BY_SIG

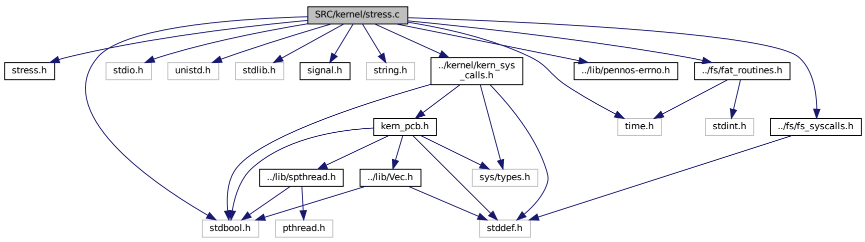
```
#define TERM_BY_SIG 22
```

Definition at line 16 of file signal.h.

4.19 SRC/kernel/stress.c File Reference

```
#include "stress.h"
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include "../kernel/kern_sys_calls.h"
#include "../lib/pennos-errno.h"
#include "../fs/fs_syscalls.h"
#include "../fs/fat_routines.h"
```

Include dependency graph for stress.c:



Functions

- void * hang (void *arg)
- void * nohang (void *arg)
- void * recur (void *arg)
- void * crash (void *arg)

4.19.1 Function Documentation

4.19.1.1 crash()

```
void* crash (
        void * arg )
```

Definition at line 233 of file stress.c.

```
233                      {
234     // This one only works on a file system big enough to hold 5480 bytes
235     crash_main();
236     s_exit();
237     return NULL;
238 }
```

4.19.1.2 hang()

```
void* hang (
        void * arg )
```

Definition at line 215 of file stress.c.

```
215                     {
216     spawn(false);
217     s_exit();
218     return NULL;
219 }
```

4.19.1.3 nohang()

```
void* nohang (
        void * arg )
```

Definition at line 221 of file stress.c.

```
221                       {
222      spawn(true);
223      s_exit();
224      return NULL;
225 }
```

4.19.1.4 recur()

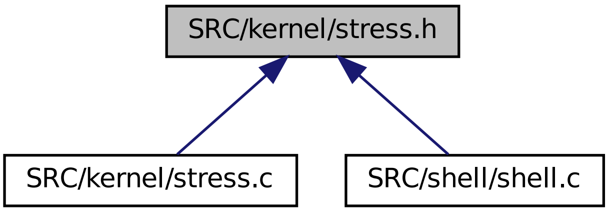
```
void* recur (
            void * arg )
```

Definition at line 227 of file stress.c.

```
227                        {
228     spawn_r(NULL);
229     s_exit();
230     return NULL;
231  }
```

4.20 SRC/kernel/stress.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void ∗ hang (void ∗)
- void ∗ nohang (void ∗)
- void ∗ recur (void ∗)
- void ∗ crash (void ∗)

4.20.1 Function Documentation

4.20.1.1 crash()

```
void* crash (
            void * arg )
```

Definition at line 233 of file stress.c.

```
233                        {
234    // This one only works on a file system big enough to hold 5480 bytes
235    crash_main();
236    s_exit();
237    return NULL;
238  }
```

4.20.1.2 hang()

```
void* hang (
            void * arg )
```

Definition at line 215 of file stress.c.

```
215                        {
216     spawn(false);
217     s_exit();
218     return NULL;
219 }
```

4.20.1.3 nohang()

```
void* nohang (
            void * arg )
```

Definition at line 221 of file stress.c.

```
221                          {
222     spawn(true);
223     s_exit();
224     return NULL;
225 }
```

4.20.1.4 recur()

```
void* recur (
            void * arg )
```

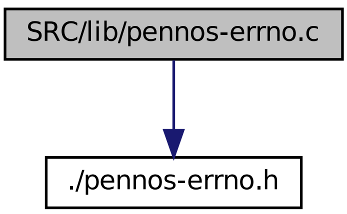
Definition at line 227 of file stress.c.

```
227                         {
228     spawn_r(NULL);
229     s_exit();
230     return NULL;
231 }
```

4.21 SRC/lib/pennos-errno.c File Reference

`#include "./pennos-errno.h"`

Include dependency graph for pennos-errno.c:



Variables

- int P_ERRNO = 0

4.21.1 Variable Documentation

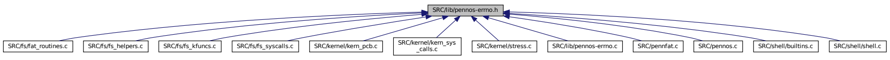
4.21.1.1 P_ERRNO

```
int P_ERRNO = 0
```

Definition at line 8 of file pennos-errno.c.

4.22 SRC/lib/pennos-errno.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define P_ENOENT 1
- #define P_EBADF 2
- #define P_EPERM 3
- #define P_EINVAL 4
- #define P_EEXIST 5
- #define P_EBUSY 6
- #define P_EFULL 7
- #define P_EFS_NOT_MOUNTED 8
- #define P_EINTR 9
- #define P_ENULL 10
- #define P_EREAD 11
- #define P_ELSEEK 12
- #define P_EMAP 13
- #define P_EFUNC 14
- #define P_EOPEN 15
- #define P_EMALLOC 16
- #define P_ESIGNAL 17
- #define P_EWRITE 18
- #define P_ECLOSE 19
- #define P_EPARSE 20
- #define P_ECOMMAND 21
- #define P_NEEDF 22
- #define P_INITFAIL 23
- #define P_EUNKNOWN 99

Variables

- int P_ERRNO

4.22.1 Macro Definition Documentation

4.22.1.1 P_EBADF

```
#define P_EBADF 2
```

Definition at line 8 of file pennos-errno.h.

4.22.1.2 P_EBUSY

```
#define P_EBUSY 6
```

Definition at line 12 of file pennos-errno.h.

4.22.1.3 P_ECLOSE

```
#define P_ECLOSE 19
```

Definition at line 25 of file pennos-errno.h.

4.22.1.4 P_ECOMMAND

```
#define P_ECOMMAND 21
```

Definition at line 27 of file pennos-errno.h.

4.22.1.5 P_EEXIST

```
#define P_EEXIST 5
```

Definition at line 11 of file pennos-errno.h.

4.22.1.6 P_EFS_NOT_MOUNTED

```
#define P_EFS_NOT_MOUNTED 8
```

Definition at line 14 of file pennos-errno.h.

4.22.1.7 P_EFULL

```
#define P_EFULL 7
```

Definition at line 13 of file pennos-errno.h.

4.22.1.8 P_EFUNC

```
#define P_EFUNC 14
```

Definition at line 20 of file pennos-errno.h.

4.22.1.9 P_EINTR

```
#define P_EINTR 9
```

Definition at line 15 of file pennos-errno.h.

4.22.1.10 P_EINVAL

```
#define P_EINVAL 4
```

Definition at line 10 of file pennos-errno.h.

4.22.1.11 P_ELSEEK

```
#define P_ELSEEK 12
```

Definition at line 18 of file pennos-errno.h.

4.22.1.12 P_EMALLOC

```
#define P_EMALLOC 16
```

Definition at line 22 of file pennos-errno.h.

4.22.1.13 P_EMAP

```
#define P_EMAP 13
```

Definition at line 19 of file pennos-errno.h.

4.22.1.14 P_ENOENT

```
#define P_ENOENT 1
```

Definition at line 7 of file pennos-errno.h.

4.22.1.15 P_ENULL

```
#define P_ENULL 10
```

Definition at line 16 of file pennos-errno.h.

4.22.1.16 P_EOPEN

```
#define P_EOPEN 15
```

Definition at line 21 of file pennos-errno.h.

4.22.1.17 P_EPARSE

```
#define P_EPARSE 20
```

Definition at line 26 of file pennos-errno.h.

4.22.1.18 P_EPERM

```
#define P_EPERM 3
```

Definition at line 9 of file pennos-errno.h.

4.22.1.19 P_EREAD

```
#define P_EREAD 11
```

Definition at line 17 of file pennos-errno.h.

4.22.1.20 P_ESIGNAL

```
#define P_ESIGNAL 17
```

Definition at line 23 of file pennos-errno.h.

4.22.1.21 P_EUNKNOWN

```
#define P_EUNKNOWN 99
```

Definition at line 30 of file pennos-errno.h.

4.22.1.22 P_EWRITE

```
#define P_EWRITE 18
```

Definition at line 24 of file pennos-errno.h.

4.22.1.23 P_INITFAIL

```
#define P_INITFAIL 23
```

Definition at line 29 of file pennos-errno.h.

4.22.1.24 P_NEEDF

```
#define P_NEEDF 22
```

Definition at line 28 of file pennos-errno.h.

4.22.2 Variable Documentation

4.22.2.1 P_ERRNO

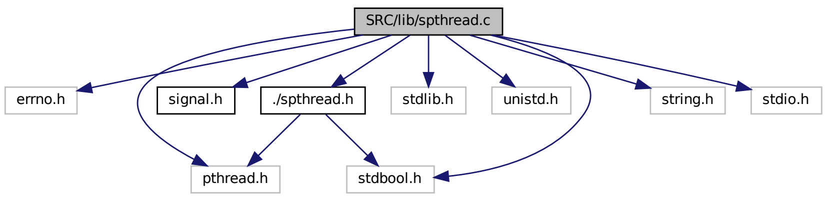
```
int P_ERRNO [extern]
```

Definition at line 8 of file pennos-errno.c.

4.23 SRC/lib/spthread.c File Reference

```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>
#include "./spthread.h"
#include <string.h>
#include <stdio.h>
```

Include dependency graph for spthread.c:



Classes

- struct spthread_fwd_args_st
- struct spthread_signal_args_st
- struct spthread_meta_st

Macros

- #define _GNU_SOURCE
- #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)
- bool spthread_equal (spthread_t first, spthread_t second)
- int spthread_disable_interrupts_self ()
- int spthread_enable_interrupts_self ()

4.23.1 Macro Definition Documentation

4.23.1.1 _GNU_SOURCE

```
#define _GNU_SOURCE
```

Definition at line 1 of file spthread.c.

4.23.1.2 MILISEC_IN_NANO

```
#define MILISEC_IN_NANO 100000
```

Definition at line 12 of file spthread.c.

4.23.1.3 SPTHREAD_RUNNING_STATE

```
#define SPTHREAD_RUNNING_STATE 0
```

Definition at line 76 of file spthread.c.

4.23.1.4 SPTHREAD_SIG_CONTINUE

```
#define SPTHREAD_SIG_CONTINUE -2
```

Definition at line 85 of file spthread.c.

4.23.1.5 SPTHREAD_SIG_SUSPEND

```
#define SPTHREAD_SIG_SUSPEND -1
```

Definition at line 84 of file spthread.c.

4.23.1.6 SPTHREAD_SUSPENDED_STATE

```
#define SPTHREAD_SUSPENDED_STATE 1
```

Definition at line 77 of file spthread.c.

4.23.1.7 SPTHREAD_TERMINATED_STATE

```
#define SPTHREAD_TERMINATED_STATE 2
```

Definition at line 78 of file spthread.c.

4.23.2 Typedef Documentation

4.23.2.1 pthread_fn

```
typedef void*(* pthread_fn) (void *)
```

Definition at line 20 of file spthread.c.

4.23.2.2 spthread_fwd_args

```
typedef struct spthread_fwd_args_st spthread_fwd_args
```

4.23.2.3 spthread_meta_t

```
typedef struct spthread_meta_st spthread_meta_t
```

4.23.2.4 spthread_signal_args

```
typedef struct spthread_signal_args_st spthread_signal_args
```

4.23.3 Function Documentation

4.23.3.1 spthread_cancel()

```
int spthread_cancel (
            spthread_t thread )
```

Definition at line 293 of file spthread.c.

```
293                                        {
294   return pthread_cancel(thread.thread);
295 }
```

4.23.3.2 spthread_continue()

```
int spthread_continue (
            spthread_t thread )
```

Definition at line 241 of file spthread.c.

```
241                                          {
242   pthread_t pself = pthread_self();
243
244   if (pthread_equal(pself, thread.thread) != 0) {
245     // I am already runnning... so just return 0
246     my_meta->state = SPTHREAD_RUNNING_STATE;
247     return 0;
248   }
249
250   spthread_signal_args args = (spthread_signal_args){
251       .signal = SPTHREAD_SIG_CONTINUE,
252       .ack = 0,
253   };
254   pthread_mutex_init(&args.shutup_mutex, NULL);
255
256   int ret = pthread_sigqueue(thread.thread, SIGPTHD,
257                              (union sigval){
258                                  .sival_ptr = &args,
```

```
                            });
  if (ret != 0) {
    pthread_mutex_destroy(&args.shutup_mutex);
    // handles the case where the thread is already dead.
    return ret;
  }

  // wait for our signal to be ack'd

  // setting up args to nanosleep
  const struct timespec t = (struct timespec){
      .tv_nsec = MILISEC_IN_NANO,
  };

  pthread_mutex_lock(&args.shutup_mutex);
  while (args.ack != 1) {
    // wait for a mili second
    pthread_mutex_unlock(&args.shutup_mutex);

    nanosleep(&t, NULL);

    // fprintf(stderr, "susp checking...\n");
    pthread_mutex_lock(&args.shutup_mutex);

    if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
      // child called exit, can break
      break;
    }
  }
  pthread_mutex_unlock(&args.shutup_mutex);
  pthread_mutex_destroy(&args.shutup_mutex);
  return ret;
}
```

4.23.3.3 spthread_create()

```
int spthread_create (
            spthread_t * thread,
            const pthread_attr_t * attr,
            pthread_fn start_routine,
            void * arg )
```

Definition at line 114 of file spthread.c.

```
                             {
  spthread_meta_t* child_meta = malloc(sizeof(spthread_meta_t));
  if (child_meta == NULL) {
    return EAGAIN;
  }

  spthread_fwd_args* fwd_args = malloc(sizeof(spthread_fwd_args));
  if (fwd_args == NULL) {
    free(child_meta);
    return EAGAIN;
  }
  *fwd_args = (spthread_fwd_args){
      .actual_routine = start_routine,
      .actual_arg = arg,
      .setup_done = false,
      .child_meta = child_meta,
  };

  int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
  if (ret != 0) {
    free(child_meta);
    free(fwd_args);
    return EAGAIN;
  }

  ret = pthread_cond_init(&(fwd_args->setup_cond), NULL);
  if (ret != 0) {
    free(child_meta);
    pthread_mutex_destroy(&(fwd_args->setup_mutex));
    free(fwd_args);
    return EAGAIN;
  }

  pthread_t pthread;
```

```
  int result = pthread_create(&pthread, attr, spthread_start, fwd_args);

  pthread_mutex_lock(&(fwd_args->setup_mutex));
  while (fwd_args->setup_done == false) {
    pthread_cond_wait(&(fwd_args->setup_cond), &(fwd_args->setup_mutex));
  }
  pthread_mutex_unlock(&(fwd_args->setup_mutex));

  pthread_cond_destroy(&(fwd_args->setup_cond));
  pthread_mutex_destroy(&(fwd_args->setup_mutex));
  free(fwd_args);

  *thread = (spthread_t){
      .thread = pthread,
      .meta = child_meta,
  };

  return result;
}
```

4.23.3.4 spthread_disable_interrupts_self()

`int spthread_disable_interrupts_self ( )`

Definition at line 326 of file spthread.c.

```
326                                       {
327   sigset_t block_set;
328   int res = sigemptyset(&block_set);
329   if (res != 0) {
330     return res;
331   }
332   res = sigaddset(&block_set, SIGPTHD);
333   if (res != 0) {
334     return res;
335   }
336   res = pthread_sigmask(SIG_BLOCK, &block_set, NULL);
337   if (res != 0) {
338     return res;
339   }
340   return 0;
341 }
```

4.23.3.5 spthread_enable_interrupts_self()

`int spthread_enable_interrupts_self ( )`

Definition at line 345 of file spthread.c.

```
345                                      {
346   sigset_t block_set;
347   int res = sigemptyset(&block_set);
348   if (res != 0) {
349     return res;
350   }
351   res = sigaddset(&block_set, SIGPTHD);
352   if (res != 0) {
353     return res;
354   }
355   res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
356   if (res != 0) {
357     return res;
358   }
359   return 0;
360 }
```

4.23.3.6 spthread_equal()

```
bool spthread_equal (
            spthread_t first,
            spthread_t second )
```

Definition at line 322 of file spthread.c.

```
{
  return pthread_equal(first.thread, second.thread) && (first.meta == second.meta);
}
```

4.23.3.7 spthread_exit()

```
void spthread_exit (
            void * status )
```

Definition at line 315 of file spthread.c.

```
{
  // necessary cleanup is registered
  // in a cleanup routine
  // that is pushed at start of an spthread
  pthread_exit(status);
}
```

4.23.3.8 spthread_join()

```
int spthread_join (
            spthread_t thread,
            void ** retval )
```

Definition at line 308 of file spthread.c.

```
{
  int res = pthread_join(thread.thread, retval);
  pthread_mutex_destroy(&thread.meta->meta_mutex);
  free(thread.meta);
  return res;
}
```

4.23.3.9 spthread_self()

```
bool spthread_self (
            spthread_t * thread )
```

Definition at line 297 of file spthread.c.

```
{
  if (my_meta == NULL) {
    return false;
  }
  *thread = (spthread_t){
      .thread = pthread_self(),
      .meta = my_meta,
  };
  return true;
}
```

4.23.3.10 spthread_suspend()

```
int spthread_suspend (
        spthread_t thread )
```

Definition at line 171 of file spthread.c.

```
                                       {
  pthread_t pself = pthread_self();

  if (pthread_equal(pself, thread.thread) != 0) {
    return spthread_suspend_self();
  }

  spthread_signal_args args = (spthread_signal_args){
      .signal = SPTHREAD_SIG_SUSPEND,
      .ack = 0,
  };
  pthread_mutex_init(&args.shutup_mutex, NULL);

  int ret = pthread_sigqueue(thread.thread, SIGPTHD,
                             (union sigval){
                                 .sival_ptr = &args,
                             });
  if (ret != 0) {
    pthread_mutex_destroy(&args.shutup_mutex);
    // handles the case where the thread is already dead.
    return ret;
  }

  // wait for our signal to be ack'd

  // setting up args to nanosleep
  const struct timespec t = (struct timespec){
      .tv_nsec = MILISEC_IN_NANO,
  };

  nanosleep(&t, NULL);

  pthread_mutex_lock(&args.shutup_mutex);
  while (args.ack != 1) {
    // wait for a mili second
    pthread_mutex_unlock(&args.shutup_mutex);


    nanosleep(&t, NULL);

    // fprintf(stderr, "susp checking...\n");
    pthread_mutex_lock(&args.shutup_mutex);

    if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
      // child called exit, can break
      break;
    }
  }
  pthread_mutex_unlock(&args.shutup_mutex);

  pthread_mutex_destroy(&args.shutup_mutex);
  return ret;
}
```

4.23.3.11 spthread_suspend_self()

```
int spthread_suspend_self ( )
```

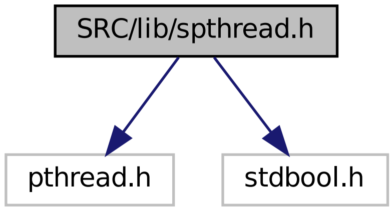
Definition at line 225 of file spthread.c.

```
                             {
  spthread_t self;
  bool am_sp = spthread_self(&self);
  if (!am_sp) {
    return ESRCH;
  }

  my_meta->state = SPTHREAD_SUSPENDED_STATE;

  do {
    sigsuspend(&my_meta->suspend_set);
  } while (my_meta->state == SPTHREAD_SUSPENDED_STATE);

  return 0;
}
```

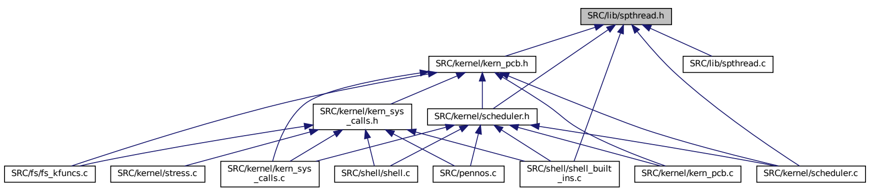
4.24 SRC/lib/spthread.h File Reference

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

Include dependency graph for spthread.h:



This graph shows which files directly or indirectly include this file:



Classes

- 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)
- bool spthread_equal (spthread_t first, spthread_t second)
- int spthread_disable_interrupts_self ()
- int spthread_enable_interrupts_self ()

4.24.1 Macro Definition Documentation

4.24.1.1 SIGPTHD

```
#define SIGPTHD SIGUSR1
```

Definition at line 19 of file spthread.h.

4.24.2 Typedef Documentation

4.24.2.1 spthread_meta_t

```
typedef struct spthread_meta_st spthread_meta_t
```

Definition at line 1 of file spthread.h.

4.24.2.2 spthread_t

```
typedef struct spthread_st spthread_t
```

4.24.3 Function Documentation

4.24.3.1 spthread_cancel()

```
int spthread_cancel (
            spthread_t thread )
```

Definition at line 293 of file spthread.c.

```
293                                           {
294   return pthread_cancel(thread.thread);
295 }
```

4.24.3.2 spthread_continue()

```
int spthread_continue (
            spthread_t thread )
```

Definition at line 241 of file spthread.c.

```
241                                             {
242   pthread_t pself = pthread_self();
243 
244   if (pthread_equal(pself, thread.thread) != 0) {
245     // I am already runnning... so just return 0
246     my_meta->state = SPTHREAD_RUNNING_STATE;
247     return 0;
248   }
249 
250   spthread_signal_args args = (spthread_signal_args){
251       .signal = SPTHREAD_SIG_CONTINUE,
252       .ack = 0,
253   };
254   pthread_mutex_init(&args.shutup_mutex, NULL);
255 
256   int ret = pthread_sigqueue(thread.thread, SIGPTHD,
257                              (union sigval){
258                                  .sival_ptr = &args,
259                              });
260   if (ret != 0) {
261     pthread_mutex_destroy(&args.shutup_mutex);
262     // handles the case where the thread is already dead.
263     return ret;
264   }
265 
266   // wait for our signal to be ack'd
267 
268   // setting up args to nanosleep
269   const struct timespec t = (struct timespec){
270       .tv_nsec = MILISEC_IN_NANO,
271   };
272 
273   pthread_mutex_lock(&args.shutup_mutex);
274   while (args.ack != 1) {
275     // wait for a mili second
276     pthread_mutex_unlock(&args.shutup_mutex);
277 
278     nanosleep(&t, NULL);
279 
280     // fprintf(stderr, "susp checking...\n");
281     pthread_mutex_lock(&args.shutup_mutex);
282 
283     if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
284       // child called exit, can break
285       break;
286     }
287   }
288   pthread_mutex_unlock(&args.shutup_mutex);
289   pthread_mutex_destroy(&args.shutup_mutex);
290   return ret;
291 }
```

4.24.3.3 spthread_create()

```
int spthread_create (
            spthread_t * thread,
            const pthread_attr_t * attr,
            void *(*)(void *) start_routine,
            void * arg )
```

Definition at line 114 of file spthread.c.

```
117                                  {
118   spthread_meta_t* child_meta = malloc(sizeof(spthread_meta_t));
119   if (child_meta == NULL) {
120     return EAGAIN;
121   }
122 
123   spthread_fwd_args* fwd_args = malloc(sizeof(spthread_fwd_args));
124   if (fwd_args == NULL) {
125     free(child_meta);
126     return EAGAIN;
127   }
128   *fwd_args = (spthread_fwd_args){
129       .actual_routine = start_routine,
130       .actual_arg = arg,
131       .setup_done = false,
132       .child_meta = child_meta,
133   };
134 
135   int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
136   if (ret != 0) {
137     free(child_meta);
138     free(fwd_args);
139     return EAGAIN;
140   }
141 
142   ret = pthread_cond_init(&(fwd_args->setup_cond), NULL);
143   if (ret != 0) {
144     free(child_meta);
145     pthread_mutex_destroy(&(fwd_args->setup_mutex));
146     free(fwd_args);
147     return EAGAIN;
148   }
149 
150   pthread_t pthread;
151   int result = pthread_create(&pthread, attr, spthread_start, fwd_args);
152 
153   pthread_mutex_lock(&(fwd_args->setup_mutex));
154   while (fwd_args->setup_done == false) {
155     pthread_cond_wait(&(fwd_args->setup_cond), &(fwd_args->setup_mutex));
156   }
157   pthread_mutex_unlock(&(fwd_args->setup_mutex));
158 
159   pthread_cond_destroy(&(fwd_args->setup_cond));
160   pthread_mutex_destroy(&(fwd_args->setup_mutex));
161   free(fwd_args);
162 
163   *thread = (spthread_t){
164       .thread = pthread,
165       .meta = child_meta,
166   };
167 
168   return result;
169 }
```

4.24.3.4 spthread_disable_interrupts_self()

```
int spthread_disable_interrupts_self ( )
```

Definition at line 326 of file spthread.c.

```
326                                           {
327   sigset_t block_set;
328   int res = sigemptyset(&block_set);
329   if (res != 0) {
330     return res;
331   }
```

```
332   res = sigaddset(&block_set, SIGPTHD);
333   if (res != 0) {
334     return res;
335   }
336   res = pthread_sigmask(SIG_BLOCK, &block_set, NULL);
337   if (res != 0) {
338     return res;
339   }
340   return 0;
341 }
```

4.24.3.5 spthread_enable_interrupts_self()

```
int spthread_enable_interrupts_self ( )
```

Definition at line 345 of file spthread.c.

```
345                                         {
346   sigset_t block_set;
347   int res = sigemptyset(&block_set);
348   if (res != 0) {
349     return res;
350   }
351   res = sigaddset(&block_set, SIGPTHD);
352   if (res != 0) {
353     return res;
354   }
355   res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
356   if (res != 0) {
357     return res;
358   }
359   return 0;
360 }
```

4.24.3.6 spthread_equal()

```
bool spthread_equal (
              spthread_t first,
              spthread_t second )
```

Definition at line 322 of file spthread.c.

```
322                                                           {
323   return pthread_equal(first.thread, second.thread) && (first.meta == second.meta);
324 }
```

4.24.3.7 spthread_exit()

```
void spthread_exit (
              void * status )
```

Definition at line 315 of file spthread.c.

```
315                                    {
316   // necessary cleanup is registered
317   // in a cleanup routine
318   // that is pushed at start of an spthread
319   pthread_exit(status);
320 }
```

4.24.3.8 spthread_join()

```
int spthread_join (
            spthread_t thread,
            void ** retval )
```

Definition at line 308 of file spthread.c.

```
308                                                     {
309   int res = pthread_join(thread.thread, retval);
310   pthread_mutex_destroy(&thread.meta->meta_mutex);
311   free(thread.meta);
312   return res;
313 }
```

4.24.3.9 spthread_self()

```
bool spthread_self (
            spthread_t * thread )
```

Definition at line 297 of file spthread.c.

```
297                                       {
298   if (my_meta == NULL) {
299     return false;
300   }
301   *thread = (spthread_t){
302       .thread = pthread_self(),
303       .meta = my_meta,
304   };
305   return true;
306 }
```

4.24.3.10 spthread_suspend()

```
int spthread_suspend (
            spthread_t thread )
```

Definition at line 171 of file spthread.c.

```
171                                        {
172   pthread_t pself = pthread_self();
173
174   if (pthread_equal(pself, thread.thread) != 0) {
175     return spthread_suspend_self();
176   }
177
178   spthread_signal_args args = (spthread_signal_args){
179       .signal = SPTHREAD_SIG_SUSPEND,
180       .ack = 0,
181   };
182   pthread_mutex_init(&args.shutup_mutex, NULL);
183
184   int ret = pthread_sigqueue(thread.thread, SIGPTHD,
185                              (union sigval){
186                                  .sival_ptr = &args,
187                              });
188   if (ret != 0) {
189     pthread_mutex_destroy(&args.shutup_mutex);
190     // handles the case where the thread is already dead.
191     return ret;
192   }
193
194   // wait for our signal to be ack'd
195
196   // setting up args to nanosleep
197   const struct timespec t = (struct timespec){
198       .tv_nsec = MILISEC_IN_NANO,
```

```
  };

  nanosleep(&t, NULL);

  pthread_mutex_lock(&args.shutup_mutex);
  while (args.ack != 1) {
    // wait for a mili second
    pthread_mutex_unlock(&args.shutup_mutex);


    nanosleep(&t, NULL);

    // fprintf(stderr, "susp checking...\n");
    pthread_mutex_lock(&args.shutup_mutex);

    if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
      // child called exit, can break
      break;
    }
  }
  pthread_mutex_unlock(&args.shutup_mutex);

  pthread_mutex_destroy(&args.shutup_mutex);
  return ret;
}
```

4.24.3.11 spthread_suspend_self()

```
int spthread_suspend_self ( )
```

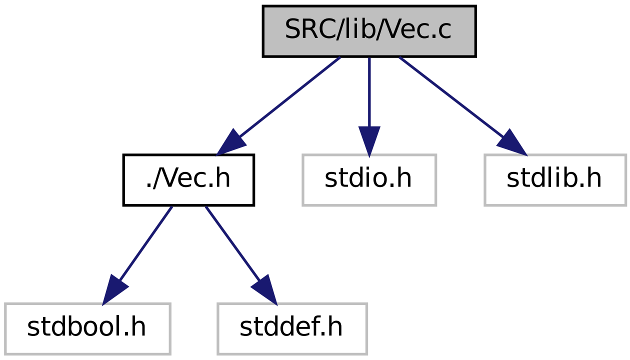
Definition at line 225 of file spthread.c.

```
                          {
  spthread_t self;
  bool am_sp = spthread_self(&self);
  if (!am_sp) {
    return ESRCH;
  }

  my_meta->state = SPTHREAD_SUSPENDED_STATE;

  do {
    sigsuspend(&my_meta->suspend_set);
  } while (my_meta->state == SPTHREAD_SUSPENDED_STATE);

  return 0;
}
```

4.25 SRC/lib/Vec.c File Reference

```
#include "./Vec.h"
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for Vec.c:



Functions

- Vec vec_new (size_t initial_capacity, ptr_dtor_fn ele_dtor_fn)
- void vec_destroy (Vec ∗self)
- void vec_clear (Vec ∗self)
- void vec_resize (Vec ∗self, size_t new_capacity)
- void vec_erase (Vec ∗self, size_t index)
- void vec_erase_no_deletor (Vec ∗self, size_t index)
- void vec_insert (Vec ∗self, size_t index, ptr_t new_ele)
- bool vec_pop_back (Vec ∗self)
- void vec_push_back (Vec ∗self, ptr_t new_ele)
- void vec_set (Vec ∗self, size_t index, ptr_t new_ele)
- ptr_t vec_get (Vec ∗self, size_t index)

4.25.1 Function Documentation

4.25.1.1 vec_clear()

```
void vec_clear (
            Vec * self )
```

Erases all elements from the container. After this, the length of the vector is zero. Capacity of the vector is unchanged.

Parameters

self	a pointer to the vector we want to clear.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up).

Definition at line 34 of file Vec.c.

```
34                              {
35   if (self->ele_dtor_fn) {
36     for (int i = 0; i < self->length; i++) {
37       self->ele_dtor_fn(self->data[i]);
38     }
39   }
40 
41   self->length = 0;
42 }
```

4.25.1.2 vec_destroy()

```
void vec_destroy (
             Vec * self )
```

Destruct the vector. All elements are destructed and storage is deallocated. Must set capacity and length to zero. Data is set to NULL.

Parameters

self	a pointer to the vector we want to destruct.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up) and data storage deallocated.

Definition at line 17 of file Vec.c.

```
17                                {
18   if (self->ele_dtor_fn) {
19     for (int i = 0; i < self->length; i++) {
20       self->ele_dtor_fn(self->data[i]);
21     }
22   }
23   free(self->data);
24 }
```

4.25.1.3 vec_erase()

```
void vec_erase (
             Vec * self,
             size_t index )
```

Erases an element at the specified valid location in the container

Parameters

self	a pointer to the vector we want to erase from.
index	the index of the element we want to erase at. Elements after this index are "shifted" down one position.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 63 of file Vec.c.

```
63                                    {
64   if (index >= self->length) {
65     perror("vec_erase: index >= vec length");
66   }
67
68   if (self->ele_dtor_fn) {
69     self->ele_dtor_fn(self->data[index]);
70   }
71
72   for (unsigned int i = index; i < self->length - 1; i++) {
73     self->data[i] = self->data[i + 1];
74   }
75
76   self->length--;
77 }
```

4.25.1.4 vec_erase_no_deletor()

```
void vec_erase_no_deletor (
            Vec * self,
            size_t index )
```

Erases an element at the specified location in the container, except it does not call the element deletor function on the element.

Parameters

self	a pointer to the vector we want to erase from
index	the index of the element we want to erase at

Definition at line 79 of file Vec.c.

```
79                                               {
80   if (index >= self->length) {
81     perror("vec_erase: index >= vec length");
82   }
83
84   for (unsigned int i = index; i < self->length - 1; i++) {
85     self->data[i] = self->data[i + 1];
86   }
87
88   self->length--;
89 }
```

4.25.1.5 vec_get()

```
ptr_t vec_get (
            Vec * self,
            size_t index )
```

Gets the specified element of the Vec

Parameters

self	a pointer to the vector who's element we want to get.
index	the index of the element to get.

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 152 of file Vec.c.

```
152                                {
153   if (index >= self->length) {
154     perror("vec_get: index greater than length");
155   }
156   return self->data[index];
157 }
```

4.25.1.6 vec_insert()

```
void vec_insert (
            Vec * self,
            size_t index,
            ptr_t new_ele )
```

Inserts an element at the specified location in the container

Parameters

self	a pointer to the vector we want to insert into.
index	the index of the element we want to insert at. Elements at this index and after it are "shifted" up one position. If index is equal to the length, then we insert at the end of the vector.
new_ele	the value we want to insert

Precondition

Assumes self points to a valid vector. If the index is > self->length then this function will call perror

Postcondition

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 91 of file Vec.c.

```c
                                                        {
  if (index > self->length) {
    perror("vec_insert: index > vec length");
  }

  if (index == self->length) {  // Insertion at end = Adding at end
    vec_push_back(self, new_ele);
  } else {  // Inserting not at the end
    // Vector is full
     if (self->length == self->capacity) {
       vec_resize(self, self->capacity * 2);
     }
     // Insertion + Displacement
     for (unsigned int i = self->length; i > index; i--) {
       self->data[i] = self->data[i - 1];
     }
     self->data[index] = new_ele;

     self->length++;
   }
}
```

4.25.1.7 vec_new()

```c
Vec vec_new (
              size_t initial_capacity,
              ptr_dtor_fn ele_dtor_fn )
```

Creates a new empty Vec(tor) with the specified initial_capacity and specified function to clean up elements in the vector.

Parameters

initial_capacity	the initial capacity of the newly created vector, non negative
ele_dtor_fn	a function pointer to a function that takes in a ptr_t (a vector element) and cleans it up. This is commonly just `free` but custom functions can be passed in. NULL can also be passed in to specify that there is no cleanup function that needs to be called on each element.

Returns

a newly created vector with specified capacity, 0 length and the specified element destructor (cleanup) function.

Postcondition

if memory allocation fails, the function will perror

Definition at line 5 of file Vec.c.

```c
                                                             {
  Vec vector;
  vector.capacity = initial_capacity;
  vector.data = malloc(sizeof(void*) * initial_capacity);
  if (vector.data == NULL) {
    perror("vec_new: malloc failed");
  }
  vector.ele_dtor_fn = ele_dtor_fn;
  vector.length = 0;
  return vector;
}
```

4.25.1.8 vec_pop_back()

```
bool vec_pop_back (
            Vec * self )
```

Removes and destroys the last element of the Vec

Parameters

self	a pointer to the vector we are popping.

Returns

true iff an element was removed.

Precondition

Assumes self points to a valid vector.

Postcondition

The capacity of self stays the same. The removed element is destructed (cleaned up) as specified by the dtor_fn provided in vec_new.

Definition at line 113 of file Vec.c.

```
113                         {
114   if (self->length == 0) {
115     return false;
116   }
117   if (self->ele_dtor_fn) {
118     self->ele_dtor_fn(self->data[--self->length]);
119   } else {
120     self->length--;
121   }
122   return true;
123 }
```

4.25.1.9 vec_push_back()

```
void vec_push_back (
            Vec * self,
            ptr_t new_ele )
```

Appends the given element to the end of the Vec

Parameters

self	a pointer to the vector we are pushing onto
new_ele	the value we want to add to the end of the container

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize is needed and it fails, then this function will call perror

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. If initial capacity is zero, it is resized to capacity 1. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 125 of file Vec.c.

```
125                                              {
126   if (self->capacity == self->length) {
127     if (self->capacity == 0) {
128       vec_resize(self, 1);
129     } else {
130       vec_resize(self, self->capacity * 2);
131     }
132   }
133 
134   if (self->capacity == self->length) {
135     perror("vec_push_back: resize failed");
136   }
137 
138   // The array is 0 indexed
139   self->data[self->length++] = new_ele;
140 }
```

4.25.1.10 vec_resize()

```
void vec_resize (
            Vec * self,
            size_t new_capacity )
```

Resizes the container to a new specified capacity. Does nothing if new_capacity <= self->length

Parameters

self	a pointer to the vector we want to resize.
new_capacity	the new capacity of the vector.

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize takes place, then a reallocation takes place and all elements are copied over. Any pointers to elements prior to this reallocation are invalidated.

The removed elements are destructed (cleaned up).

Definition at line 44 of file Vec.c.

```
44                                                     {
45   if (new_capacity * sizeof(void*) < new_capacity) {
46     perror("vec_resize: new capacity too large");
```

```
  }
  if (new_capacity > self->length) {
    self->capacity = new_capacity;
    ptr_t* new_data = malloc(sizeof(void*) * self->capacity);

    // Copy over old elements
    for (int i = 0; i < self->length; i++) {
      new_data[i] = self->data[i];
    }

    free(self->data);

    self->data = new_data;
  }
}
```

4.25.1.11 vec_set()

```
void vec_set (
              Vec * self,
              size_t index,
              ptr_t new_ele )
```

Sets the specified element of the Vec to the specified value

Parameters

self	a pointer to the vector who's element we want to set.
index	the index of the element to set.
new_ele	the value we want to set the element at that index to

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

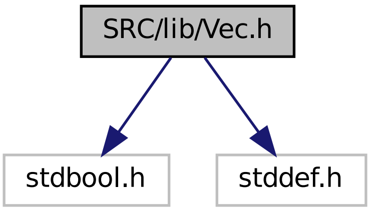
Definition at line 142 of file Vec.c.

```
142                                                {
143   if (index >= self->length) {
144     perror("vec_set: idx >= len");
145   }
146   if (self->ele_dtor_fn) {
147     self->ele_dtor_fn(self->data[index]);
148   }
149   self->data[index] = new_ele;
150 }
```

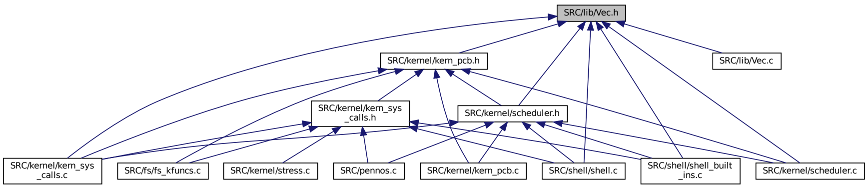
4.26 SRC/lib/Vec.h File Reference

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

Include dependency graph for Vec.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct vec_st

Macros

- #define vec_capacity(vec) ((vec)->capacity)
- #define vec_len(vec) ((vec)->length)
- #define vec_is_empty(vec) ((vec)->length == 0)

Typedefs

- typedef void * ptr_t
- typedef void(* ptr_dtor_fn) (ptr_t)
- typedef struct vec_st Vec

Functions

- Vec vec_new (size_t initial_capacity, ptr_dtor_fn ele_dtor_fn)
- ptr_t vec_get (Vec *self, size_t index)
- void vec_set (Vec *self, size_t index, ptr_t new_ele)
- void vec_push_back (Vec *self, ptr_t new_ele)
- bool vec_pop_back (Vec *self)
- void vec_insert (Vec *self, size_t index, ptr_t new_ele)
- void vec_erase (Vec *self, size_t index)
- void vec_erase_no_deletor (Vec *self, size_t index)
- void vec_resize (Vec *self, size_t new_capacity)
- void vec_clear (Vec *self)
- void vec_destroy (Vec *self)

4.26.1 Macro Definition Documentation

4.26.1.1 vec_capacity

```
#define vec_capacity(
              vec ) ((vec)->capacity)
```

Returns the current capacity of the Vec Written as a function-like macro

Parameters

vec,a	pointer to the vector we want to grab the capacity of.

Definition at line 40 of file Vec.h.

4.26.1.2 vec_is_empty

```
#define vec_is_empty(
              vec ) ((vec)->length == 0)
```

Checks if the Vec is empty written as a function-like macro

Parameters

vec,a	pointer to the vector we want to check emptiness of.

Definition at line 54 of file Vec.h.

4.26.1.3 vec_len

```
#define vec_len(
              vec ) ((vec)->length)
```

Returns the current length of the Vec written as a function-like macro

Parameters

vec,a	pointer to the vector we want to grab the len of.

Definition at line 47 of file Vec.h.

4.26.2 Typedef Documentation

4.26.2.1 ptr_dtor_fn

```
typedef void(* ptr_dtor_fn) (ptr_t)
```

Definition at line 8 of file Vec.h.

4.26.2.2 ptr_t

```
typedef void* ptr_t
```

Definition at line 7 of file Vec.h.

4.26.2.3 Vec

```
typedef struct vec_st Vec
```

4.26.3 Function Documentation

4.26.3.1 vec_clear()

```
void vec_clear (
            Vec * self )
```

Erases all elements from the container. After this, the length of the vector is zero. Capacity of the vector is unchanged.

Parameters

self	a pointer to the vector we want to clear.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up).

Definition at line 34 of file Vec.c.

```
34                            {
35   if (self->ele_dtor_fn) {
36     for (int i = 0; i < self->length; i++) {
37       self->ele_dtor_fn(self->data[i]);
38     }
39   }
40
41   self->length = 0;
42 }
```

4.26.3.2 vec_destroy()

```
void vec_destroy (
              Vec * self )
```

Destruct the vector. All elements are destructed and storage is deallocated. Must set capacity and length to zero. Data is set to NULL.

Parameters

self	a pointer to the vector we want to destruct.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up) and data storage deallocated.

Definition at line 17 of file Vec.c.

```
17                              {
18   if (self->ele_dtor_fn) {
19     for (int i = 0; i < self->length; i++) {
20       self->ele_dtor_fn(self->data[i]);
21     }
22   }
23   free(self->data);
24 }
```

4.26.3.3 vec_erase()

```
void vec_erase (
              Vec * self,
              size_t index )
```

Erases an element at the specified valid location in the container

Parameters

self	a pointer to the vector we want to erase from.
index	the index of the element we want to erase at. Elements after this index are "shifted" down one position.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 63 of file Vec.c.

```
63                                   {
64   if (index >= self->length) {
65     perror("vec_erase: index >= vec length");
66   }
67
68   if (self->ele_dtor_fn) {
69     self->ele_dtor_fn(self->data[index]);
70   }
71
72   for (unsigned int i = index; i < self->length - 1; i++) {
73     self->data[i] = self->data[i + 1];
74   }
75
76   self->length--;
77 }
```

4.26.3.4 vec_erase_no_deletor()

```
void vec_erase_no_deletor (
            Vec * self,
            size_t index )
```

Erases an element at the specified location in the container, except it does not call the element deletor function on the element.

Parameters

self	a pointer to the vector we want to erase from
index	the index of the element we want to erase at

Definition at line 79 of file Vec.c.

```
79                                               {
80   if (index >= self->length) {
81     perror("vec_erase: index >= vec length");
82   }
83
84   for (unsigned int i = index; i < self->length - 1; i++) {
85     self->data[i] = self->data[i + 1];
86   }
87
88   self->length--;
89 }
```

4.26.3.5 vec_get()

```
ptr_t vec_get (
            Vec * self,
            size_t index )
```

Gets the specified element of the Vec

Parameters

self	a pointer to the vector who's element we want to get.
index	the index of the element to get.

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 152 of file Vec.c.

```
152                                  {
153   if (index >= self->length) {
154     perror("vec_get: index greater than length");
155   }
156   return self->data[index];
157 }
```

4.26.3.6 vec_insert()

```
void vec_insert (
            Vec * self,
            size_t index,
            ptr_t new_ele )
```

Inserts an element at the specified location in the container

Parameters

self	a pointer to the vector we want to insert into.
index	the index of the element we want to insert at. Elements at this index and after it are "shifted" up one position. If index is equal to the length, then we insert at the end of the vector.
new_ele	the value we want to insert

Precondition

Assumes self points to a valid vector. If the index is > self->length then this function will call perror

Postcondition

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 91 of file Vec.c.

```
                                                          {
  if (index > self->length) {
    perror("vec_insert: index > vec length");
  }

  if (index == self->length) {  // Insertion at end = Adding at end
    vec_push_back(self, new_ele);
  } else {  // Inserting not at the end
    // Vector is full
    if (self->length == self->capacity) {
      vec_resize(self, self->capacity * 2);
    }
    // Insertion + Displacement
    for (unsigned int i = self->length; i > index; i--) {
      self->data[i] = self->data[i - 1];
    }
    self->data[index] = new_ele;

    self->length++;
  }
}
```

4.26.3.7 vec_new()

```
Vec vec_new (
              size_t initial_capacity,
              ptr_dtor_fn ele_dtor_fn )
```

Creates a new empty Vec(tor) with the specified initial_capacity and specified function to clean up elements in the vector.

Parameters

initial_capacity	the initial capacity of the newly created vector, non negative
ele_dtor_fn	a function pointer to a function that takes in a ptr_t (a vector element) and cleans it up. This is commonly just `free` but custom functions can be passed in. NULL can also be passed in to specify that there is no cleanup function that needs to be called on each element.

Returns

a newly created vector with specified capacity, 0 length and the specified element destructor (cleanup) function.

Postcondition

if memory allocation fails, the function will perror

Definition at line 5 of file Vec.c.

```
                                                                 {
  Vec vector;
  vector.capacity = initial_capacity;
  vector.data = malloc(sizeof(void*) * initial_capacity);
  if (vector.data == NULL) {
    perror("vec_new: malloc failed");
  }
  vector.ele_dtor_fn = ele_dtor_fn;
  vector.length = 0;
  return vector;
}
```

4.26.3.8 vec_pop_back()

```
bool vec_pop_back (
            Vec * self )
```

Removes and destroys the last element of the Vec

Parameters

self	a pointer to the vector we are popping.

Returns

true iff an element was removed.

Precondition

Assumes self points to a valid vector.

Postcondition

The capacity of self stays the same. The removed element is destructed (cleaned up) as specified by the dtor_fn provided in vec_new.

Definition at line 113 of file Vec.c.

```
113                           {
114   if (self->length == 0) {
115     return false;
116   }
117   if (self->ele_dtor_fn) {
118     self->ele_dtor_fn(self->data[--self->length]);
119   } else {
120     self->length--;
121   }
122   return true;
123 }
```

4.26.3.9 vec_push_back()

```
void vec_push_back (
            Vec * self,
            ptr_t new_ele )
```

Appends the given element to the end of the Vec

Parameters

self	a pointer to the vector we are pushing onto
new_ele	the value we want to add to the end of the container

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize is needed and it fails, then this function will call perror

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. If initial capacity is zero, it is resized to capacity 1. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 125 of file Vec.c.

```
125                                            {
126   if (self->capacity == self->length) {
127     if (self->capacity == 0) {
128       vec_resize(self, 1);
129     } else {
130       vec_resize(self, self->capacity * 2);
131     }
132   }
133
134   if (self->capacity == self->length) {
135     perror("vec_push_back: resize failed");
136   }
137
138   // The array is 0 indexed
139   self->data[self->length++] = new_ele;
140 }
```

4.26.3.10 vec_resize()

```
void vec_resize (
            Vec * self,
            size_t new_capacity )
```

Resizes the container to a new specified capacity. Does nothing if new_capacity <= self->length

Parameters

self	a pointer to the vector we want to resize.
new_capacity	the new capacity of the vector.

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize takes place, then a reallocation takes place and all elements are copied over. Any pointers to elements prior to this reallocation are invalidated.

The removed elements are destructed (cleaned up).

Definition at line 44 of file Vec.c.

```
44                                                  {
45   if (new_capacity * sizeof(void*) < new_capacity) {
46     perror("vec_resize: new capacity too large");
```

```
  }
  if (new_capacity > self->length) {
    self->capacity = new_capacity;
    ptr_t* new_data = malloc(sizeof(void*) * self->capacity);

    // Copy over old elements
    for (int i = 0; i < self->length; i++) {
      new_data[i] = self->data[i];
    }

    free(self->data);

    self->data = new_data;
  }
}
```

4.26.3.11 vec_set()

```
void vec_set (
            Vec * self,
            size_t index,
            ptr_t new_ele )
```

Sets the specified element of the Vec to the specified value

Parameters

self	a pointer to the vector who's element we want to set.
index	the index of the element to set.
new_ele	the value we want to set the element at that index to

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 142 of file Vec.c.

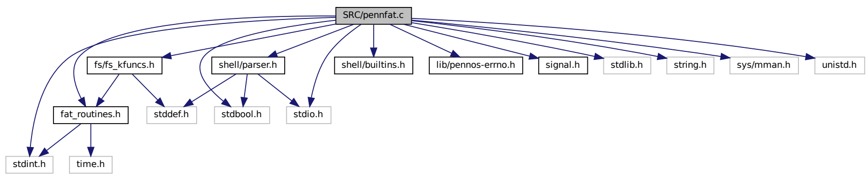
```
142                                                 {
143   if (index >= self->length) {
144     perror("vec_set: idx >= len");
145   }
146   if (self->ele_dtor_fn) {
147     self->ele_dtor_fn(self->data[index]);
148   }
149   self->data[index] = new_ele;
150 }
```

4.27 SRC/pennfat.c File Reference

```
#include "shell/parser.h"
#include "shell/builtins.h"
#include "lib/pennos-errno.h"
#include "fs/fs_kfuncs.h"
```

```
#include "fs/fat_routines.h"
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdbool.h>
```

Include dependency graph for pennfat.c:



Macros

- #define PROMPT "pennfat# "

Functions

- int main (int argc, char ∗argv[ ])

4.27.1 Macro Definition Documentation

4.27.1.1 PROMPT

```
#define PROMPT "pennfat# "
```

Definition at line 16 of file pennfat.c.

4.27.2 Function Documentation

4.27.2.1 main()

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

Definition at line 24 of file pennfat.c.

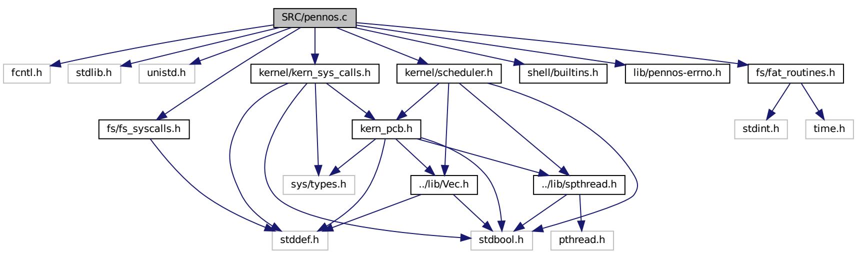
```
                                    {
    // register signal handlers
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    // set up handler for SIGINT (ctrl-c)
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        P_ERRNO = P_ESIGNAL;
        u_perror("Error setting up SIGINT handler");
        return EXIT_FAILURE;
    }

    // set up handler for SIGTSTP (ctrl-z)
    if (sigaction(SIGTSTP, &sa, NULL) == -1) {
        P_ERRNO = P_ESIGNAL;
        u_perror("Error setting up SIGTSTP handler");
        return EXIT_FAILURE;
    }

    char input_buffer[1024];

    while (true) {
        // print prompt
        if (k_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {
            P_ERRNO = P_EWRITE;
            u_perror("prompt write error");
            break;
        }

        // read user input
        int bytes_read = k_read(STDIN_FILENO, input_buffer, sizeof(input_buffer) - 1);

        // check for EOF (ctrl-D)
        if (bytes_read <= 0) {
            k_write(STDOUT_FILENO, "\n", 1);
            break;
        }

        // remove trailing newline if present
        if (bytes_read > 0 && input_buffer[bytes_read - 1] == '\n') {
            input_buffer[bytes_read - 1] = '\0';
        }

        // parse command and check error
        struct parsed_command *parsed_command = NULL;
        int parse_result = parse_command(input_buffer, &parsed_command);
        if (parse_result != 0) {
            if (parse_result == -1) {
                P_ERRNO = P_EINVAL;
                u_perror("Error parsing command");
            } else {
                print_parser_errcode(stderr, parse_result);
            }
            continue;
        }

        // skip empty commands
        if (parsed_command->num_commands == 0) {
            free(parsed_command);
            continue;
        }

        // extract command and arguments
        char **args = parsed_command->commands[0];

        // execute command
        if (strcmp(args[0], "mkfs") == 0) {
            if (args[1] == NULL || args[2] == NULL || args[3] == NULL) {
                P_ERRNO = P_EINVAL;
                u_perror("mkfs");
            } else {
                int blocks_in_fat = atoi(args[2]);
                int block_size = atoi(args[3]);
                if (mkfs(args[1], blocks_in_fat, block_size) != 0) {
                    u_perror("mkfs");
```

```
                }
            }
        } else if (strcmp(args[0], "mount") == 0) {
            if (args[1] == NULL) {
                P_ERRNO = P_EINVAL;
                u_perror("mount");
            } else {
                if (mount(args[1]) != 0) {
                    u_perror("mount");
                }
            }
        } else if (strcmp(args[0], "unmount") == 0) {
            if (unmount() != 0) {
                u_perror("unmount");
            }
        } else if (strcmp(args[0], "ls") == 0) {
            ls(args);
        } else if (strcmp(args[0], "touch") == 0) {
            touch(args);
        } else if (strcmp(args[0], "cat") == 0) {
            cat(args);
        } else if (strcmp(args[0], "chmod") == 0) {
            chmod(args);
        }  else if (strcmp(args[0], "mv") == 0) {
            mv(args);
        } else if (strcmp(args[0], "rm") == 0) {
            rm(args);
        } else if (strcmp(args[0], "cp") == 0) {
            cp(args);
        } else if (strcmp(args[0], "cmpctdir") == 0) { // extra credit
            cmpctdir(args);
        } else {
            P_ERRNO = P_ECOMMAND;
            u_perror("shell");
        }

        free(parsed_command);
    }
    return EXIT_SUCCESS;
}
```

4.28 SRC/pennos.c File Reference

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include "fs/fs_syscalls.h"
#include "kernel/kern_sys_calls.h"
#include "kernel/scheduler.h"
#include "shell/builtins.h"
#include "lib/pennos-errno.h"
#include "fs/fat_routines.h"
```

Include dependency graph for pennos.c:



Functions

- int main (int argc, char ∗argv[ ])

Variables

- int tick_counter
- int log_fd

4.28.1 Function Documentation

4.28.1.1 main()

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

Definition at line 14 of file pennos.c.

```
14                                  {
15   // mount the filesystem
16   if (argc < 2) {
17     P_ERRNO = P_NEEDF;
18     u_perror("need a pennfat file to mount");
19     return -1;
20   } else {
21     if (mount(argv[1]) == -1) {
22       u_perror("mount failed");
23       return -1;
24     }
25   }
26
27   // get the log fd
28   if (argc >= 3) {
29     log_fd = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0644);
30   } else {
31     log_fd = open("log/log", O_RDWR | O_CREAT | O_TRUNC, 0644);
32   }
33
34   // initialize scheduler architecture and init process
35   initialize_scheduler_queues();
36
37   pid_t init_pid = s_spawn_init();
38   if (init_pid == -1) {
39     P_ERRNO = P_INITFAIL;
40     u_perror("init spawn failed");
41     return -1;
42   }
43
44   scheduler();
45
46   // cleanup
47   s_cleanup_init_process();
48   free_scheduler_queues();
49   unmount();
50   close(log_fd);
51 }
```

4.28.2 Variable Documentation

4.28.2.1 log_fd

```
int log_fd [extern]
```

Definition at line 36 of file scheduler.c.

4.28.2.2 tick_counter

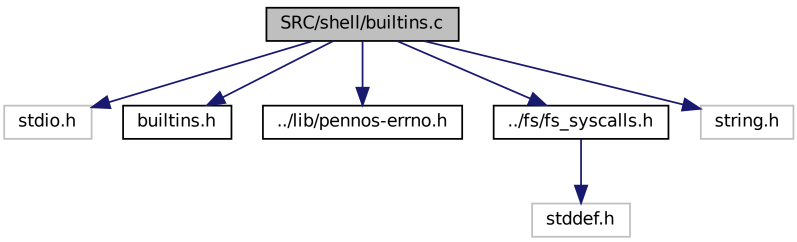
```
int tick_counter  [extern]
```

Definition at line 35 of file scheduler.c.

4.29 SRC/shell/builtins.c File Reference

```
#include <stdio.h>
#include "builtins.h"
#include "../lib/pennos-errno.h"
#include "../fs/fs_syscalls.h"
#include <string.h>
```

Include dependency graph for builtins.c:



Functions

- void u_perror (const char *msg)

4.29.1 Function Documentation

4.29.1.1 u_perror()

```
void u_perror (
            const char * msg )
```

Creates a user-level error message similar to perror.

Parameters

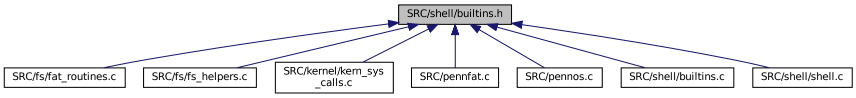
msg	A string representing the error message from the shell.

Definition at line 12 of file builtins.c.

```
                                {
    char buffer[256];
    const char *error_msg;

    switch (P_ERRNO) {
        case P_ENOENT:
            error_msg = "file does not exist";
            break;
        case P_EBADF:
            error_msg = "bad file descriptor";
            break;
        case P_EPERM:
            error_msg = "operation not permitted";
            break;
        case P_EINVAL:
            error_msg = "invalid arg";
            break;
        case P_EEXIST:
            error_msg = "file already exists";
            break;
        case P_EBUSY:
            error_msg = "file is busy or open";
            break;
        case P_EFULL:
            error_msg = "no space left on device";
            break;
        case P_EINTR:
            error_msg = "interrupted system call";
            break;
        case P_ENULL:
            error_msg = "NULL returned unexpectedly";
            break;
        case P_EUNKNOWN:
            error_msg = "unknown error";
            break;
        case P_EREAD:
            error_msg = "interrupted read call";
            break;
        case P_ELSEEK:
            error_msg = "interrupted lseek call";
            break;
        case P_EMAP:
            error_msg = "interrupted mmap/munmap call";
            break;
        case P_EFUNC:
            error_msg = "interrupted system call";
            break;
        case P_EOPEN:
            error_msg = "interrupted open call";
            break;
        case P_EMALLOC:
            error_msg = "error when trying to malloc";
            break;
        case P_EFS_NOT_MOUNTED:
            error_msg = "file system not mounted yet";
            break;
        case P_ESIGNAL:
            error_msg = "error with signal handling";
            break;
        case P_EWRITE:
            error_msg = "interrupted write call";
            break;
        case P_ECLOSE:
            error_msg = "interrupted close call";
            break;
        case P_EPARSE:
            error_msg = "error when trying to parse a command";
            break;
        case P_ECOMMAND:
            error_msg = "command not found";
            break;
        case P_NEEDF:
            error_msg = "no file provided to mount";
            break;
        default:
            error_msg = "Unknown error";
            break;
    }

    snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
    s_write(STDERR_FILENO, buffer, strlen(buffer));
}
```

4.30 SRC/shell/builtins.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void u_perror (const char *msg)

4.30.1 Function Documentation

4.30.1.1 u_perror()

```
void u_perror (
               const char * msg )
```

Creates a user-level error message similar to perror.

Parameters

msg	A string representing the error message from the shell.

Definition at line 12 of file builtins.c.

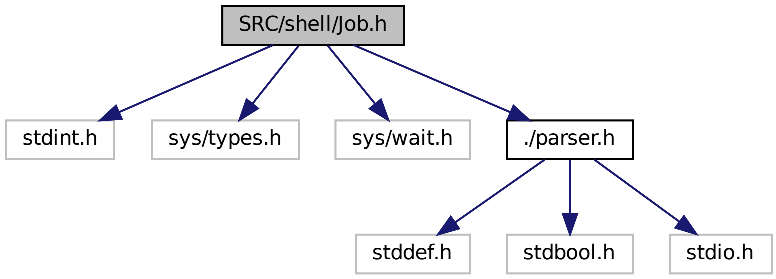
```
12                              {
13     char buffer[256];
14     const char *error_msg;
15 
16     switch (P_ERRNO) {
17         case P_ENOENT:
18             error_msg = "file does not exist";
19             break;
20         case P_EBADF:
21             error_msg = "bad file descriptor";
22             break;
23         case P_EPERM:
24             error_msg = "operation not permitted";
25             break;
26         case P_EINVAL:
27             error_msg = "invalid arg";
28             break;
29         case P_EEXIST:
30             error_msg = "file already exists";
31             break;
32         case P_EBUSY:
33             error_msg = "file is busy or open";
34             break;
35         case P_EFULL:
36             error_msg = "no space left on device";
37             break;
38         case P_EINTR:
39             error_msg = "interrupted system call";
```

```
            break;
        case P_ENULL:
            error_msg = "NULL returned unexpectedly";
            break;
        case P_EUNKNOWN:
            error_msg = "unknown error";
            break;
        case P_EREAD:
            error_msg = "interrupted read call";
            break;
        case P_ELSEEK:
            error_msg = "interrupted lseek call";
            break;
        case P_EMAP:
            error_msg = "interrupted mmap/munmap call";
            break;
        case P_EFUNC:
            error_msg = "interrupted system call";
            break;
        case P_EOPEN:
            error_msg = "interrupted open call";
            break;
        case P_EMALLOC:
            error_msg = "error when trying to malloc";
            break;
        case P_EFS_NOT_MOUNTED:
            error_msg = "file system not mounted yet";
            break;
        case P_ESIGNAL:
            error_msg = "error with signal handling";
            break;
        case P_EWRITE:
            error_msg = "interrupted write call";
            break;
        case P_ECLOSE:
            error_msg = "interrupted close call";
            break;
        case P_EPARSE:
            error_msg = "error when trying to parse a command";
            break;
        case P_ECOMMAND:
            error_msg = "command not found";
            break;
        case P_NEEDF:
            error_msg = "no file provided to mount";
            break;
        default:
            error_msg = "Unknown error";
            break;
    }

    snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
    s_write(STDERR_FILENO, buffer, strlen(buffer));
}
```

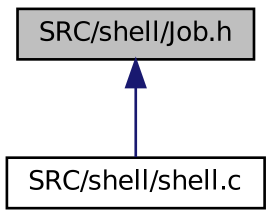
4.31 SRC/shell/Job.h File Reference

```
#include <stdint.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "./parser.h"
```

Include dependency graph for Job.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct job_st

Typedefs

- typedef uint64_t jid_t
- typedef struct job_st job

Enumerations

- enum job_state_t { RUNNING , STOPPED , FINISHED }

4.31.1 Typedef Documentation

4.31.1.1 jid_t

```
typedef uint64_t jid_t
```

Definition at line 10 of file Job.h.

4.31.1.2 job

```
typedef struct job_st job
```

4.31.2 Enumeration Type Documentation

4.31.2.1 job_state_t

```
enum job_state_t
```

Enumerator

RUNNING	
STOPPED	
FINISHED	

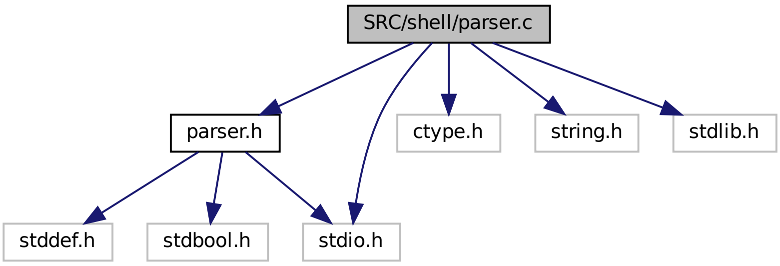
Definition at line 13 of file Job.h.

```
13 { RUNNING, STOPPED, FINISHED } job_state_t;
```

4.32 SRC/shell/parser.c File Reference

```
#include "parser.h"
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

Include dependency graph for parser.c:



Macros

- #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)

Functions

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

4.32.1 Macro Definition Documentation

4.32.1.1 JUMP_OUT

```
#define JUMP_OUT(
              code ) do {ret_code = code; goto PROCESS_ERROR;} while (0)
```

4.32.2 Function Documentation

4.32.2.1 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. layout of memory for `struct parsed_command` bool is_background; bool is_file_append;

const char *stdin_file; const char *stdout_file;

size_t num_commands;

commands are pointers to `arguments` char **commands[num_commands];

below are hidden in memory **

arguments are pointers to `original_string + num_commands` because all argv are null-terminated char *arguments[total_strings + num_commands];

original_string is a copy of the cmdline but with each token null-terminated char *original_string;

Definition at line 16 of file parser.c.

```
16                                                                                   {
17 #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)
18
19     int ret_code = -1;
20
21     const char *start = cmd_line;
22     const char *end = cmd_line + strlen(cmd_line);
23
24     for (const char *cur = start; cur < end; ++cur)
25         if (*cur == '#') {
26             // all subsequent characters following '#'
27             // shall be discarded as a comment.
28             end = cur;
29             break;
30         }
31
32     // trimming leading and trailing whitespaces
33     while (start < end && isspace(*start)) ++start;
34     while (start < end && isspace(end[-1])) --end;
35
36     struct parsed_command *pcmd = calloc(1, sizeof(struct parsed_command));
37     if (pcmd == NULL) return -1;
38     if (start == end) goto PROCESS_SUCCESS; // empty line, fast pass
39
40     // If a command is terminated by the control operator ampersand ( '&' ),
41     // the shell shall execute the command in background.
42     if (end[-1] == '&') {
43         pcmd->is_background = true;
44         --end;
```

```
    }

    // first pass, check token
    int total_strings = 0; // number of total arguments
    {
        bool has_token_last = false, has_file_input = false, has_file_output = false;
        const char *skipped;
        for (const char *cur = start; cur < end; skip_space(&cur, end))
            switch (cur[0]) {
                case '&':
                    JUMP_OUT(UNEXPECTED_AMPERSAND); // does not expect anymore ampersand
                case '<':
                    // if already had pipeline or had file input, error
                    if (pcmd->num_commands > 0 || has_file_input) JUMP_OUT(UNEXPECTED_FILE_INPUT);

                    ++cur; // skip '<'
                    skip_space(&cur, end);

                    // test if we indeed have a filename following '<'
                    skipped = cur;
                    skip_word(&skipped, end);
                    if (skipped <= cur) JUMP_OUT(EXPECT_INPUT_FILENAME);

                    // fast-forward to the end of the filename
                    cur = skipped;
                    has_file_input = true;
                    break;
                case '>':
                    // if already had file output, error
                    if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
                    if (cur + 1 < end && cur[1] == '>') { // dealing with '»' append
                        pcmd->is_file_append = true;
                        ++cur;
                    }

                    ++cur; // skip '>'
                    skip_space(&cur, end);

                    // test filename, as the case above
                    skipped = cur;
                    skip_word(&skipped, end);
                    if (skipped <= cur) JUMP_OUT(EXPECT_OUTPUT_FILENAME);

                    // fast-forward to the end of the filename
                    cur = skipped;
                    has_file_output = true;
                    break;
                case '|':
                    // if already had file output but encourter a pipeline, it should
                    // rather be a file output error instead of a pipeline one.
                    if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
                    // if no tokens between two pipelines (or before the first one)
                    // should throw a pipeline error
                    if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
                    has_token_last = false;
                    ++pcmd->num_commands;
                    ++cur; // skip '|'
                    break;
                default:
                    has_token_last = true;
                    ++total_strings;
                    skip_word(&cur, end); // skip that argument
            }

        if (total_strings == 0) {
            // if there are no arguments but has ampersand or file input/output
            // then we have an error
            if (pcmd->is_background || has_file_input || has_file_output)
                JUMP_OUT(EXPECT_COMMANDS);
            // otherwise it's an empty line
            goto PROCESS_SUCCESS;
        }

        // handle edge case where the command ends with a pipeline
        // (not supporting line continuation)
        if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
    }
    ++pcmd->num_commands;


    const size_t start_of_array = offsetof(struct parsed_command, commands) +pcmd->num_commands *
    sizeof(char **);
    const size_t start_of_str = start_of_array + (pcmd->num_commands + total_strings) * sizeof(char *);
    const size_t slen = end - start;

    char *const new_buf = realloc(pcmd, start_of_str + slen + 1);
    if (new_buf == NULL) goto PROCESS_ERROR;
```

```
    pcmd = (struct parsed_command *) new_buf;

    // copy string to the new place
    char *const new_start = memcpy(new_buf + start_of_str, start, slen);

    // second pass, put stuff in
    // no need to check for error anymore
    size_t cur_cmd = 0;
    char **argv_ptr = (char **) (new_buf + start_of_array);

    pcmd->commands[cur_cmd] = argv_ptr;
    for (const char *cur = start; cur < end; skip_space(&cur, end)) {
        switch (cur[0]) {
            case '<':
                ++cur;
                skip_space(&cur, end);
                // store input file name into `stdin_file`
                pcmd->stdin_file = new_start + (cur - start);
                skip_word(&cur, end);
                // at end of the input file name
                new_start[cur - start] = '\0';
                break;
            case '>':
                if (pcmd->is_file_append) ++cur; // skip another '>'
                ++cur;
                skip_space(&cur, end);
                // store output file name into `stdout_file`
                pcmd->stdout_file = new_start + (cur - start);
                skip_word(&cur, end);
                // at end of the output file name
                new_start[cur - start] = '\0';
                break;
            case '|':
                // null-terminate the current argv
                *(argv_ptr++) = NULL;
                // store the next argv head
                pcmd->commands[++cur_cmd] = argv_ptr;
                ++cur;
                break;
            default:
                // at start of the argument string
                // store it into the arguments array
                *(argv_ptr++) = new_start + (cur - start);
                skip_word(&cur, end);
                // at end of the argument string
                new_start[cur - start] = '\0';
        }
    }
    // null-terminate the last argv
    *argv_ptr = NULL;

PROCESS_SUCCESS:
    *result = pcmd;
    return 0;
PROCESS_ERROR:
    free(pcmd);
    return ret_code;
}
```

4.32.2.2 print_parsed_command()

```
void print_parsed_command (
            const struct parsed_command *const cmd )
```

Definition at line 214 of file parser.c.

```
                                                                  {
    for (size_t i = 0; i < cmd->num_commands; ++i) {
        for (char **arguments = cmd->commands[i]; *arguments != NULL; ++arguments)
            printf("%s ", *arguments);

        if (i == 0 && cmd->stdin_file != NULL)
            printf("< %s ", cmd->stdin_file);

        if (i == cmd->num_commands - 1) {
            if (cmd->stdout_file != NULL)
                printf(cmd->is_file_append ? "» %s " : "> %s ", cmd->stdout_file);
        } else printf("| ");
    }
    puts(cmd->is_background ? "&" : "");
}
```

4.32.2.3 print_parser_errcode()

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

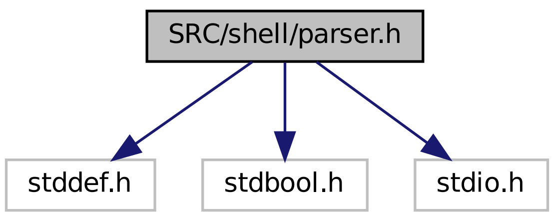
Definition at line 230 of file parser.c.

```
230                                                      {
231   switch (err_code) {
232     case UNEXPECTED_FILE_INPUT:
233       fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234       break;
235     case UNEXPECTED_FILE_OUTPUT:
236       fprintf(output, "UNEXPECTED OUTPUT REDIRECTION TO A FILE\n");
237       break;
238     case UNEXPECTED_PIPELINE:
239       fprintf(output, "UNEXPECTED PIPE\n");
240       break;
241     case UNEXPECTED_AMPERSAND:
242       fprintf(output, "UNEXPECTED AMPERESAND\n");
243       break;
244     case EXPECT_INPUT_FILENAME:
245       fprintf(output, "COULD NOT FINE FILENAME FOR INPUT REDIRECTION \"<\"\n");
246       break;
247     case EXPECT_OUTPUT_FILENAME:
248       fprintf(output, "COULD NOT FIND FILENAME FOR OUTPUT REDIRECTION \"<\"\n");
249       break;
250     case EXPECT_COMMANDS:
251       fprintf(output, "COULD NOT FIND ANY COMMANDS OR ARGS\n");
252       break;
253     default:
254       break;
255   }
256 }
```

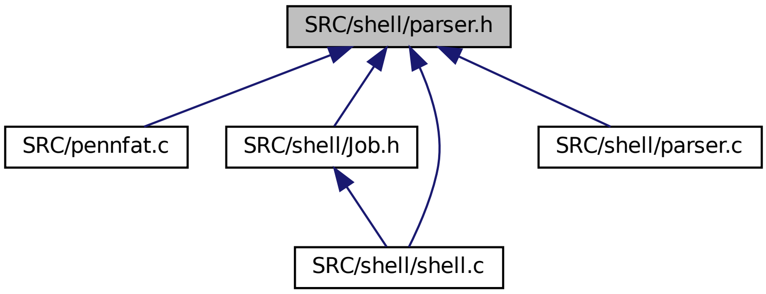
4.33 SRC/shell/parser.h File Reference

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

Include dependency graph for parser.h:



This graph shows which files directly or indirectly include this file:



Classes

- 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 (const struct parsed_command ∗cmd)
- void print_parser_errcode (FILE ∗output, int err_code)

4.33.1 Macro Definition Documentation

4.33.1.1 EXPECT_COMMANDS

```
#define EXPECT_COMMANDS 7
```

Definition at line 30 of file parser.h.

4.33.1.2 EXPECT_INPUT_FILENAME

```
#define EXPECT_INPUT_FILENAME 5
```

Definition at line 24 of file parser.h.

4.33.1.3 EXPECT_OUTPUT_FILENAME

```
#define EXPECT_OUTPUT_FILENAME 6
```

Definition at line 27 of file parser.h.

4.33.1.4 UNEXPECTED_AMPERSAND

```
#define UNEXPECTED_AMPERSAND 4
```

Definition at line 21 of file parser.h.

4.33.1.5 UNEXPECTED_FILE_INPUT

```
#define UNEXPECTED_FILE_INPUT 1
```

Definition at line 12 of file parser.h.

4.33.1.6 UNEXPECTED_FILE_OUTPUT

```
#define UNEXPECTED_FILE_OUTPUT 2
```

Definition at line 15 of file parser.h.

4.33.1.7 UNEXPECTED_PIPELINE

```
#define UNEXPECTED_PIPELINE 3
```

Definition at line 18 of file parser.h.

4.33.2 Function Documentation

4.33.2.1 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. layout of memory for `struct parsed_command` bool is_background; bool is_file_append;

const char *stdin_file; const char *stdout_file;

size_t num_commands;

commands are pointers to `arguments` char **commands[num_commands];

below are hidden in memory **

arguments are pointers to `original_string + num_commands` because all argv are null-terminated char *arguments[total_strings + num_commands];

original_string is a copy of the cmdline but with each token null-terminated char *original_string;

Definition at line 16 of file parser.c.

```
16                                                                                  {
17 #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)
18
19     int ret_code = -1;
20
21     const char *start = cmd_line;
22     const char *end = cmd_line + strlen(cmd_line);
23
24     for (const char *cur = start; cur < end; ++cur)
25         if (*cur == '#') {
26             // all subsequent characters following '#'
27             // shall be discarded as a comment.
28             end = cur;
29             break;
30         }
31
32     // trimming leading and trailing whitespaces
33     while (start < end && isspace(*start)) ++start;
34     while (start < end && isspace(end[-1])) --end;
35
36     struct parsed_command *pcmd = calloc(1, sizeof(struct parsed_command));
```

```
    if (pcmd == NULL) return -1;
    if (start == end) goto PROCESS_SUCCESS; // empty line, fast pass

    // If a command is terminated by the control operator ampersand ( ’&’ ),
    // the shell shall execute the command in background.
    if (end[-1] == ’&’) {
        pcmd->is_background = true;
        --end;
    }

    // first pass, check token
    int total_strings = 0; // number of total arguments
    {
        bool has_token_last = false, has_file_input = false, has_file_output = false;
        const char *skipped;
        for (const char *cur = start; cur < end; skip_space(&cur, end))
            switch (cur[0]) {
                case ’&’:
                    JUMP_OUT(UNEXPECTED_AMPERSAND); // does not expect anymore ampersand
                case ’<’:
                    // if already had pipeline or had file input, error
                    if (pcmd->num_commands > 0 || has_file_input) JUMP_OUT(UNEXPECTED_FILE_INPUT);

                    ++cur; // skip ’<’
                    skip_space(&cur, end);

                    // test if we indeed have a filename following ’<’
                    skipped = cur;
                    skip_word(&skipped, end);
                    if (skipped <= cur) JUMP_OUT(EXPECT_INPUT_FILENAME);

                    // fast-forward to the end of the filename
                    cur = skipped;
                    has_file_input = true;
                    break;
                case ’>’:
                    // if already had file output, error
                    if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
                    if (cur + 1 < end && cur[1] == ’>’) { // dealing with ’»’ append
                        pcmd->is_file_append = true;
                        ++cur;
                    }

                    ++cur; // skip ’>’
                    skip_space(&cur, end);

                    // test filename, as the case above
                    skipped = cur;
                    skip_word(&skipped, end);
                    if (skipped <= cur) JUMP_OUT(EXPECT_OUTPUT_FILENAME);

                    // fast-forward to the end of the filename
                    cur = skipped;
                    has_file_output = true;
                    break;
                case ’|’:
                    // if already had file output but encourter a pipeline, it should
                    // rather be a file output error instead of a pipeline one.
                    if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
                    // if no tokens between two pipelines (or before the first one)
                    // should throw a pipeline error
                    if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
                    has_token_last = false;
                     ++pcmd->num_commands;
                     ++cur; // skip ’|’
                     break;
                 default:
                     has_token_last = true;
                     ++total_strings;
                     skip_word(&cur, end); // skip that argument
             }

         if (total_strings == 0) {
             // if there are no arguments but has ampersand or file input/output
             // then we have an error
             if (pcmd->is_background || has_file_input || has_file_output)
                 JUMP_OUT(EXPECT_COMMANDS);
             // otherwise it’s an empty line
             goto PROCESS_SUCCESS;
         }

         // handle edge case where the command ends with a pipeline
         // (not supporting line continuation)
         if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
     }
     ++pcmd->num_commands;

```

```

    const size_t start_of_array = offsetof(struct parsed_command, commands) +pcmd->num_commands * sizeof(char **);
    const size_t start_of_str = start_of_array + (pcmd->num_commands + total_strings) * sizeof(char *);
    const size_t slen = end - start;

    char *const new_buf = realloc(pcmd, start_of_str + slen + 1);
    if (new_buf == NULL) goto PROCESS_ERROR;
    pcmd = (struct parsed_command *) new_buf;

    // copy string to the new place
    char *const new_start = memcpy(new_buf + start_of_str, start, slen);

    // second pass, put stuff in
    // no need to check for error anymore
    size_t cur_cmd = 0;
    char **argv_ptr = (char **) (new_buf + start_of_array);

    pcmd->commands[cur_cmd] = argv_ptr;
    for (const char *cur = start; cur < end; skip_space(&cur, end)) {
        switch (cur[0]) {
            case '<':
                ++cur;
                skip_space(&cur, end);
                // store input file name into `stdin_file`
                pcmd->stdin_file = new_start + (cur - start);
                skip_word(&cur, end);
                // at end of the input file name
                new_start[cur - start] = '\0';
                break;
            case '>':
                if (pcmd->is_file_append) ++cur; // skip another '>'
                ++cur;
                skip_space(&cur, end);
                // store output file name into `stdout_file`
                pcmd->stdout_file = new_start + (cur - start);
                skip_word(&cur, end);
                // at end of the output file name
                new_start[cur - start] = '\0';
                break;
            case '|':
                // null-terminate the current argv
                *(argv_ptr++) = NULL;
                // store the next argv head
                pcmd->commands[++cur_cmd] = argv_ptr;
                ++cur;
                break;
            default:
                // at start of the argument string
                // store it into the arguments array
                *(argv_ptr++) = new_start + (cur - start);
                skip_word(&cur, end);
                // at end of the argument string
                new_start[cur - start] = '\0';
        }
    }
    // null-terminate the last argv
    *argv_ptr = NULL;

PROCESS_SUCCESS:
    *result = pcmd;
    return 0;
PROCESS_ERROR:
    free(pcmd);
    return ret_code;
}
```

4.33.2.2 print_parsed_command()

```
void print_parsed_command (
            const struct parsed_command * cmd )
```

Definition at line 214 of file parser.c.

```
{
    for (size_t i = 0; i < cmd->num_commands; ++i) {
        for (char **arguments = cmd->commands[i]; *arguments != NULL; ++arguments)
            printf("%s ", *arguments);

```

```
        if (i == 0 && cmd->stdin_file != NULL)
            printf("< %s ", cmd->stdin_file);

        if (i == cmd->num_commands - 1) {
            if (cmd->stdout_file != NULL)
                printf(cmd->is_file_append ? "» %s " : "> %s ", cmd->stdout_file);
        } else printf("| ");
    }
    puts(cmd->is_background ? "&" : "");
}
```

4.33.2.3 print_parser_errcode()

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

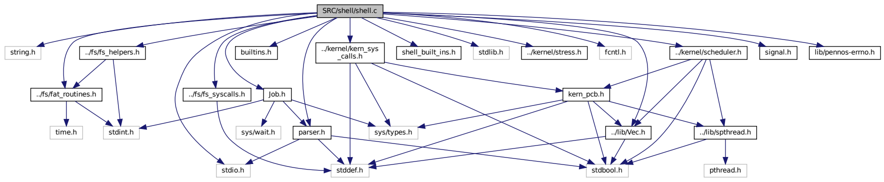
Definition at line 230 of file parser.c.

```
230                                                         {
231   switch (err_code) {
232     case UNEXPECTED_FILE_INPUT:
233       fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234       break;
235     case UNEXPECTED_FILE_OUTPUT:
236       fprintf(output, "UNEXPECTED OUTPUT REDIRECTION TO A FILE\n");
237       break;
238     case UNEXPECTED_PIPELINE:
239       fprintf(output, "UNEXPECTED PIPE\n");
240       break;
241     case UNEXPECTED_AMPERSAND:
242       fprintf(output, "UNEXPECTED AMPERESAND\n");
243       break;
244     case EXPECT_INPUT_FILENAME:
245       fprintf(output, "COULD NOT FINE FILENAME FOR INPUT REDIRECTION \"<\"\n");
246       break;
247     case EXPECT_OUTPUT_FILENAME:
248       fprintf(output, "COULD NOT FIND FILENAME FOR OUTPUT REDIRECTION \"<\"\n");
249       break;
250     case EXPECT_COMMANDS:
251       fprintf(output, "COULD NOT FIND ANY COMMANDS OR ARGS\n");
252       break;
253     default:
254       break;
255   }
256 }
```

4.34 SRC/shell/shell.c File Reference

```
#include <string.h>
#include "../fs/fat_routines.h"
#include "../fs/fs_syscalls.h"
#include "../fs/fs_helpers.h"
#include "../kernel/kern_sys_calls.h"
#include "builtins.h"
#include "parser.h"
#include "shell_built_ins.h"
#include "stdlib.h"
#include "../kernel/stress.h"
#include <fcntl.h>
#include "../kernel/scheduler.h"
#include "../lib/Vec.h"
#include "Job.h"
#include "signal.h"
#include "lib/pennos-errno.h"
```

`#include "stdio.h"`
Include dependency graph for shell.c:



Macros

- #define PROMPT "$ "
- #define MAX_BUFFER_SIZE 4096
- #define MAX_LINE_BUFFER_SIZE 128

Functions

- void shell_sigint_handler (int sig)
- void shell_sigstp_handler (int sig)
- void setup_terminal_signal_handlers (void)
- void free_job_ptr (void *ptr)
- void fill_buffer_until_full_or_newline (int fd, char *buffer)

  Helper function that fills a buffer with characters read from a given file descriptor until the buffer is full (rare and impractical case), a newline is encountered, or EOF is reached.
- pid_t u_execute_command (struct parsed_command *cmd)

  Helper function that will execute a given command so long as it's one of the built-ins. Notably, its output and input are determined by the spawning script process.
- void * u_read_and_execute_script (void *arg)

  Helper function that reads a script file line by line, parses each line as a command, and executes it.
- pid_t execute_command (struct parsed_command *cmd)

  Helper function to execute a parsed command from the shell. In particular, it spawns a child process to execute the command if the built-in should run as a separate process. Otherwise, it just calls the subroutine directly.
- void * shell (void *)

  The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

Variables

- pid_t current_fg_pid
- Vec job_list
- jid_t next_job_id = 1
- int script_fd = -1
- int input_fd_script = -1
- int output_fd_script = -1

4.34.1 Macro Definition Documentation

4.34.1.1 MAX_BUFFER_SIZE

```
#define MAX_BUFFER_SIZE 4096
```

Definition at line 29 of file shell.c.

4.34.1.2 MAX_LINE_BUFFER_SIZE

```
#define MAX_LINE_BUFFER_SIZE 128
```

Definition at line 30 of file shell.c.

4.34.1.3 PROMPT

```
#define PROMPT "$ "
```

Definition at line 26 of file shell.c.

4.34.2 Function Documentation

4.34.2.1 execute_command()

```
pid_t execute_command (
            struct parsed_command * cmd )
```

Helper function to execute a parsed command from the shell. In particular, it spawns a child process to execute the command if the built-in should run as a separate process. Otherwise, it just calls the subroutine directly.

Parameters

cmd	the parsed command to execute, assumed non-null

Returns

the created child id on successful spawn, 0 on successful subroutine call, -1 when nothing was called

Definition at line 257 of file shell.c.

```
257                                                 {
258
259   // setup fds
260   int input_fd = STDIN_FILENO; // standard fds
261   int output_fd = STDOUT_FILENO;
262
```

```
if (cmd->stdin_file != NULL) {
  input_fd = s_open(cmd->stdin_file, F_READ);
  if (input_fd < 0) {
    input_fd = STDIN_FILENO; // reset to default
  }
}

if (cmd->is_file_append) {
  output_fd = s_open(cmd->stdout_file, F_APPEND);
} else {
  output_fd = s_open(cmd->stdout_file, F_WRITE);
}
if (output_fd < 0) {
  output_fd = STDOUT_FILENO; // reset to default
}

// check for independently scheduled processes
if (strcmp(cmd->commands[0][0], "cat") == 0) {
  return s_spawn(u_cat, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
  return s_spawn(u_sleep, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "busy") == 0) {
  return s_spawn(u_busy, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "echo") == 0) {
  return s_spawn(u_echo, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "ls") == 0) {
  return s_spawn(u_ls, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
  return s_spawn(u_touch, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "mv") == 0) {
  return s_spawn(u_mv, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "cp") == 0) {
  return s_spawn(u_cp, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
  return s_spawn(u_rm, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
  return s_spawn(u_chmod, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "ps") == 0) {
  return s_spawn(u_ps, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
  return s_spawn(u_kill, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
  return s_spawn(u_zombify, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
  return s_spawn(u_orphanify, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "hang") == 0) {
  return s_spawn(hang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
  return s_spawn(nohang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "recur") == 0) {
  return s_spawn(recur, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "crash") == 0) {
  return s_spawn(crash, cmd->commands[0], input_fd, output_fd);
}

// check for sub-routines
if (strcmp(cmd->commands[0][0], "nice") == 0) {
  u_nice(cmd->commands[0]);
  return 0;
} else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
  u_nice_pid(cmd->commands[0]);
  return 0;
} else if (strcmp(cmd->commands[0][0], "man") == 0) {
  u_man(cmd->commands[0]);
  return 0;
} else if (strcmp(cmd->commands[0][0], "bg") == 0) {
  u_bg(cmd->commands[0]);
  return 0;
} else if (strcmp(cmd->commands[0][0], "fg") == 0) {
  u_fg(cmd->commands[0]);
  return 0;
} else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
  u_jobs(cmd->commands[0]);
  return 0;
} else if (strcmp(cmd->commands[0][0], "logout") == 0) {
  u_logout(cmd->commands[0]);
  return 0;
}

// otherwise, try to run command as a script
int script_fd_open = s_open(cmd->commands[0][0], F_READ);
if (script_fd_open < 0) { // if not a file, just move on
  return -1;
}
if (has_executable_permission(script_fd_open) != 1) {
  s_close(script_fd_open);
  return -1;
```

```
  } else {
    script_fd = script_fd_open; // update global
    input_fd_script = input_fd;
    output_fd_script = output_fd;

    char* script_argv[] = {cmd->commands[0][0], NULL};
    pid_t wait_on = s_spawn(u_read_and_execute_script, script_argv, input_fd,
              output_fd);
    int status;
    s_waitpid(wait_on, &status, false); // wait for script to finish
    script_fd = -1; // reset global
    input_fd_script = STDIN_FILENO;
    output_fd_script = STDOUT_FILENO;
    s_close(script_fd_open);
    return 0;
  }

  return -1; // no matches case
}
```

4.34.2.2 fill_buffer_until_full_or_newline()

```
void fill_buffer_until_full_or_newline (
            int fd,
            char * buffer )
```

Helper function that fills a buffer with characters read from a given file descriptor until the buffer is full (rare and impractical case), a newline is encountered, or EOF is reached.

Parameters

fd	the file descriptor to read from, assumed to be open
buffer	the buffer to fill with characters

Definition at line 113 of file shell.c.

```
113                                                          {
114   int i = 0;
115   char currChar;
116   while (i < MAX_LINE_BUFFER_SIZE - 1) {
117     int bytes_read = s_read(fd, &currChar, 1);
118     if (bytes_read <= 0 || currChar == '\n') { // EOF or newline cases
119       break;
120     }
121     buffer[i] = currChar;
122     i++;
123   }
124   buffer[i] = '\0'; // Null-terminate the string, replaces \n
125 }
```

4.34.2.3 free_job_ptr()

```
void free_job_ptr (
            void * ptr )
```

Definition at line 94 of file shell.c.

```
94                              {
95   job* job_ptr = (job*)ptr;
96   free(job_ptr->pids);
97   free(job_ptr);
98 }
```

4.34.2.4 setup_terminal_signal_handlers()

```
void setup_terminal_signal_handlers (
            void )
```

Definition at line 73 of file shell.c.

```
73                                     {
74   struct sigaction sa_int = {0};
75   sa_int.sa_handler = shell_sigint_handler;
76   sigemptyset(&sa_int.sa_mask);
77   sa_int.sa_flags = SA_RESTART;
78   sigaction(SIGINT, &sa_int, NULL);
79 
80   struct sigaction sa_stp = {0};
81   sa_stp.sa_handler = shell_sigstp_handler;
82   sigemptyset(&sa_stp.sa_mask);
83   sa_stp.sa_flags = SA_RESTART;
84   sigaction(SIGTSTP, &sa_stp, NULL);
85 }
```

4.34.2.5 shell()

```
void* shell (
            void * input )
```

The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

Parameters

input	unused

Definition at line 375 of file shell.c.

```
375                      {
376 
377   job_list = vec_new(0, free_job_ptr);
378 
379   setup_terminal_signal_handlers();
380 
381   while (true) {
382     int status;
383     pid_t child_pid;
384     while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
385       // Child process has completed, no need to do anything special
386       // The s_waitpid function already handles cleanup
387     }
388 
389     // prompt
390     if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {
391         u_perror("prompt write error");
392         break;
393     }
394 
395     // parse user input
396     char buffer[MAX_BUFFER_SIZE];
397     ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
398     if (user_input < 0) {
399       u_perror("shell read error");
400       break;
401     } else if (user_input == 0) {  // EOF case
402       s_shutdown_pennos();
403       break;
404     }
405 
406     buffer[user_input] = '\0';
407     if (buffer[user_input - 1] == '\n') {
408       buffer[user_input - 1] = '\0';
409     }
410 
```

```
    struct parsed_command* cmd = NULL;
    int cmd_parse_res = parse_command(buffer, &cmd);
    if (cmd_parse_res != 0 || cmd == NULL) {
      P_ERRNO = P_EPARSE;
      u_perror("parse_command");
      continue;
    }

    // handle the command
    if (cmd->num_commands == 0) {
      free(cmd);
      continue;
    }

    child_pid = execute_command(cmd);
    if (child_pid < 0) {
      free(cmd);
      continue;
    } else if (child_pid == 0) {
      free(cmd);
      continue;
    }

    // If background, add the process to the job list.
    if (cmd->is_background) {
      // Create a new job entry.
      job* new_job = malloc(sizeof(job));
      if (new_job == NULL) {
        perror("Error: mallocing new_job failed");
        free(cmd);
        continue;
      }
      new_job->id = next_job_id++;
      new_job->pgid = child_pid;  // For single commands, child's pid = pgid.
      new_job->num_pids = 1;
      new_job->pids = malloc(sizeof(pid_t));
      if (new_job->pids == NULL) {
        perror("Error: mallocing new_job->pids failed");
        free(new_job);
        free(cmd);
        continue;
      }
      new_job->pids[0] = child_pid;
      new_job->state = RUNNING;
      new_job->cmd = cmd;  // Retain command info; do not free here.
      new_job->finished_count = 0;
      vec_push_back(&job_list, new_job);

      // Print job control information in the format: "[job_id] child_pid"
      char msg[128];
      snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
               child_pid);
      s_write(STDOUT_FILENO, msg, strlen(msg));
    } else {
      // Foreground execution.
      current_fg_pid = child_pid;
      int status;
      s_waitpid(child_pid, &status, false);
      current_fg_pid = 2;
    }
  }

  vec_destroy(&job_list);
  s_exit();
  return 0;
}
```

4.34.2.6 shell_sigint_handler()

```
void shell_sigint_handler (
              int sig )
```

Definition at line 52 of file shell.c.

```
52                                       {
53   // If there's a foreground process, forward SIGINT (terminate) to it, as long
54   // as it's not the shell current_fg_pid will also never be 1 (INIT)
55   if (current_fg_pid != 2) {
56     s_kill(current_fg_pid, 2);  // P_SIGTERM
```

```
57   }
58
59   s_write(STDOUT_FILENO, "\n", 1);
60 }
```

4.34.2.7 shell_sigstp_handler()

```
void shell_sigstp_handler (
        int sig )
```

Definition at line 63 of file shell.c.

```
63                                    {
64   // If there's a foreground process, forward SIGTSTP (stop) to it
65   if (current_fg_pid != 2) {
66     s_kill(current_fg_pid, 0);  // P_SIGSTOP
67   }
68
69   s_write(STDOUT_FILENO, "\n", 1);
70 }
```

4.34.2.8 u_execute_command()

```
pid_t u_execute_command (
        struct parsed_command * cmd )
```

Helper function that will execute a given command so long as it's one of the built-ins. Notably, its output and input are determined by the spawning script process.

Parameters

cmd	the parsed command to try executing

Returns

the pid of the process if one was spawned, 0 if a routine was run or -1 if not matches found

Definition at line 136 of file shell.c.

```
136                                                    {
137
138   // check for independently scheduled processes
139   if (strcmp(cmd->commands[0][0], "cat") == 0) {
140     return s_spawn(u_cat, cmd->commands[0], input_fd_script, output_fd_script);
141   } else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
142     return s_spawn(u_sleep, cmd->commands[0], input_fd_script, output_fd_script);
143   } else if (strcmp(cmd->commands[0][0], "busy") == 0) {
144     return s_spawn(u_busy, cmd->commands[0], input_fd_script, output_fd_script);
145   } else if (strcmp(cmd->commands[0][0], "echo") == 0) {
146     return s_spawn(u_echo, cmd->commands[0], input_fd_script, output_fd_script);
147   } else if (strcmp(cmd->commands[0][0], "ls") == 0) {
148     return s_spawn(u_ls, cmd->commands[0], input_fd_script, output_fd_script);
149   } else if (strcmp(cmd->commands[0][0], "touch") == 0) {
150     return s_spawn(u_touch, cmd->commands[0], input_fd_script, output_fd_script);
151   } else if (strcmp(cmd->commands[0][0], "mv") == 0) {
152     return s_spawn(u_mv, cmd->commands[0], input_fd_script, output_fd_script);
153   } else if (strcmp(cmd->commands[0][0], "cp") == 0) {
154     return s_spawn(u_cp, cmd->commands[0], input_fd_script, output_fd_script);
155   } else if (strcmp(cmd->commands[0][0], "rm") == 0) {
156     return s_spawn(u_rm, cmd->commands[0], input_fd_script, output_fd_script);
157   } else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
```

```
    return s_spawn(u_chmod, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "ps") == 0) {
    return s_spawn(u_ps, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "kill") == 0) {
    return s_spawn(u_kill, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
    return s_spawn(u_zombify, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
    return s_spawn(u_orphanify, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "hang") == 0) {
    return s_spawn(hang, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
    return s_spawn(nohang, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "recur") == 0) {
    return s_spawn(recur, cmd->commands[0], input_fd_script, output_fd_script);
  } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
    return s_spawn(crash, cmd->commands[0], input_fd_script, output_fd_script);
  }

  // check for sub-routines
  if (strcmp(cmd->commands[0][0], "nice") == 0) {
    u_nice(cmd->commands[0]);
    return 0;
  } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
    u_nice_pid(cmd->commands[0]);
    return 0;
  } else if (strcmp(cmd->commands[0][0], "man") == 0) {
    u_man(cmd->commands[0]);
    return 0;
  } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
    u_bg(cmd->commands[0]);
    return 0;
  } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
    u_fg(cmd->commands[0]);
    return 0;
  } else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
    u_jobs(cmd->commands[0]);
    return 0;
  } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
    u_logout(cmd->commands[0]);
    return 0;
  } else {
    return -1; // no matches, no scripts now
  }

  return 0;
}
```

4.34.2.9 u_read_and_execute_script()

```
void* u_read_and_execute_script (
            void * arg )
```

Helper function that reads a script file line by line, parses each line as a command, and executes it.

Parameters

arg	standard {function name, NULL} args

Returns

NULL

Definition at line 214 of file shell.c.

```
214                                               {
215   // read the script line by line, parse each line, and execute the command
216   while (true) {
217     char buffer[MAX_LINE_BUFFER_SIZE];
218     fill_buffer_until_full_or_newline(script_fd, buffer);
219     if (buffer[0] == '\0') {
```

```
      break; // EOF case
    }

    // parse the command
    struct parsed_command* cmd = NULL;
    int parse_result = parse_command(buffer, &cmd);
    if (parse_result != 0 || cmd == NULL) {
      P_ERRNO = P_EPARSE;
      u_perror("parse_command");
      free(cmd);
    }

    // execute the command
    pid_t child_pid = u_execute_command(cmd);
    if (child_pid > 0) { // if process was spawned, wait for it to finish
      int status;
      s_waitpid(child_pid, &status, false);
    } else if (child_pid < 0) { // nothing spawning so safe to free cmd
      free(cmd);
    }
  }

  s_exit(); // exit the script
  return NULL;
}
```

4.34.3 Variable Documentation

4.34.3.1 current_fg_pid

```
pid_t current_fg_pid [extern]
```

Definition at line 31 of file kern_sys_calls.c.

4.34.3.2 input_fd_script

```
int input_fd_script = -1
```

Definition at line 42 of file shell.c.

4.34.3.3 job_list

```
Vec job_list
```

Definition at line 37 of file shell.c.

4.34.3.4 next_job_id

```
jid_t next_job_id = 1
```

Definition at line 38 of file shell.c.

4.34.3.5 output_fd_script

```
int output_fd_script = -1
```

Definition at line 43 of file shell.c.

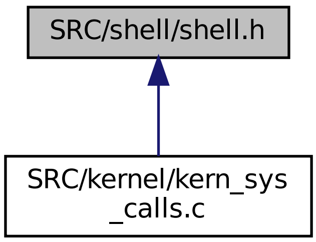
4.34.3.6 script_fd

```
int script_fd = -1
```

Definition at line 41 of file shell.c.

4.35 SRC/shell/shell.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void ∗ shell (void ∗input)

  The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

4.35.1 Function Documentation

4.35.1.1 shell()

```
void* shell (
            void * input )
```

The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

Parameters

input	unused

Definition at line 375 of file shell.c.

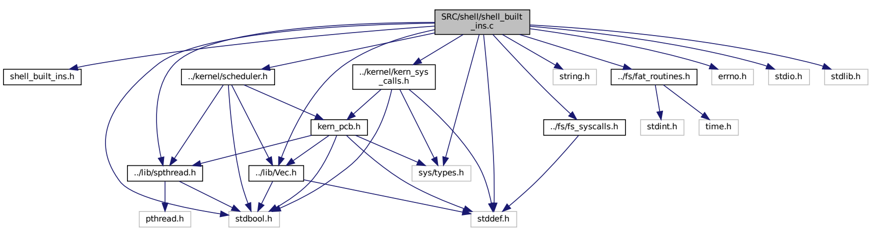
```
375                    {
376 
377   job_list = vec_new(0, free_job_ptr);
378 
379   setup_terminal_signal_handlers();
380 
381   while (true) {
382     int status;
383     pid_t child_pid;
384     while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
385       // Child process has completed, no need to do anything special
386       // The s_waitpid function already handles cleanup
387     }
388 
389     // prompt
390     if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {
391         u_perror("prompt write error");
392         break;
393     }
394 
395     // parse user input
396     char buffer[MAX_BUFFER_SIZE];
397     ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
398     if (user_input < 0) {
399       u_perror("shell read error");
400       break;
401     } else if (user_input == 0) {  // EOF case
402       s_shutdown_pennos();
403       break;
404     }
405 
406     buffer[user_input] = '\0';
407     if (buffer[user_input - 1] == '\n') {
408       buffer[user_input - 1] = '\0';
409     }
410 
411     struct parsed_command* cmd = NULL;
412     int cmd_parse_res = parse_command(buffer, &cmd);
413     if (cmd_parse_res != 0 || cmd == NULL) {
414       P_ERRNO = P_EPARSE;
415       u_perror("parse_command");
416       continue;
417     }
418 
419     // handle the command
420     if (cmd->num_commands == 0) {
421       free(cmd);
422       continue;
423     }
424 
425     child_pid = execute_command(cmd);
426     if (child_pid < 0) {
427       free(cmd);
428       continue;
429     } else if (child_pid == 0) {
430       free(cmd);
431       continue;
432     }
433 
434     // If background, add the process to the job list.
435     if (cmd->is_background) {
436       // Create a new job entry.
437       job* new_job = malloc(sizeof(job));
438       if (new_job == NULL) {
439         perror("Error: mallocing new_job failed");
440         free(cmd);
441         continue;
442       }
443       new_job->id = next_job_id++;
444       new_job->pgid = child_pid;  // For single commands, child's pid = pgid.
445       new_job->num_pids = 1;
446       new_job->pids = malloc(sizeof(pid_t));
447       if (new_job->pids == NULL) {
448         perror("Error: mallocing new_job->pids failed");
449         free(new_job);
450         free(cmd);
451         continue;
452       }
```

```
      new_job->pids[0] = child_pid;
      new_job->state = RUNNING;
      new_job->cmd = cmd;  // Retain command info; do not free here.
      new_job->finished_count = 0;
      vec_push_back(&job_list, new_job);

      // Print job control information in the format: "[job_id] child_pid"
      char msg[128];
      snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
               child_pid);
      s_write(STDOUT_FILENO, msg, strlen(msg));
    } else {
      // Foreground execution.
      current_fg_pid = child_pid;
      int status;
      s_waitpid(child_pid, &status, false);
      current_fg_pid = 2;
    }
  }

  vec_destroy(&job_list);
  s_exit();
  return 0;
}
```

4.36 SRC/shell/shell_built_ins.c File Reference

```
#include "shell_built_ins.h"
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#include <sys/types.h>
#include "../fs/fat_routines.h"
#include "../fs/fs_syscalls.h"
#include "../kernel/kern_sys_calls.h"
#include "../kernel/scheduler.h"
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for shell_built_ins.c:



Functions

- void ∗ u_cat (void ∗arg)

  The usual `cat` *program.*
- void ∗ u_sleep (void ∗arg)

  Sleep for n *seconds.*
- void ∗ u_busy (void ∗arg)

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

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

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

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

  Rename a file. If the `dst_file` file already exists, overwrite it.
- void * u_cp (void *arg)
- void * u_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 * u_ps (void *arg)

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

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

  Adjust the priority level of an existing process.
- void * u_man (void *arg)

  Lists all available commands.
- void * u_bg (void *arg)

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

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

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

  Exits the shell and shutsdown PennOS.
- void * zombie_child (void *arg)

  Helper for zombify.
- void * u_zombify (void *arg)

  Used to test zombifying functionality of your kernel.
- void * orphan_child (void *arg)

  Helper for orphanify.
- void * u_orphanify (void *arg)

  Used to test orphanifying functionality of your kernel.

Variables

- void *(*)(void *) get_associated_ufunc (char *func)

  Helper function to get the associated "u-version" of a function given its standalone version. As a conrete example, if we pass in "cat", we will output "u_cat".

4.36.1 Function Documentation

4.36.1.1 orphan_child()

```
void* orphan_child (
            void * arg )
```

Helper for orphanify.

Definition at line 314 of file shell_built_ins.c.

```
314                                 {
315   while (1)
316     ;
317   s_exit();
318 }
```

4.36.1.2 u_bg()

```
void* u_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)

Definition at line 271 of file shell_built_ins.c.

```
271                         {
272   // TODO --> implement bg
273   return NULL;
274 }
```

4.36.1.3 u_busy()

```
void* u_busy (
            void * arg )
```

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

Example Usage: busy

Definition at line 53 of file shell_built_ins.c.

```
53                           {
54   while (1)
55     ;
56   s_exit();
57   return NULL;
58 }
```

4.36.1.4 u_cat()

```
void* u_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)

Definition at line 28 of file shell_built_ins.c.

```
28                        {
29   cat(arg);
30   s_exit();
31   return NULL;
32 }
```

4.36.1.5 u_chmod()

```
void* u_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)

Definition at line 72 of file shell_built_ins.c.

```
72                          {
73   chmod(arg);
74   s_exit();
75   return NULL;
76 }
```

4.36.1.6 u_cp()

```
void* u_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

Definition at line 90 of file shell_built_ins.c.

```
90                      {
91   cp(arg);
92   s_exit();
93   return NULL;
94 }
```

4.36.1.7 u_echo()

```
void* u_echo (
            void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

Definition at line 60 of file shell_built_ins.c.

```
60                        {
61   s_echo(arg);
62   s_exit();
63   return NULL;
64 }
```

4.36.1.8 u_fg()

```
void* u_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)

Definition at line 276 of file shell_built_ins.c.

```
276                     {
277   // TODO --> implement fg
278   return NULL;
279 }
```

4.36.1.9 u_jobs()

```
void* u_jobs (
            void * arg )
```

Lists all jobs.

Example Usage: jobs

Definition at line 281 of file shell_built_ins.c.

```
281                        {
282   // TODO --> implement jobs
283   return NULL;
284 }
```

4.36.1.10 u_kill()

```
void* u_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)

Definition at line 108 of file shell_built_ins.c.

```
108                        {
109   char** argv = (char**)arg;
110   int sig = 2;             // Default signal: term (2)
111   int start_index = 1;     // Start after the "kill" command word.
112   char err_buf[128];
113 
114   // Check if the first argument specifies a signal
115   if (argv[start_index] && argv[start_index][0] == '-') {
116     if (strcmp(argv[start_index], "-term") == 0) {
117       sig = 2;
118     } else if (strcmp(argv[start_index], "-stop") == 0) {
119       sig = 0;
120     } else if (strcmp(argv[start_index], "-cont") == 0) {
121       sig = 1;
122     } else {
123       // Construct error message
124       s_exit();
125       return NULL;
126     }
127     start_index++;
128   }
129 
130   // Process each PID argument using strtol
131   for (int i = start_index; argv[i] != NULL; i++) {
132     char* endptr;
133     long pid_long = strtol(argv[i], &endptr, 10);
134     if (*endptr != '\0' || pid_long <= 0) {
135       snprintf(err_buf, 128, "Invalid PID: %s\n", argv[i]);
136       s_write(STDERR_FILENO, err_buf, strlen(err_buf));
137       continue;
138     }
139     pid_t pid = (pid_t)pid_long;
140     if (s_kill(pid, sig) < 0) {
141       snprintf(err_buf, 128, "b_kill error on PID %d\n", pid);
142       s_write(STDERR_FILENO, err_buf, strlen(err_buf));
143     }
144   }
145   s_exit();
146   return NULL;
147 }
```

4.36.1.11 u_logout()

```
void* u_logout (
            void * arg )
```

Exits the shell and shutsdown PennOS.

Example Usage: logout

Definition at line 286 of file shell_built_ins.c.

```
286                          {
287   s_shutdown_pennos();
288   return NULL;
289 }
```

4.36.1.12 u_ls()

```
void* u_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)

Definition at line 66 of file shell_built_ins.c.

```
66                       {
67   ls(arg);
68   s_exit();
69   return NULL;
70 }
```

4.36.1.13 u_man()

```
void* u_man (
            void * arg )
```

Lists all available commands.

Example Usage: man

Definition at line 233 of file shell_built_ins.c.

```
233                       {
234   const char* man_string =
235       "cat f1 f2 ...         : concatenates provided files (if none, reads from "
236       "std in), and writes to std out\n"
237       "sleep n                : sleeps for n seconds\n"
238       "busy                   : busy waits indefinitely\n"
239       "echo str               : echoes back the input string str\n"
240       "ls                     : lists all files in the working directory\n"
241       "touch f1 f2 ...        : for each file, creates empty file if it doesn't "
242       "exist yet, otherwise updates its timestamp\n"
243       "mv f1 f2               : renames f1 to f2 (overwrites f2 if it exists)\n"
244       "cp f1 f2               : copies f1 to f2 (overwrites f2 if it exists)\n"
245       "rm f1 f2 ...           : removes the input list of files\n"
246       "chmod +_ f1            : changes f1 permissions to +_ specifications "
247       "(+x, +rw, etc)\n"
248       "ps                     : lists all processes on PennOS, displaying PID, "
249       "PPID, priority, status, and command name\n"
250       "kill (-__) pid1 pid 2 : sends specified signal (term default) to list "
251       "of processes\n"
```

```
252       "nice n command        : spawns a new process for command and sets its "
253       "priority to n\n"
254       "nice_pid n pid        : adjusts the priority level of an existing "
255       "process to n\n"
256       "man                   : lists all available commands in PennOS\n"
257       "bg                    : resumes most recently stopped process in "
258       "background or the one specified by job_id\n"
259       "fg                    : brings most recently stopped or background job "
260       "to foreground or the one specifed by job_id\n"
261       "jobs                  : lists all jobs\n"
262       "logout                : exits the shell and shuts down PennOS\n"
263       "zombify               : creates a child process that becomes a zombie\n"
264       "orphanify             : creates a child process that becomes an "
265       "orphan\n";
266
267   s_write(STDOUT_FILENO, man_string, strlen(man_string));
268   return NULL;
269 }
```

4.36.1.14 u_mv()

```
void* u_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

Definition at line 84 of file shell_built_ins.c.

```
84                        {
85   mv(arg);
86   s_exit();
87   return NULL;
88 }
```

4.36.1.15 u_nice()

```
void* u_nice (
              void * arg )
```

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

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

Definition at line 194 of file shell_built_ins.c.

```
194                         {
195   char* endptr;
196   errno = 0;
197   int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
198   if (*endptr != '\0' || errno != 0 || new_priority > 2 ||
199       new_priority < 0) {  // error catch
200     return NULL;
201   }
```

```
202
203   char* command = ((char**)arg)[2];
204   void* (*ufunc)(void*) = get_associated_ufunc(command);
205   if (ufunc == NULL) {
206     return NULL;  // no matches, don't spawn
207   }
208
209   pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
210
211   if (new_proc_pid != -1) {  // non-error case
212     s_nice(new_proc_pid, new_priority);
213   }
214
215   return NULL;
216 }
```

4.36.1.16 u_nice_pid()

```
void* u_nice_pid (
            void * arg )
```

Adjust the priority level of an existing process.

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

Definition at line 218 of file shell_built_ins.c.

```
218                               {
219   char* endptr;
220   errno = 0;
221   int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
222   if (*endptr != '\0' || errno != 0) {  // error catch
223     return NULL;
224   }
225   pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
226   if (*endptr != '\0' || errno != 0) {
227     return NULL;
228   }
229   s_nice(pid, new_priority);
230   return NULL;
231 }
```

4.36.1.17 u_orphanify()

```
void* u_orphanify (
            void * arg )
```

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Definition at line 320 of file shell_built_ins.c.

```
320                                {
321   char* orphan_child_argv[] = {"orphan_child", NULL};
322   s_spawn(orphan_child, orphan_child_argv, STDIN_FILENO, STDOUT_FILENO);
323   s_exit();
324   return NULL;
325 }
```

4.36.1.18 u_ps()

```
void* u_ps (
            void * arg )
```

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

Example Usage: ps

Definition at line 102 of file shell_built_ins.c.

```
102                          {
103   s_ps(arg);
104   s_exit();
105   return NULL;
106 }
```

4.36.1.19 u_rm()

```
void* u_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

Definition at line 96 of file shell_built_ins.c.

```
96                          {
97   rm(arg);
98   s_exit();
99   return NULL;
100 }
```

4.36.1.20 u_sleep()

```
void* u_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

Definition at line 34 of file shell_built_ins.c.

```
34                             {
35   char* endptr;
36   errno = 0;
37   if (((char**)arg)[1] == NULL) {  // no arg case
38     s_exit();
39     return NULL;
40   }
41   int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
42   if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
43     s_exit();
44     return NULL;
45   }
46
47   int sleep_ticks = sleep_secs * 10;
48   s_sleep(sleep_ticks);
49   s_exit();
50   return NULL;
51 }
```

4.36.1.21 u_touch()

```
void* u_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

Definition at line 78 of file shell_built_ins.c.

```
78                        {
79   touch(arg);
80   s_exit();
81   return NULL;
82 }
```

4.36.1.22 u_zombify()

```
void* u_zombify (
            void * arg )
```

Used to test zombifying functionality of your kernel.

Example Usage: zombify

Definition at line 303 of file shell_built_ins.c.

```
303                          {
304   char* zombie_child_argv[] = {"zombie_child", NULL};
305   s_spawn(zombie_child, zombie_child_argv, STDIN_FILENO, STDOUT_FILENO);
306   while (1)
307     ;
308   return NULL;
309 }
```

4.36.1.23 zombie_child()

```
void* zombie_child (
            void * arg )
```

Helper for zombify.

Definition at line 298 of file shell_built_ins.c.

```
298                             {
299   s_exit();
300   return NULL;
301 }
```

4.36.2 Variable Documentation

4.36.2.1 get_associated_ufunc

```
void*(*)(void*) get_associated_ufunc(char *func) (
            char * func )
```

Helper function to get the associated "u-version" of a function given its standalone version. As a conrete example, if we pass in "cat", we will output "u_cat".

Parameters

func	A string of the function name to get the associated ufunc for

Returns

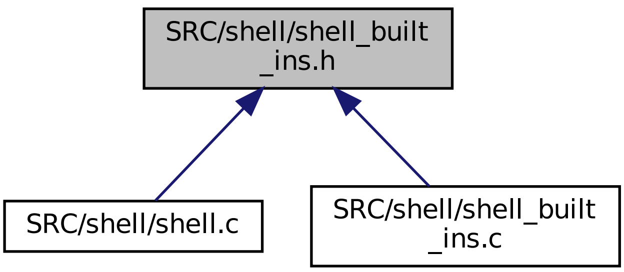
A ptr to the associated u-version function or NULL if no matches are found

Definition at line 164 of file shell_built_ins.c.

```
164                                                   {
165   if (strcmp(func, "cat") == 0) {
166     return u_cat;
167   } else if (strcmp(func, "sleep") == 0) {
168     return u_sleep;
169   } else if (strcmp(func, "busy") == 0) {
170     return u_busy;
171   } else if (strcmp(func, "echo") == 0) {
172     return u_echo;
173   } else if (strcmp(func, "ls") == 0) {
174     return u_ls;
175   } else if (strcmp(func, "touch") == 0) {
176     return u_touch;
177   } else if (strcmp(func, "mv") == 0) {
178     return u_mv;
179   } else if (strcmp(func, "cp") == 0) {
180     return u_cp;
181   } else if (strcmp(func, "rm") == 0) {
182     return u_rm;
183   } else if (strcmp(func, "chmod") == 0) {
184     return u_chmod;
185   } else if (strcmp(func, "ps") == 0) {
186     return u_ps;
187   } else if (strcmp(func, "kill") == 0) {
188     return u_kill;
189   }
190
191   return NULL;  // no matches case
192 }
```

4.37 SRC/shell/shell_built_ins.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void ∗ u_cat (void ∗arg)

    The ususal `cat` program.
- void ∗ u_sleep (void ∗arg)

    Sleep for `n` seconds.
- void ∗ u_busy (void ∗arg)

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

    Echo back an input string.
- void ∗ u_ls (void ∗arg)

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

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

    Rename a file. If the `dst_file` file already exists, overwrite it.
- void ∗ u_cp (void ∗arg)
- void ∗ u_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 ∗ u_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 ∗ u_ps (void ∗arg)

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

    Spawn a new process for `command` and set its priority to `priority`.
- void ∗ u_nice_pid (void ∗arg)

    Adjust the priority level of an existing process.
- void ∗ u_man (void ∗arg)

    Lists all available commands.
- void ∗ u_bg (void ∗arg)

    Resumes the most recently stopped job in the background, or the job specified by job_id.
- void ∗ u_fg (void ∗arg)

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

    Lists all jobs.
- void ∗ u_logout (void ∗arg)

    Exits the shell and shutsdown PennOS.
- void ∗ u_zombify (void ∗arg)

    Used to test zombifying functionality of your kernel.
- void ∗ u_orphanify (void ∗arg)

    Used to test orphanifying functionality of your kernel.

4.37.1 Function Documentation

4.37.1.1 u_bg()

```
void* u_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)

Definition at line 271 of file shell_built_ins.c.

```
271                     {
272   // TODO --> implement bg
273   return NULL;
274 }
```

4.37.1.2 u_busy()

```
void* u_busy (
            void * arg )
```

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

Example Usage: busy

Definition at line 53 of file shell_built_ins.c.

```
53                        {
54   while (1)
55     ;
56   s_exit();
57   return NULL;
58 }
```

4.37.1.3 u_cat()

```
void* u_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)

Definition at line 28 of file shell_built_ins.c.

```
28                       {
29   cat(arg);
30   s_exit();
31   return NULL;
32 }
```

4.37.1.4 u_chmod()

```
void* u_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)

Definition at line 72 of file shell_built_ins.c.

```
72                          {
73   chmod(arg);
74   s_exit();
75   return NULL;
76 }
```

4.37.1.5 u_cp()

```
void* u_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

Definition at line 90 of file shell_built_ins.c.

```
90                       {
91   cp(arg);
92   s_exit();
93   return NULL;
94 }
```

4.37.1.6 u_echo()

```
void* u_echo (
            void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

Definition at line 60 of file shell_built_ins.c.

```
60                        {
61    s_echo(arg);
62    s_exit();
63    return NULL;
64 }
```

4.37.1.7 u_fg()

```
void* u_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)

Definition at line 276 of file shell_built_ins.c.

```
276                       {
277    // TODO --> implement fg
278    return NULL;
279 }
```

4.37.1.8 u_jobs()

```
void* u_jobs (
            void * arg )
```

Lists all jobs.

Example Usage: jobs

Definition at line 281 of file shell_built_ins.c.

```
281                         {
282    // TODO --> implement jobs
283    return NULL;
284 }
```

4.37.1.9 u_kill()

```
void* u_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)

Definition at line 108 of file shell_built_ins.c.

```
108                        {
109   char** argv = (char**)arg;
110   int sig = 2;           // Default signal: term (2)
111   int start_index = 1;  // Start after the "kill" command word.
112   char err_buf[128];
113
114   // Check if the first argument specifies a signal
115   if (argv[start_index] && argv[start_index][0] == '-') {
116     if (strcmp(argv[start_index], "-term") == 0) {
117       sig = 2;
118     } else if (strcmp(argv[start_index], "-stop") == 0) {
119       sig = 0;
120     } else if (strcmp(argv[start_index], "-cont") == 0) {
121       sig = 1;
122     } else {
123       // Construct error message
124       s_exit();
125       return NULL;
126     }
127     start_index++;
128   }
129
130   // Process each PID argument using strtol
131   for (int i = start_index; argv[i] != NULL; i++) {
132     char* endptr;
133     long pid_long = strtol(argv[i], &endptr, 10);
134     if (*endptr != '\0' || pid_long <= 0) {
135       snprintf(err_buf, 128, "Invalid PID: %s\n", argv[i]);
136       s_write(STDERR_FILENO, err_buf, strlen(err_buf));
137       continue;
138     }
139     pid_t pid = (pid_t)pid_long;
140     if (s_kill(pid, sig) < 0) {
141       snprintf(err_buf, 128, "b_kill error on PID %d\n", pid);
142       s_write(STDERR_FILENO, err_buf, strlen(err_buf));
143     }
144   }
145   s_exit();
146   return NULL;
147 }
```

4.37.1.10 u_logout()

```
void* u_logout (
            void * arg )
```

Exits the shell and shutsdown PennOS.

Example Usage: logout

Definition at line 286 of file shell_built_ins.c.

```
286                          {
287   s_shutdown_pennos();
288   return NULL;
289 }
```

4.37.1.11 u_ls()

```
void* u_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)

Definition at line 66 of file shell_built_ins.c.

```
66                    {
67    ls(arg);
68    s_exit();
69    return NULL;
70 }
```

4.37.1.12 u_man()

```
void* u_man (
            void * arg )
```

Lists all available commands.

Example Usage: man

Definition at line 233 of file shell_built_ins.c.

```
233                       {
234    const char* man_string =
235        "cat f1 f2 ...         : concatenates provided files (if none, reads from "
236        "std in), and writes to std out\n"
237        "sleep n                : sleeps for n seconds\n"
238        "busy                   : busy waits indefinitely\n"
239        "echo str               : echoes back the input string str\n"
240        "ls                     : lists all files in the working directory\n"
241        "touch f1 f2 ...        : for each file, creates empty file if it doesn't "
242        "exist yet, otherwise updates its timestamp\n"
243        "mv f1 f2               : renames f1 to f2 (overwrites f2 if it exists)\n"
244        "cp f1 f2               : copies f1 to f2 (overwrites f2 if it exists)\n"
245        "rm f1 f2 ...           : removes the input list of files\n"
246        "chmod +_ f1            : changes f1 permissions to +_ specifications "
247        "(+x, +rw, etc)\n"
248        "ps                     : lists all processes on PennOS, displaying PID, "
249        "PPID, priority, status, and command name\n"
250        "kill (-__) pid1 pid 2 : sends specified signal (term default) to list "
251        "of processes\n"
252        "nice n command         : spawns a new process for command and sets its "
253        "priority to n\n"
254        "nice_pid n pid         : adjusts the priority level of an existing "
255        "process to n\n"
256        "man                    : lists all available commands in PennOS\n"
257        "bg                     : resumes most recently stopped process in "
258        "background or the one specified by job_id\n"
259        "fg                     : brings most recently stopped or background job "
260        "to foreground or the one specifed by job_id\n"
261        "jobs                   : lists all jobs\n"
262        "logout                 : exits the shell and shuts down PennOS\n"
263        "zombify                : creates a child process that becomes a zombie\n"
264        "orphanify              : creates a child process that becomes an "
265        "orphan\n";
266
267    s_write(STDOUT_FILENO, man_string, strlen(man_string));
268    return NULL;
269 }
```

4.37.1.13 u_mv()

```
void* u_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

Definition at line 84 of file shell_built_ins.c.

```
84                        {
85   mv(arg);
86   s_exit();
87   return NULL;
88 }
```

4.37.1.14 u_nice()

```
void* u_nice (
            void * arg )
```

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

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

Definition at line 194 of file shell_built_ins.c.

```
194                         {
195   char* endptr;
196   errno = 0;
197   int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
198   if (*endptr != '\0' || errno != 0 || new_priority > 2 ||
199       new_priority < 0) {  // error catch
200     return NULL;
201   }
202
203   char* command = ((char**)arg)[2];
204   void* (*ufunc)(void*) = get_associated_ufunc(command);
205   if (ufunc == NULL) {
206     return NULL;  // no matches, don't spawn
207   }
208
209   pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
210
211   if (new_proc_pid != -1) {  // non-error case
212     s_nice(new_proc_pid, new_priority);
213   }
214
215   return NULL;
216 }
```

4.37.1.15 u_nice_pid()

```
void* u_nice_pid (
            void * arg )
```

Adjust the priority level of an existing process.

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

Definition at line 218 of file shell_built_ins.c.

```
218                                 {
219   char* endptr;
220   errno = 0;
221   int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
222   if (*endptr != '\0' || errno != 0) {  // error catch
223     return NULL;
224   }
225   pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
226   if (*endptr != '\0' || errno != 0) {
227     return NULL;
228   }
229   s_nice(pid, new_priority);
230   return NULL;
231 }
```

4.37.1.16 u_orphanify()

```
void* u_orphanify (
            void * arg )
```

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Definition at line 320 of file shell_built_ins.c.

```
320                                  {
321   char* orphan_child_argv[] = {"orphan_child", NULL};
322   s_spawn(orphan_child, orphan_child_argv, STDIN_FILENO, STDOUT_FILENO);
323   s_exit();
324   return NULL;
325 }
```

4.37.1.17 u_ps()

```
void* u_ps (
            void * arg )
```

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

Example Usage: ps

Definition at line 102 of file shell_built_ins.c.

```
102                           {
103   s_ps(arg);
104   s_exit();
105   return NULL;
106 }
```

4.37.1.18 u_rm()

```
void* u_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

Definition at line 96 of file shell_built_ins.c.

```
96                      {
97   rm(arg);
98   s_exit();
99   return NULL;
100 }
```

4.37.1.19 u_sleep()

```
void* u_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

Definition at line 34 of file shell_built_ins.c.

```
34                         {
35   char* endptr;
36   errno = 0;
37   if (((char**)arg)[1] == NULL) {  // no arg case
38     s_exit();
39     return NULL;
40   }
41   int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
42   if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
43     s_exit();
44     return NULL;
45   }
46
47   int sleep_ticks = sleep_secs * 10;
48   s_sleep(sleep_ticks);
49   s_exit();
50   return NULL;
51 }
```

4.37.1.20 u_touch()

```
void* u_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

Definition at line 78 of file shell_built_ins.c.

```
78                        {
79   touch(arg);
80   s_exit();
81   return NULL;
82 }
```

4.37.1.21 u_zombify()

```
void* u_zombify (
            void * arg )
```

Used to test zombifying functionality of your kernel.

Example Usage: zombify

Definition at line 303 of file shell_built_ins.c.

```
303                          {
304   char* zombie_child_argv[] = {"zombie_child", NULL};
305   s_spawn(zombie_child, zombie_child_argv, STDIN_FILENO, STDOUT_FILENO);
306   while (1)
307     ;
308   return NULL;
309 }
```

Index